

Potree: Rendering Large Point Clouds in Web Browsers

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Markus Schuetz

Matrikelnummer 0825723

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Michael Wimmer

Wien, 1. Juni 2015

Markus Schuetz

Michael Wimmer

Potree: Rendering Large Point Clouds in Web Browsers

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Markus Schuetz

Registration Number 0825723

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Michael Wimmer

Vienna, 1st June, 2015

Markus Schuetz

Michael Wimmer

Erklärung zur Verfassung der Arbeit

Markus Schuetz
Alszeile 78/6, 1170 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Juni 2015

Markus Schuetz

Acknowledgements

I would like to thank anyone who has contributed to Potree and this thesis in one way or another. To my advisor, Michael Wimmer, for his help with this thesis and his abnormally fast response times, even on weekends. To Claus Scheiblauber as well as Michael Wimmer for their previous works, on which this thesis is based.

Thanks to Daniel Kastl from Georepublic and Martin Isenburg from rapidlasso, not only for their funding but also for their extensive help in the development process.

Thanks to Ricardo Cabello, developer and maintainer of the three.js WebGL rendering library. Three.js made it possible to leave low-level graphics programming aside and concentrate on the actual target, the processing and rendering of large data sets.

Thanks to Howard Butler, Connor Manning, and Uday Verma, who also contributed to the state-of-the-art in browser based point cloud rendering and made their work publicly available.

Thanks to Christian Boucheny, developer of the Eye-Dome Lighting illumination model, and Daniel Girardeau-Montaut, developer of the CloudCompare point cloud viewer, for making their amazing contributions to point cloud rendering public and open source.

Thanks to the *Ludwig Boltzmann Institute for Archaeological Prospection and Virtual Archaeology* for the Heidendor point cloud and Riegl for the Retz point cloud, which are used for the cover pages of each chapter, as well as anyone else who provided point clouds that are used throughout this thesis.

Special thanks to family and friends, who had to endure a lot of talks about point clouds.

Potree was funded by rapidlasso, Georepublic, Veesus, sigeom sa, sitn (ne.ch), the Ludwig Boltzmann Institute Archaeological Prospection and Virtual Archaeology, and Pix4D.

This research was supported by the EU FP7 project HARVEST4D(no. 323567) [23].

Kurzfassung

Im Zuge dieser Arbeit stellen wir einen Punktwolkenrederer namens Potree vor, der es ermöglicht riesige Datensätze mit Milliarden von Punkten, wie sie zum Beispiel durch LIDAR Scanner oder Photogrammetrie Software entstehen, in einem Web Browser zu betrachten.

Einer der Vorteile davon Punktwolken in Web Browsern zu rendern ist, dass es Benutzern erlaubt ihre Datensätze mit Partnern oder der Öffentlichkeit zu teilen, ohne dass diese erst große Datenmengen herunterladen oder eine Drittanwendung installieren müssen. Der Fokus auf große Datensätze und die zahlreichen Messwerkzeuge erlaubt es Benutzern ausserdem, aufgenommene Datensätze zu analysieren und Messungen durchzuführen, ohne dass die Daten vorher in einem kosten- und zeitintensiven Schritt in Dreiecksmodelle umgewandelt werden müssen.

Das Laden und Rendern von Milliarden von Punkten in Potree wird durch eine hierarchische Datenstruktur ermöglicht, in der Teilmengen der Punktwolke in unterschiedlichen Auflösungen gespeichert werden. Teilmengen mit geringer Dichte werden im Root Node gespeichert. Mit jedem weiteren Level erhöht sich die Dichte, und somit der Detailgrad, der Daten die in den Nodes abgespeichert werden. Durch diese Struktur kann Potree sich auf diejenigen Teile der Punktwolke beschränken die für einen gegebenen Blickwinkel am wichtigsten sind. Punkte die sich nicht im Blickfeld befinden werden übersprungen, und für entfernte Bereiche wird eine niedrigere Detailstufe aus geladen und gerendert.

Das Endergebnis dieser Arbeit ist ein open source und web-basierter Punktwolken-renderer, der erfolgreich mit Datensätzen bis zu 597 Milliarden Punkten getestet wurde.

Abstract

This thesis introduces Potree, a web-based renderer for large point clouds. It allows users to view data sets with billions of points, from sources such as LIDAR or photogrammetry, in real time in standard web browsers.

One of the main advantages of point cloud visualization in web browser is that it allows users to share their data sets with clients or the public without the need to install third-party applications and transfer huge amounts of data in advance. The focus on large point clouds, and a variety of measuring tools, also allows users to use Potree to look at, analyze and validate raw point cloud data, without the need for a time-intensive and potentially costly meshing step.

The streaming and rendering of billions of points in web browsers, without the need to load large amounts of data in advance, is achieved with a hierarchical structure that stores subsamples of the original data at different resolutions. A low resolution is stored in the root node and with each level, the resolution gradually increases. The structure allows Potree to cull regions of the point cloud that are outside the view frustum, and to render distant regions at a lower level of detail.

The result is an open source point cloud viewer, which was able to render point cloud data sets of up to 597 billion points, roughly 1.6 terabytes after compression, in real time in a web browser.

Contents

Kurzfassung	ix
Abstract	xi
Contents	xiii
1 Introduction	1
1.1 Motivation	2
1.2 Problem Definition	2
1.3 Contributions	3
1.4 Structure of the Work	5
2 Related Work	7
2.1 Rendering Massive Point Clouds	7
2.2 Web-Based Massive Point Cloud and Voxel Rendering	9
2.3 Desktop-Based Massive Point Cloud And Voxel Rendering	9
2.4 High-Quality Point-Based Rendering	10
3 Data Structure	13
3.1 Overview	14
3.2 Modifiable Nested Octree	14
3.3 Potree's Octree Structure	15
3.4 Octree Traversal and Visible Node Determination	21
4 Point Cloud Rendering	25
4.1 Point Attribute Coloring	26
4.2 Point Splatting	27
4.3 Determining Point Sizes	33
4.4 Eye-Dome Lighting	38
5 Implementation and Features	43
5.1 WebGL	44
5.2 Asynchronous and Parallel Execution	45
5.3 Tools and Interaction	47

5.4	Georeferencing	56
5.5	Data storage	59
6	Results	63
6.1	Performance	64
6.2	Applications	67
6.3	Showcase	71
7	Conclusion and Future Work	77
	Bibliography	79

CHAPTER 1

Introduction



1.1 Motivation

Point clouds are three-dimensional models that consist of points rather than the more widely-used triangle models. They are most commonly obtained as a result of scanning the real world through various scanning methods, such as laser scanning and photogrammetry. Use cases include the generation of three-dimensional maps and globes (e.g. Google Maps, Cesium), keeping track of building progress or changes in urban, forest or other types of landscapes, the generation of three-dimensional assets for games and movies, or to capture movement and poses (e.g. Kinect). In many use cases, the point clouds are treated as raw data, which is then refined by converting it into triangle models or two-dimensional images.

In order to present these models to clients or an interested audience, it was traditionally necessary to transfer large amounts of data and to install third-party applications to view it. Sometimes, the data has to be transferred by sending hard disks by mail, due to the large amount of space they require.

With the release of WebGL, 3D content distribution over web browsers has become increasingly popular. It has evolved into a standard that is natively supported by all major browsers, on desktop and even mobile devices. WebGL now allows developers, artists, companies, researchers and others to share their content with a wide audience, without the need to install additional software. Many services, like Sketchfab [61], have emerged that allow users to upload, share and view content without any knowledge about the underlying WebGL mechanics.

In most cases, the content is relatively small. Small, in this context, means that the full data set fits into memory, can be downloaded in a reasonable amount of time and can be rendered in real time. Some types of content, however, exceed these requirements. The data may not fit into memory, or downloading the full data would take minutes or even hours.

The goal of this thesis is to develop a viewer that is capable of streaming and rendering point cloud data sets with billions of points, without the need to transfer the whole data set first or to install a third-party viewer.

1.2 Problem Definition

3D scanning technologies such as laser scanners or photogrammetry produce enormous amounts of data, often exceeding hundreds of millions or billions of points. Due to the nature of point data, a high number of points is required to accurately represent even simple models. A flat wall, for example, can be represented by a single quad and a texture, but thousands or millions of coloured points may be required to reach the same amount of detail.

While many use cases convert dense point cloud models to a more compact textured triangle mesh, this is not always desirable or possible. The generation of a low-resolution triangle mesh comes along with a loss of information, may not always be possible due to a low scan density for complex objects or surfaces, and it can be very time consuming

and costly. Apart from that, it is often necessary to be able to verify the result from the conversion of point clouds to three-dimensional triangle meshes, two-dimensional maps or any other end product.

One of the major challenges of point cloud data is the processing and rendering of data sets that do not fit into memory. These kinds of data sets require the use of out-of-core algorithms. Out-of-core algorithms load and process only small chunks of data at a time. Once a chunk has been processed, or it is no longer needed, it is removed from memory to make place for the next chunk.

As an example for the dimension of point cloud data, the United States Geological Survey (USGS) is currently undertaking a nation-wide scan of the whole United States. Assuming an Elevation Quality Level of 2, with an effective point density of 3 points per square meter, about 27 trillion points are expected [63][62]. This translates to roughly 540 terabyte in uncompressed storage.

Apart from processing and rendering huge data sets, making them readily available is another big challenge. Surveyers may want to share data with their customers or advertise their past projects; archaeologists, artists or scientists may want to share their data sets with the public to get them interested in their work; and others may want to be able to quickly analyze a point cloud without copying or downloading huge amounts of data first. The easier it is to access, the greater the audience. Few users would upgrade their hardware just to take a look a data set for fun. More users will be interested if all that is required is a semi-large download and a third party viewer. One of the aims of this thesis is to maximize the audience by making the process as easy as visiting a web page.

In order to be able to view the complete dataset with all its details in real time in a web browser, this master thesis is based on the modifiable nested octree (MNO) structure introduced by Scheiblauer [54]. This structure makes it possible to cull points outside the view frustum, and to render distant regions at a lower level of detail.

1.3 Contributions

The main contribution of this thesis is the adoption of a proven point cloud rendering method to a less flexible and resource limited, but widely available runtime environment: a standard web browser.

This adoption includes modifications to account for network transfer rates that are two orders of magnitudes lower than loading data from disk, and rendering modes that display data in useful ways or higher quality, with a lower impact on performance.

Contributions that improve behaviour from slow connections are a new file format, the progressive loading and rendering of two-dimensional height profiles, and point-wise adaptive point sizes, which adjusts the size of each point to the level of detail as additional points are streamed in over time.

Contributions that improve rendering results are the Poisson-disk subsampling method, which produces more evenly spaced subsets, an interpolation method that reduces occlusions from overlapping points, and adaptive blend-depths for high-quality splatting.

The following list provides a short description of individual contributions:
Contributions that improve behaviour from slow connections:

1. A new file format with a partitioning of the hierarchy that allows Potree to load large point cloud hierarchies on demand. Multi-resolution hierarchies of large data sets may consist of millions of nodes. Hierarchies of that size may be hundreds of megabytes in size, which significantly increases initial load times if they are stored in a single file.
2. A fast and progressive height-profile query method. Even profiles that contain millions or billions of points are quickly displayed, since only the most important points are loaded and rendered.
3. An adaptive point-size mode that adjusts point sizes to the level of detail. This adaption allows to hide density differences between different levels of detail. As a side effect, this method can make a point cloud appear like a closed surface with little overdraw at any but the closest zoom levels. The difference to the point-size heuristic of Scanopy is that the adaptive point size mode of Potree adjusts the size point-wise to the level of detail, whereas the point-size heuristic of Scanopy adjusts the size node-wise, based on an estimation of the density of the points in a node.

Contributions that improve rendering results:

1. A fast Poisson-disk subsampling method that creates more natural-looking subsets than gridded approaches. Poisson-disk samples are uniform point sets with a minimum distance between each point.
2. An interpolation splat mode which produces nearest-neighbor-like renderings that resemble Voronoi diagrams. This method is implemented as a single-pass shader that modifies fragment depths in order to render points as paraboloids instead of screen-aligned squares. It provides a trade-off between the performance of square or circle splats and the quality of high-quality splatting methods.
3. Adaptive blend-depths for high-quality splatting (Gaussian splats). The blend-depth specifies the range within which points are blended together, in order to create a smoother and anti-aliased result. A constant value is not suitable for arbitrary zoom-levels in a multi-resolution rendering system, due to the increased distance between points at lower levels of detail. Adaptive blend-depths adjust the blend-depth to the world-space size of a point, instead. If combined with adaptive point-sizes, it sets the blend-depth equal to the spacing between points.

And more generally, this thesis contributes a state-of-the-art WebGL point cloud rendering system with high performance, high-quality rendering modes and a variety of useful tools. Additionally, various state-of-the-art methods were implemented, such as Eye-Dome Lighting, which computes illumination without the need for normals, fast point picking on the GPU, clip boxes and annotations.

1.4 Structure of the Work

Chapter 2 gives an overview of related work, including other viewers for large point clouds, and work related to high-quality rendering of point clouds. A description of the adopted modifiable nested octree model, the build-up of this structure and how it is traversed during rendering is given in Chapter 3. Chapter 4 describes how a point cloud is rendered. This includes the calculation of colors and point sizes, as well as high-quality rendering modes and the Eye-Dome Lighting shader, which computes illumination and outlines without the need for normals. Chapter 5 covers implementation details of Javascript and WebGL, and describes some of the functionality in Potree. Performance evaluations, applications of Potree by third parties, and some of our own results are shown in Chapter 6. A conclusion and a non-exhaustive list of future tasks is given in Chapter 7.

Related Work

Levoy and Whitted [29] were the first to suggest points as a rendering primitive. Sufficiently dense sets of points are able to represent continuous three-dimensional surfaces, and their simplicity makes points a viable alternative to more complex models.

In this chapter, we will cover previous and ongoing research and projects that are related to the rendering of a large amount of points, and methods to improve the quality of point-based rendering.

2.1 Rendering Massive Point Clouds

Handling massive amounts of points, which generally do not fit into memory, require out-of-core algorithms that stream in, process and render only a small subset of the whole data. Most methods employ variations of hierarchical space-partitioning structures, also referred to as multiresolution structures, such as kd-trees, octrees, or quadtrees, and populate all nodes with data that represents the original model at different resolutions. Some of these methods redistribute the original point data within the hierarchical structure, while others store the original data only in the leaf nodes and downsampled averages in inner nodes.

The QSplat rendering system by Rusinkiewicz and Levoy [52] was the first multiresolution system that was capable of rendering hundreds of millions of points. The QSplat method creates a bounding-sphere hierarchy out of an input point cloud or mesh that can be traversed to create a progressively more refined rendering of the model. Each leaf node represents a single point sample, whereas inner nodes represent a bounding sphere that encompasses their respective subtrees. Rendering is done by traversing the hierarchy until a leaf is encountered, or until a desired level of detail is reached, i.e., when the screen-projected bounding sphere is sufficiently small. A point splat that represents the current bounding sphere or vertex of a node is drawn whenever traversal in a subtree is suspended. QSplats uses a binary tree that is always split along its longest axis.

Gobbetti and Marton [20] developed a GPU-friendly multiresolution structure, Layered Point Clouds (LPC), that significantly reduces the cost of traversal on the CPU-side and exploits the proficiency of GPUs at rendering thousands of geometric primitives in parallel. The LPC structure stores subsamples of the point cloud in each node of the hierarchy. The cost of traversal is reduced as it is only necessary to traverse down to one of the subsamples that cover larger areas, not to individual points that cover small areas. The subsamples are static, sent to the GPU once, and subsequently rendered by the GPU, which is particularly good at rendering thousands of polygons or points in parallel. LPC uses a binary tree that is always split along its longest axis.

Wimmer and Scheiblauer [75] introduced nested octrees, a structure that is similar to LPC in that it stores subsamples of the original data in its nodes, but it accepts arbitrary point data sets as input, whereas LPC assumes input to be uniformly sampled. The subsamples in each node are built with the help of an inscribed octree. These inscribed octrees are called inner octrees, and the nodes they are inscribed in are part of the outer octree, hence the name nested octrees. The outer octree is used to determine visibility and the inner octrees are used to create the subsample.

Further research led to structures that are not only suitable for the rendering but also the modification of massive point clouds. Wand et al. [70] proposed an octree structure that distributes the original data sets in leaf nodes and simplified multiresolution representations in inner nodes. The simplified multiresolution representation may consist of a selection of representative points in the subtrees or averages of points in subtrees. In the former case, duplicates are created and in the latter case, new points are being created. Newly inserted points always travel down to a leaf node and update the multiresolution representation along the way. Similarly, when a point is deleted, it is removed from the corresponding leaf, and the multiresolution representation, up to the root, is updated. Wand et al. also describe the case of a newly added point that is outside the bounding box of the root. In this case, a new node with double the size will be created and the current root will become a child of this new node. This process is repeated in the direction of the point, until the octree encompasses the new point.

Another structure for the editing of point clouds is the modifiable nested octree (MNO) by Scheiblauer and Wimmer [56]. In addition to an octree structure that is suitable for insertion and deletion operations, they also introduced a selection octree that allows users to select points with a volumetric brush. The MNO is based on the nested octrees, but in order to improve the performance of insertion and deletion operations, the inner octree has been replaced with a grid. In an MNO, the point samples are created by accepting points that fall into a cell of a node's inscribed grid. The selection octree was introduced so that users can select points with a volumetric brush, without disregarding points in higher detail nodes that were not visible during the selection. A basic volume brush tool would only be able to select points that are in core during the selection. The selection octree, however, remembers which parts of the volume were part of the selection and will make sure that the respective points are marked as selected if they are loaded later on.

2.2 Web-Based Massive Point Cloud and Voxel Rendering

This section describes web-based point cloud renderers that also aim to render large data sets.

Entwine, [26] **Greyhound** and **Plasio** [44] are part of a point cloud rendering stack by Howard Butler, Connor Manning and Uday Verma. Entwine is the indexing library which takes care of building an optimized data structure for efficient streaming of point clouds. Subsamples in a node are obtained by keeping the point that is closest to the center of a cell in the inscribed grid. Greyhound is the HTTP server that streams the indexed data to a client upon request. Plasio is a WebGL-based point cloud renderer that can render point clouds in las or laz format or stream a point cloud from a greyhound server. They also created a Javascript port of laszip in order to be able to decompress laz files directly inside the browser.

PointCloudViz server and the corresponding web client [45] are a commercial service by Mirage Technologies [35] and a complement to their free desktop LIDAR viewer. Their system uses a multiresolution structure for efficient streaming and rendering. Notable features include oriented splats, lighting, different materials such as RGB, intensity and classification and the modification of the color gradients for elevation and intensity through sliders.

ShareLIDAR [59] is a multi-resolution point cloud renderer with hosting service. Notable features include illumination through normals, an orthographic top view, a section (height-profile) tool and the adjustment of point sizes to reduce holes. A downside is that it loads data in smaller tiles that do not cover the whole data set. This leads to large empty space while the user waits for the data to stream in.

udWeb Demo [66] is a web browser based demonstration of Euclidean Unlimited Detail / Geoverse technology. Elements are rendered as blocks instead of points, which leads to the assumption that Euclidean uses voxel-based rendering methods rather than point clouds. This allows to close holes in the data set without causing overlaps, both of which are common in point cloud renderings. Blocks are initially very large, giving a coarse representation of the data, and shrink in size as new data is streamed in.

2.3 Desktop-Based Massive Point Cloud And Voxel Rendering

Scanopy [54] was a project by the computer graphics department at TU Wien and one of the first point cloud renderers that was able to render and also edit point clouds with billions of points. Potree originated as a web-based version of the structures and algorithms that were developed with Scanopy.

Arena4D by Veesus [67] consists of a desktop renderer and a point server that allows to stream point clouds. It is able to render massive point cloud data sets and includes various tools for selection and editing of point clouds.

Geoverse [19] is a point cloud viewer by Euclidean, which is known for the marketing of its Unlimited Detail technology, and supports rendering of data sets in the terabyte range. Although it is advertised as a point cloud renderer, the nature of its renderings may indicate that it is actually a voxel renderer.

PointCloudViz is a free point cloud viewer by Mirage Technologies [35]. It includes features such as the generation of digital elevation models (DEM), overlay of georeferenced images and measurement.

Voxel Quest [68] was a project by Gavan Woolery that was able to create and render huge procedural voxel environments. The author experimented with different rendering techniques during its lifetime, including the raycasting on signed distance fields and rasterizing voxels as point primitives, effectively treating a voxel data set as a point cloud.

2.4 High-Quality Point-Based Rendering

Unlike polygon meshes, point clouds do not contain connectivity information and represent a set of points on a surface rather than a closed surface. Points are usually rendered as rectangles, circles or single pixels on the screen. These primitives are fast to render on GPUs, but the results suffer from problems such as occlusions between overlapping points and aliasing artifacts. High-quality rendering methods provide means of improving the quality of point cloud renderings without the need to convert them to meshes.

Surfels, short for surface elements, were introduced by Herman [24] and later suggested as a rendering primitive by Pfister et al. [41]. In the context of point-based rendering, a Surfel is an oriented disk or ellipse in a three-dimensional space. The oriented ellipses of a Surfel help to avoid holes between adjacent samples.

Surface splatting by Zwicker et al. [76] describes a high-quality, filtered method in which the pixels of a projected point are weighted by a Gaussian filter function. Pixels close to the center of a point are weighted higher than pixels further away. The weighted contributions of all points are summed up and then normalized by the sum of all weights. To ensure that only points that belong to the same surface contribute to the result, the method compares depth values and only accepts new contributions if they are within a certain threshold. This method produces high-quality, anti-aliased results, but it was realized in a software renderer at this time.

After GPUs evolved, Botsch et al. [7] introduced *High-Quality Surface Splatting on Today's GPUs*, which is an approximation of the original surface splatting that finally made it possible to exploit the GPU in order to produce high-quality, anti-aliased, renderings of point clouds. Their method consists of a depth pass, an attribute pass and a normalization pass. The depth pass ensures that only points of a visible surface, i.e., points closest to the viewer plus an additional offset, contribute to the result. The attribute pass uses additive blending to accumulate weighted contributions of each point, and the normalization pass normalizes the weighted contributions by the sum of weights.

Surface splatting was developed with oriented splats in mind, but since most point cloud data sets do not contain the necessary normals or radii, Scheiblaue [55] suggested

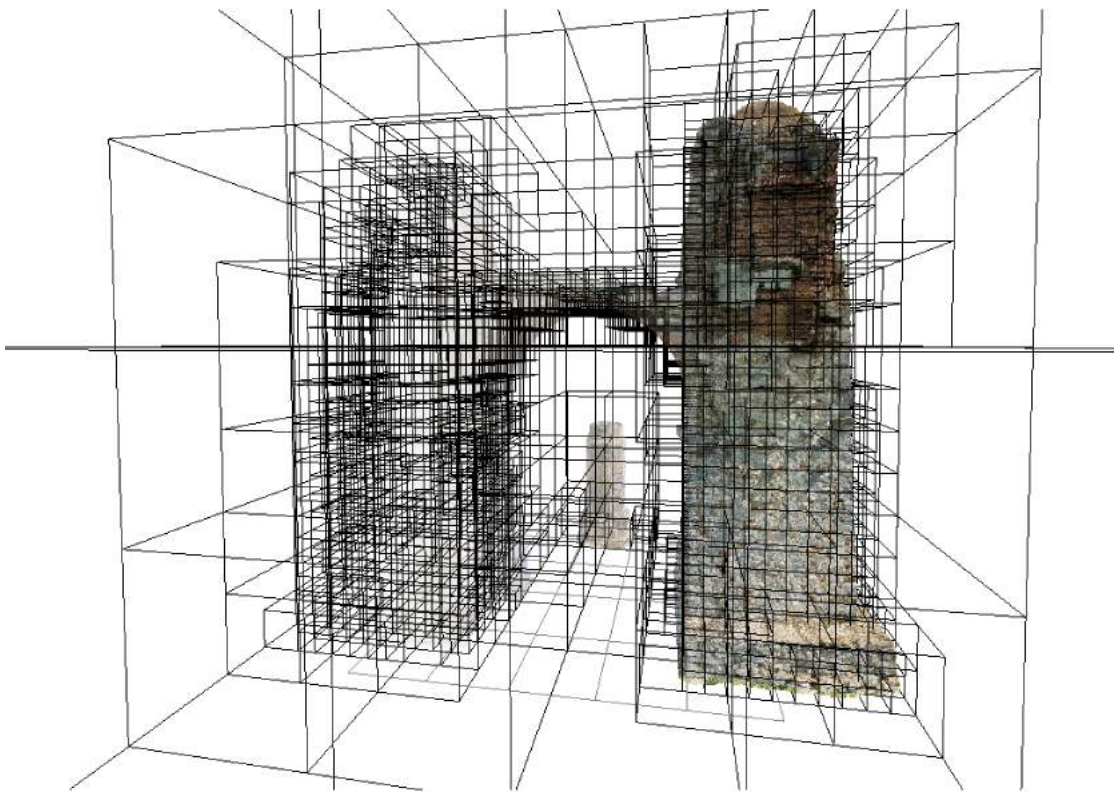
to use screen-aligned circles instead, which is also what we are doing in Potree.

Preiner et al. [47] later developed *Auto Splats*, a method to calculate missing normals and radii on the fly during rendering in screen-space, in order to allow for higher-quality surface splatting than what is possible with screen-aligned circles. Calculating normals on the fly also made it possible to apply illumination models such as Phong.

Eye-Dome-Lighting, a technique that creates illuminated point cloud renderings without requiring normals, was described by Boucheny [8]. This technique is similar to an edge-detection filter in that it samples the neighborhood around a point and creates a response based on the differences in depth. Small differences in depth correlate with surfaces that face the user, while larger differences in depth correlate with surfaces that point increasingly further away. This characteristic is used to compute a shaded surface without relying on normals.

CHAPTER 3

Data Structure



3.1 Overview

Point clouds are often too large to fit into memory as a whole and therefore have to be processed using out-of-core algorithms. One possible out-of-core option is to split the data into multiple tiles and process one or a few tiles at a time. This approach works well for processing, but for visualizations it is often desired to display the whole data set and not only a few tiles at a time.

Storing various levels of detail of the original model in a hierarchical space-partitioning data structure allows a point cloud renderer to quickly load and display the relevant parts of a point cloud. Regions that are close to the camera are rendered at a higher level of detail than distant regions, and regions that are outside of the view frustum are discarded entirely. Variations of octrees and kd-trees are two popular space-partitioning structures for the rendering of large point clouds. Some variations store subsamples of the original point cloud. Others store the original data in leaf nodes and downsampled averages or the bounding volume of a subtree in inner nodes. We have chosen a structure that subsamples the original point cloud, because it does not create new points that require additional disk space, and because it allows users to do point picking and measurements on original, unaltered, data at any zoom level, without the need to wait until a leaf node is loaded. An overview of structures for the rendering and modification of large point clouds is given in Chapter 2.

Potree’s structure is based on a slightly adapted modifiable nested octree (MNO) structure, which was introduced, and built into the Scanopy point cloud renderer, by Scheiblaue [54]. Figure 3.1 shows a spherical point cloud that has been partitioned into a MNO.

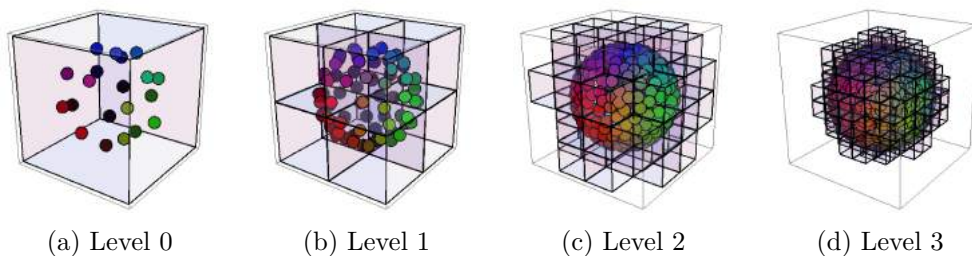


Figure 3.1: Low-level nodes (left) contain sparse models over large regions. Each level exponentially increases the number of points and details.

3.2 Modifiable Nested Octree

We briefly describe the modifiable nested octree structure, on which Potree’s octree structure is based.

The modifiable nested octree structure stores subsamples of the original point cloud in each node. Low-level nodes contain sparse subsamples over large volumes. With each level, the size of a node shrinks while the point density increases. Each point of the

original data set is assigned to exactly one octree node. This means that no new points or duplicates are created, and that combining all points in all nodes returns the original data set.

The original MNO structure obtains its subsamples through an inscribed three-dimensional grid with 128^3 cells. Initially, points are added to the root node and a point will occupy the first cell it falls into. If a point falls into an already occupied cell, and the total number of points in the node is below a threshold, then the point will be assigned to this node anyway, but stored inside a padding array instead of the grid. The padding array holds further points in the same node, additionally to the grid, in order to avoid the need to immediately create a new child node to store these points. New child nodes are created as soon as enough points that fall into a potential new child node have accumulated. New child nodes are therefore immediately populated with a minimum amount of points.

This subsampling approach leads to varying point densities at different octree levels, and it also avoids mostly empty nodes because new nodes are only created after a minimum amount of points have been amassed for a new child. It does, however, not guarantee a certain minimum distance between points. Adjacent grid cells may contain points that are arbitrarily close to each other.

The complete hierarchy of the MNO is stored in a single file. The nodes are stored in one file for each node. Node files are named after the identifier of the corresponding node, which is made up of the indices from the root to the node in question. The root itself does not have an index and so the character *r* is used, instead. The identifier *r042*, for example, stands for the node *root.children[0].children[4].children[2]*.

3.3 Potree’s Octree Structure

Potree uses a variation of the MNO structure with a different subsampling method and a partition of the hierarchy into smaller, quickly streamable, chunks. In order to avoid confusion with the original modifiable nested octree structure, and because Potree does not offer functionality to modify the point cloud, we will refer to it as Potree’s Octree Structure or simply as octree in this thesis.

The resolution of a node is defined by the spacing property, which specifies the minimum distance between points. The spacing is initially computed for the root node, based on the size of the bounding box, and then halved at each level. For example, a spacing of 1 meter may be used for a data set with an extent of 200 meters in each direction. The root node will then contain a low-resolution version of the original data, in which each point is at least 1 meter apart from the next one. The root node’s children will have a spacing of 0.5 meters, which effectively doubles the resolution.

Different values for the spacing affect the number of points in a node, the number of nodes that are required to store all points, and the depth of the tree. A lower spacing leads to a higher amount of points in each node, a lower number of nodes overall and a shallower tree depth. The optimal value of the spacing is difficult to define and depends on various factors, such as CPU and GPU processing power and connection

Low Spacing	Large Spacing
<ul style="list-style-type: none"> • More points in each node. • Fewer nodes overall. 	<ul style="list-style-type: none"> • Fewer points in each node. • More nodes overall.
+ Faster octree traversal.	+ Finer spatial partitioning allows to cull away more points.
+ Fewer draw calls and GPU state changes.	+ Individual nodes are quickly downloaded.
+ Reduces overhead that is associated with each file download.	– Slower octree traversal.
– Inefficient culling due to coarser spatial partitioning.	– More draw calls and GPU state changes.
– Each node takes longer to download	– Many small files must be loaded.

Table 3.1: Advantages and disadvantages of small or large values for the spacing.

speed. Table 3.1 lists some advantages and disadvantages between low and high values for the spacing.

We have chosen a value of $\text{boundingCubeWidth} / 128$ for the spacing. It is synonymous to the partition of a node into a grid with 128^3 cells, which is used for the MNO structure of Scanopy, and it provides a reasonable trade-off between the advantages and disadvantages of low and high values for spacing.

As with MNOs, a node is only split if a certain amount of points are added to it during build-up. Any leaf node in the multiresolution octree is a node that has not been split further, due to the small number of points that have been added to this node. They serve as buckets for remaining points and, as a consequence, do not enforce a spacing between points.

Apart from the spacing, the number of input points is another factor that affects the depth of the octree and the number of nodes in the output. The conversion of the AHN2 point cloud, consisting of 640 billion points, resulted in an octree with 13 levels and 38 million files [33]. The octree hierarchy alone requires 190MB disk space, assuming the hierarchy is encoded in 5 bytes for each node. The client needs the hierarchy to find out which nodes have to be loaded and which nodes are visible, but sending hundreds of megabytes would result in long initial load times. In order to reduce initial load times, the hierarchy is split into multiple smaller chunks that contain *hierarchyStepSize* levels. Figure 3.2 shows an example of a multiresolution octree hierarchy that has been split into chunks with 2 levels each. Essentially, this produces another, shallow, octree that stores the deep hierarchy of the multiresolution octree. This hierarchy-octree allows

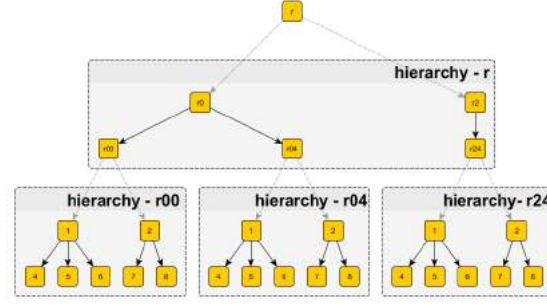


Figure 3.2: The octree hierarchy is partitioned into batches with *hierarchyStepSize*, in this example 2, levels. The hierarchy is then loaded on demand.

quick loading of the actually needed hierarchy on demand.

3.3.1 Poisson-Disk Subsampling

This section will go into the details of the Poisson-disk subsampling in Potree, which is used to generate uniformly spaced subsamples with a minimum distance between points.

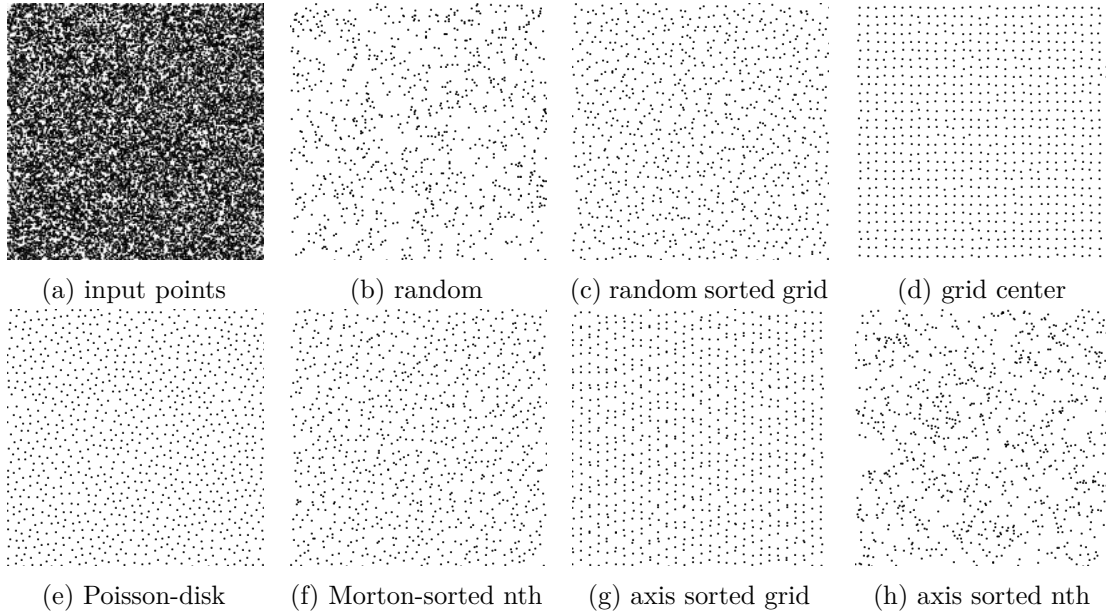


Figure 3.3: Approximately the same number of points selected from an input set (a) through different sampling strategies. (b) random subset. (c) Points in random order. First to fall into a cell is selected. (d) Point closest to the center of a grid cell is selected. (e) Points with a minimum distance to each other are selected. (f) Points sorted in Morton-order. Every *nth* point is selected. (g) Points sorted along axes. First to fall into a cell is selected. (h) Points sorted along axes. Every *nth* point is selected.

Each node stores a subset of points with a certain resolution. A variety of sampling methods, some of which were made up and tested in the process of finding a suitable one, are shown in Figure 3.3.

The modifiable nested octree structure uses a grid-based method where a three-dimensional grid is inscribed into the volume, and the first point that falls into a cell will be accepted. If a point falls inside a cell that is already occupied, it is discarded. This method is simple and fast, but it has the disadvantage that it does not enforce a minimum spacing between points. Points inside adjacent cells may be selected even if they are close to each other. Higher-quality sampling methods ensure that there is a certain distance between points. For example, an improved grid-based method that is used in Entwine [26] favours points at the center of a cell and it may swap an already accepted point with a new one, if the new one is closer to the center.

For this thesis, we opted to use a Poisson-disk sampled approach. In a Poisson-disk sample, each point has a minimum distance to all other points. The resulting data shows more visually pleasing patterns than grid-sampled sets, and it also provides excellent coverage with a small number of points. Enforcing a strict minimum distance between points is a computationally expensive task, however, and requires more complex data structures.

Poisson-disk samples can be created by a class of algorithms that are referred to as dart throwing. They create Poisson-disk samples by creating points and then checking that the distance to previously created and accepted points is sufficiently large. If the distance is too small, the point will be discarded. Cook [11] was the first to propose Poisson-disk samples for computer graphics and also gave an example for a naive dart throwing algorithm that should be avoided due to its cost. Naive dart throwing checks the distances from each new point to all previously accepted points, which is not feasible when processing millions of points. Improved dart-throwing algorithms create samples in empty regions and check against a local neighborhood [9]. Many dart-throwing algorithms are concerned with how and where point samples are produced, in order to improve performance. In subsampling a point cloud, we already have a fixed set of samples and are only concerned about discarding points that are too close to each other.

In order to reduce the amount of distance checks between points, we divide each node into a grid and only compute distances to points inside the same and adjacent cells.

The sample-grids are realized as sparse arrays, since point clouds usually depict surfaces rather than volumes. As such, only a small fraction of cells are actually populated, making sparse arrays a significantly more memory-friendly choice. The C++ standard library offers an implementation of hash maps, called `std::unordered_map`, that is used as a three-dimensional sparse array. The `unordered_map` is a 1-dimensional collection by design, but it can be used as a three-dimensional sparse array by encoding 3 integer components into a single 64bit integer key.

Each cell stores the accepted points as well as references to adjacent cells in order to quickly iterate through points in neighbors, without the need for relatively costly hash-map accesses. New points are accepted by a cell if the smallest distance to all points inside that cell, and to all points in adjacent cells, is larger than the spacing. Cell

instances are created when a newly added point falls into it for the first time. During cell creation, adjacent spaces are checked for already existing neighbors. If a neighbor exists, it is added to the new cells list of neighbors and the newly created cell will also be added to the neighbors list.

The size of a cell can be any value between the spacing and the size of the node itself. Only cell sizes equal or larger than the spacing ensure that points that are relevant for the distance check are stored in the same or adjacent cells. Initially, the spacing itself was chosen for the cell size to keep the amount of distance checks as low as possible. Yin Fei [40] discovered that this leads to unnecessarily high memory usage and low performance, and suggested to use larger cell sizes instead. Increasing the size of a grid cell reduces the number of cells, and with it the memory usage as well as the processing overhead of managing numerous cell instances.

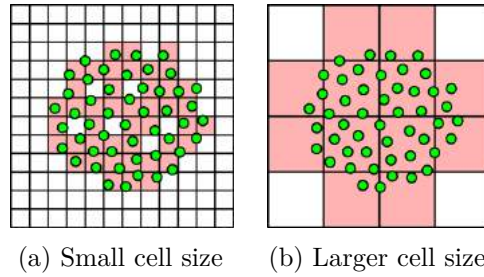


Figure 3.4: Sparse distance-check grid. Distance checks have to be done for points inside the same and to points inside neighboring cells. (a) Fewer distance checks but larger memory footprint and processing overhead. (b) Higher number of distance checks but memory-friendly. Spacing and cell sizes in the actual implementation are lower than depicted.

Figure 3.4 shows the effect of different grid sizes. Whenever a point is added, distances to points inside the same cell and its neighbors are calculated. Points that pass the minimum distance check are added to the grid. Subsequent points are then distance-checked against points that already passed the test. Only occupied cells (red) are initialized and stored in the sparse structure. Each cell keeps a list of accepted points and a list of occupied neighbors.

3.3.2 Build-Up

The build-up process takes one or more point clouds as input, partitions the points into an octree, and stores the result on the filesystem with one file for each node.

The conversion process follows these rules:

1. Initially, the octree consists of a single root node, which also serves as a leaf at this point.
2. Points are added, one-by-one, to the root node.

3. Internal nodes keep a point if no other point is within minimum distance (spacing) and pass it down to its children otherwise.
4. The spacing is halved at each level.
5. Leaf nodes keep all points at first.
6. If a certain threshold of points is reached, a leaf node is expanded. It becomes an internal node and adds all the stored points to itself, but this time following the internal node rules. Points with a certain minimum distance remain in the former leaf node and all the other points are passed down to its newly created child nodes.
7. The data is regularly flushed to the disk, for example each time 10 million points have been processed.
8. If a node has not been touched since the previous flush, its data will be removed from memory during the next flush.
9. If a point is about to be added to a node that has been removed from memory, the data will be read back from disk to memory first.

Rules 3 and 4 lead to subsamples with a low resolution in lower levels and a gradually higher resolution in higher levels. Low-level nodes keep points with a large distance from each other. This distance is halved with each level, thus increasing the resolution.

Rules 5 and 6 allow the octree hierarchy to be expanded on demand. The hierarchy is shallow at first and increases in depth as new points are added. Points inside internal nodes have a certain spacing. Leaf nodes, on the other hand, are buckets where all remaining points are stored.

Rule 7 allows users to view the current state of the conversion in Potree after each flush, without the need to wait until the conversion has been finished. Since browsers tend to cache files, it may be necessary to disable caching or to refresh the page instead of simply visiting an URL. Otherwise, old data from a previous flush may be shown.

Rules 8 and 9 ensure that memory usage is kept low by removing data that has not been used in a while. This works best if subsequent points possess a certain locality, i.e. they are relatively close to each other and therefore fall into the same octree nodes. It increases the chance that a high-level node is fully processed in only one or a few flush cycles and not required to be in memory most of the time. Storing the whole data set in multiple tiles is an effective way to ensure a certain amount of locality and therefore reduce memory usage and to increase performance by reducing the amount of times a node has to be read back to memory. Scheiblauer [54] and Leimer [28] have shown that build-up times can be reduced if the point cloud is sorted in advance. The best case is a Morton-ordered data set, in which all points that fall into the same node are stored next to each other. Nodes are therefore fully processed at once and once unloaded, will not have to be touched again. If subsequent points are relatively uniformly distributed over the whole volume, on the other hand, the converter has to keep all or most of the data in memory or read it back more often.

Issues

While this implementation creates Poisson-disk sampled subsets within each node, it does not do so for the combination of nodes. During rendering, multiple nodes are combined, but the Poisson-disk property has not been enforced for adjacent or intersecting nodes. For adjacent nodes, this can lead to a noticeably higher density of points near their borders. The adjacent nodes issue may be resolved by doing distance checks to points in these nodes as well. This is very costly, however. A cheap heuristic to avoid increased point densities at borders would be to discard all points within a certain distance to the border and immediately pass them to the next level, instead.

Although the current sampling algorithm produces Poisson-disk sets, the coverage may not be optimal. The order in which points are processed influences the results, and unfavourable cases can lead to noticeable stripes and holes. Figure 3.5 shows an optimally and a badly sampled point set. Initial tests have shown that randomizing the input order of points helps to resolve this issue.

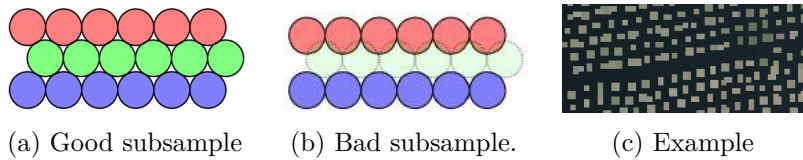


Figure 3.5: (a) Input points were processed in optimal order. The result has very good coverage. (b) Input was processed in an unfavourable order. The gap between sampled points leaves no place for any more points without violating the minimum distance constraint. (c) A real-world example of a point cloud that has been processed in an unfavourable point order.

3.4 Octree Traversal and Visible Node Determination

The octree structure allows for efficient rendering of large data sets through view-frustum culling and rendering at a higher level of detail near the camera, as shown in Figure 3.6. View-frustum culling skips nodes that are outside the visible region. Octree nodes with different levels overlap and are rendered jointly to increase the level of detail. The level of detail in a region is equal to the level of the highest-level node therein. Level of detail constraints ensure that nodes closer to the camera are favoured over nodes that are far away. A point budget limits the number of points loaded and rendered at any given time, which helps to adapt performance requirements to the capabilities of different hardware.

The nodes that should be rendered are determined in an octree traversal step. Traversal is done in a screen-projected-size order. The largest node on screen is visited first, then the second largest, and so on. The projected size is obtained as a function of the field of view, the distance to the center of the node, the node's bounding sphere radius, and the height of the screen. Figure 3.7 and Equation 3.1 show how the field of view relates to the slope of the view frustum. Equation 3.2 gives the projected size of the

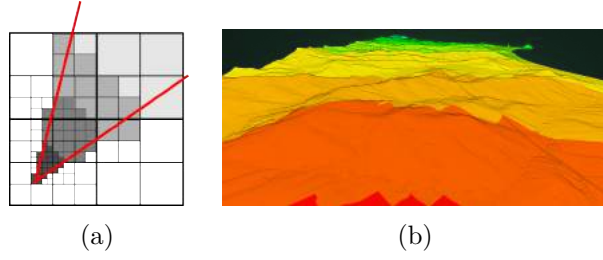


Figure 3.6: (a) View frustum culling discarding nodes outside the frustum and LOD favouring high-resolution nodes close to the viewer. (b) Color-coded LOD.

node, which is inversely proportional to the slope and the distance. The radius of the node and the height of the screen are used to scale the result to a pixel size of the node. At this time, Potree only considers the screen-projected-size in the traversal order. The Scanopy renderer also accounts for the distance of a node to the center of the screen, which leads to a higher amount of detail at the center where it matters the most.

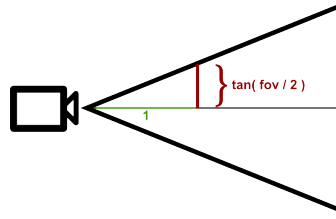


Figure 3.7: The slope of the view frustum.

$$slope = \tan\left(\frac{fov}{2}\right) \quad (3.1)$$

$$projectedSize = \frac{screenHeight}{2} * \frac{radius}{slope * distance} \quad (3.2)$$

During the traversal, Potree keeps a list of visible nodes, a list of nodes that are visible but have not been loaded, and the sum of points in all visible nodes. A node is considered visible if its bounding box intersects with the view frustum. If this is not the case, the node will be discarded and its children will not be traversed further. Traversal continues until there are no more nodes to visit, or until one of two conditions is met. The most important condition, and usually the limiting factor, is that the sum of visible points remains below a threshold, the point budget. The point budget allows users to reduce the number of rendered points for a better performance, or increase it for a better quality. The second condition is that a visited node must have a minimum screen projected size. This condition reduces unnecessary work when a point cloud is far away, in which case rendering a number of points well below the point budget will be sufficient.

After the traversal, nodes in the list of visible but unloaded nodes are scheduled to be loaded. To avoid loading too many nodes at once, and also to avoid loading nodes that may not be visible anymore in a few frames time, only the first X nodes in the list are scheduled to be loaded. Due to the traversal order, the first X nodes in the list are the nodes with the largest screen-projected-size and therefore the most important nodes. A value of $X = 5$ has proven to work well in practice.

An important consideration for the traversal is that the octree hierarchy is loaded on demand. It is therefore possible that the traversal considers a node visible at first but invisible at a later time. This will happen if nodes of the unloaded hierarchy have a higher importance than already available nodes. Once this part of the hierarchy has been loaded it will take up a share of the point budget that will no longer be available for the previously visible node.

The rendering of the visible nodes is handled by three.js. Visible nodes are marked as such with a flag, and the three.js renderer will then do its own traversal to invoke draw calls for all nodes that were marked as visible.

CHAPTER 4

Point Cloud Rendering



This chapter covers various rendering techniques for point clouds that were implemented in Potree. It describes how data stored as point clouds can be visualized and different methods to illuminate and draw points.

4.1 Point Attribute Coloring

Point clouds may be colored using point attributes including, but not limited to, RGB. Attributes other than RGB have to be mapped to an RGB color at runtime. Point attributes are stored in the point cloud for each point in addition to the coordinates or, in the case of elevation, as one of the coordinate axes. The level of detail is the only coordinate that is computed at runtime and not stored on disk.

Depending on the scanning devices that were used to capture the data, and the post-processing algorithms used to augment it, different kinds of attributes are stored in a point cloud. Laser scanners usually provide at least intensity, whereas photogrammetry usually provides at least RGB data. Some attributes, such as classification, require post-processing steps after the data has been captured.

The following paragraphs describe attributes that are supported by Potree.

RGB usually describes the observed real-world color of a point. RGB colors are automatically captured by photogrammetry software, as they are part of the input images. They are not an inherent property of laser scans, but additionally captured photos can be used to project colors onto points. However, this attribute may also be used to store any other kind of information that is mapped to RGB. Some software packages, for example, can calculate ambient occlusion and store it in the RGB channels. Height and intensity are also frequently stored in RGB to provide colours for an otherwise uncoloured point cloud.

Intensity indicates the strength of the backscattered signal in a laser scan, or the derived surface reflectance. The intensity is affected by various conditions such as distance between scanner and surface, atmospheric conditions, type of scanner and surface reflectance [69]. Due to this, merging multiple airborne laser scans will often lead to inhomogenous results, with noticeable differences in intensity for the same type of surface.

Elevation, or height, does not require additional space since it is already part of the Euclidean coordinates. The elevation value is mapped to a color with the help of a color gradient. The gradients are defined through an ordered list of elevation values and their respective colors. Values in-between are interpolated and values outside are clamped. Colored gradients allow users to quickly identify highs and lows, such as between mountains and valleys or buildings and terrain.

Classification is derived from other point attributes, or related resources such as georeferenced images. It indicates whether a point is part of the ground, vegetation, buildings or other classes. In a most basic approach, bottom-most points may be classified as ground. In forest regions, anything above the ground may be classified as vegetation. The class of a point is stored as a single number. Each number is then mapped to a color. A look-up table is used to map from a class to a color.

Return number specifies the order in which points were captured from a single beam. Certain materials, like leaves on a tree, reflect or absorb a portion of a laser beam and let another portion pass through. This remaining energy may hit another object and be reflected as well. Some laser scanners are able to capture not only the first but also subsequent hits. Trees are especially likely to generate multiple hits. First hits are generated by the topmost layer of trees and last hits are often generated by the ground, trunk, or any structure below the canopy.

Point Source indicates from which file or source a point originated. In case of airborne laser scans, the point source usually indicates the flight line.

Level of Detail is calculated on the fly during rendering. It is equal to the level of the most detailed visible node in a region. The LOD is an integer value between 0 and the depth of the visible hierarchy, which is then mapped to a color through a gradient look-up.

Figure 4.1 shows examples for attributes available in Potree.

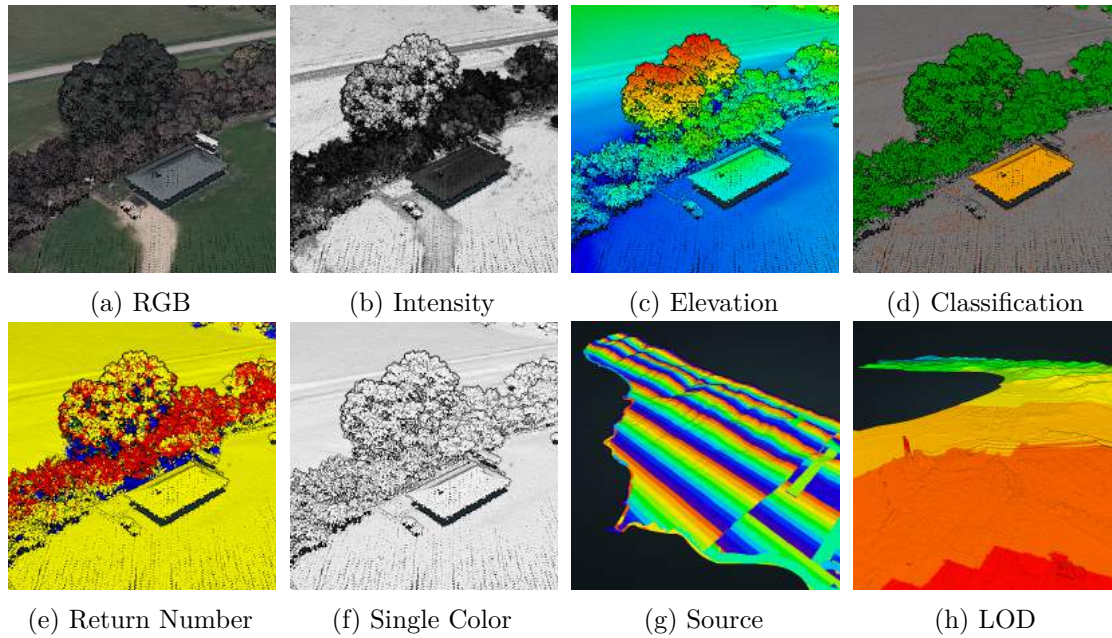


Figure 4.1: Various point attributes. CA13 point cloud courtesy of Open Topography and PG&E [42]

4.2 Point Splatting

Due to the missing connectivity between vertices, points are often rendered as single pixels or screen-aligned squares or circles. These primitives are fast to render and they are natively supported by graphics libraries or trivial to implement. The visual quality is, however, low compared to high-quality splatting and interpolation techniques. The

following sections provide a description of the advantages and disadvantages of different point-splatting techniques.

4.2.1 Squares and Circles

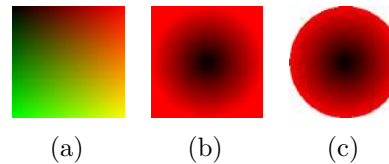


Figure 4.2: Points are rendered as squares in WebGL. Circles are obtained by discarding fragments. (a) *gl_PointCoord* stores the normalized coordinates of a square with the origin at the top left. (b) Coordinates are mapped to the distance from the center of the square. (c) Fragments with a distance larger than 1 are discarded.

Squares are natively supported by WebGL with the *gl.POINTS* primitive. The vertex shader member *gl_PointSize* defines the size of the square in pixels. Circles are not natively supported. Instead, circles are displayed by rendering squares and discarding all fragments that fall outside the circle’s boundary, as shown in Figure 4.2 and Listing 4.1.

Listing 4.1: Rendering circles in WebGL by discarding some fragments of the rendered square.

```
float distanceFromCenter = length(2.0 * gl_PointCoord - 1.0);

if(distanceFromCenter > 1.0) {
    discard;
}
```

A common characteristic of this rendering mode are aliasing artifacts that give point cloud renderings a noisy appearance. This is especially noticeable in point clouds with high-frequency color information and during movement.

4.2.2 High-Quality Splats

Botsch et al. [7] proposed an efficient and GPU-friendly algorithm that increases the quality by blending points together, instead of just taking the closest one. The idea of this method is that all points within a certain range are considered to be part of a surface patch and should therefore contribute to the result. This method reduces occlusions, creates smooth transitions between points and reduces the aliasing artifacts that are common in point cloud renderings.

The method consists of a depth, an attribute and a normalization pass. It was originally developed with oriented splats in mind, but since few point cloud data sets contain the necessary normal information, and even fewer the additionally required ellipsoid

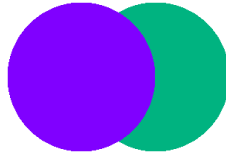


Figure 4.3: One point being occluded by the other. Points with the same or a similar depth should be blended together instead.

radii, our implementation uses screen-aligned circles instead, as proposed by Scheiblauer et al. [55].

The **Depth Pass** writes linear depth values of the nearest fragments into a frame buffer.

The **Attribute Pass** produces a weighted sum of attribute values (RGB, Normals, ...) and a sum of weights. Floating point textures are used due to the high precision requirements of the result, and blending mode is set to additive to obtain the respective sums.

In the first step, the attribute-pass compares the depth of a fragment to the depth inside the depth buffer. All fragments within a certain range pass this test and contribute to the result. Fragments that are farther behind will be discarded. In the next step, a weight is assigned to fragments that passed the depth test. This weight depends on the distance from the fragment to the center of the corresponding point primitive. The closer to the center, the higher the weight. In the final step, the fragment value is multiplied by the weight and stored in the rgb channel. The weight itself is stored in the alpha channel. Due to the additive blending, the result of this pass is a weighted sum of attributes in the color channel and a sum of weights in the alpha channel, as shown in Figure 4.4.

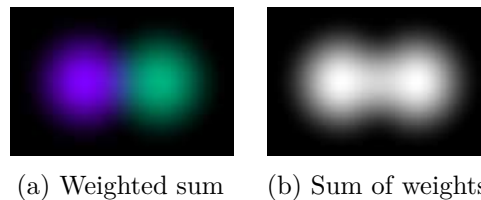


Figure 4.4: The attribute pass generates a weighted sum of attribute values and a sum of weights.

The **Normalization Pass** divides the weighted sum-of-attribute values by the sum of weights. This brings the attribute values from an arbitrary range back to a range of 0 to 1.

The smoothness of the transitions depends on the weight function that is used in the attribute pass. Botsch and Scheiblauer suggested a Gaussian weight function, which has its peak at the center and a smooth falloff as the distance to the center increases. For Potree, we have experimented with other weight functions as well and we have chosen the function shown in Equation 4.1. This weight function behaves similar in that it has

its peak at the center and a smooth falloff, but it approaches zero at a distance of one. This property leads to a smoother transition near the border of the intersection of two points. A Gaussian function, on the other hand, does not approach zero at a distance of one, which leads to a sudden jump in the sum of weights where two points intersect, as shown in Figure 4.5. It is, however, also possible to offset the Gaussian weight such that it approaches zero at a distance of one. In practice, the results of Gaussian weights and the weight function that we use are hardly distinguishable, although there may be signal processing related differences that we did not explore at this time.



Figure 4.5: (a) The Gaussian weight function exhibits a disconnect in the sum of weights (green) where two points meet. (b) The parabolic function starts from zero at the edge, thereby avoiding a disconnect in the sum of weights.

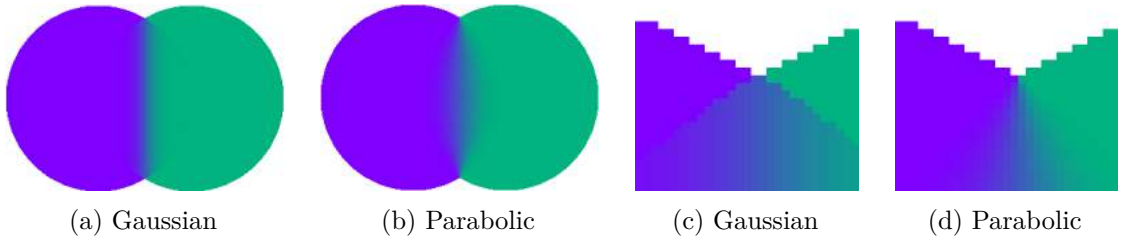


Figure 4.6: Differences in Gaussian and parabolic weights. The jumps in the sum of weights are noticeable only at the border of the intersection.

The weight function in Equation 4.1 is defined for a normalized distance with 0 at the center of the point and 1 at its border. The hardness factor allows adjusting the smoothness of the transition. Figure 4.7 shows how weight function and the final, normalized, result are affected by the hardness.

$$weight = (1 - distance^2)^{hardness}, distance \in [0, 1] \quad (4.1)$$

The range within which points are blended together is specified by the blend depth. Figure 4.9 shows the results for different values. With a value of zero, only the fragment closest to the viewer and fragments with exactly the same depth value will be blended together. When using a value of 10, all fragments that are up to 10 units behind the closest one will be blended as well. Small values are useful to create smoothly blended surfaces, while larger values allow users to see through objects.

Using a constant blend depth is not always useful, especially in a hierarchical rendering system. If the viewer zooms out, detailed nodes are culled away. What remains

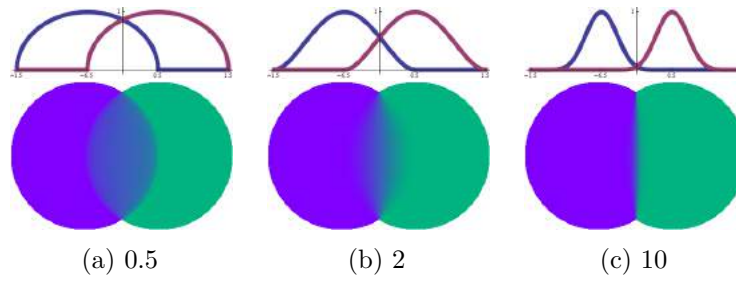


Figure 4.7: Effect of different values for hardness. Top row: Weight function. Bottom row: Image after normalization. The hardness factor affects the smoothness of the transition between points. High values produce results similar to Voronoi diagrams.

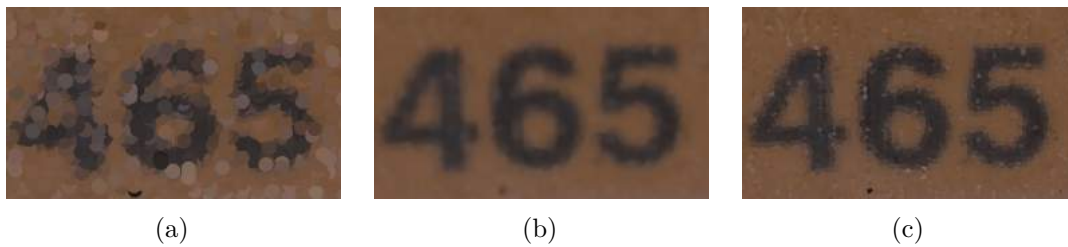


Figure 4.8: Comparison of circles(a) and high-quality splats with hardness 2(b) and 50(c). Subsea Equipment point cloud courtesy of Weiss AG [73]

are points with a large distance from each other, which won't be blended together with a low and fixed blend depth. In our implementation, the world space radius of a point plus an optional offset is therefore used as blend depth. The world space radius is obtained by projecting a point from screen space back to world space. As a result, points are automatically blended together at arbitrary distances and object scales and without seeing through the surface. If see-through is desired, the user can manually define an additional offset.

4.2.3 Interpolation

The interpolation mode was developed for Potree to create high-quality point cloud renderings in a single pass [57]. This is achieved by rendering points as paraboloids rather than flat, screen-aligned rectangles or circles, as shown in Figure 4.10. The result is a nearest-neighbor-like interpolation of points that closely resembles a Voronoi-diagram.

Other shapes like cones and spheres can also be used to improve quality, but paraboloids have a few advantages. Spheres can only be used with circular shapes, while cones and paraboloids work with rectangular shapes, too. The weight function for paraboloids is also the simplest of all three. The most important advantage, however, is that overlapping paraboloids have straight intersections, whereas cones and spheres at different depths will produce curved edges. Straight edges are arguably more pleasant and less irritating to look at in many cases.



Figure 4.9: Large blend depth values allow seeing through surfaces. Overpass point cloud courtesy of Surface and Edge [64]

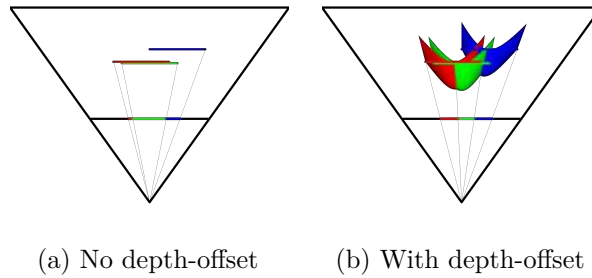


Figure 4.10: Adding a depth-offset to the fragments transforms flat screen-aligned squares into paraboloids. Rendering points as paraboloids reduces undesired occlusions and results in nearest-neighbor-like interpolation. Figure taken from [57].

This method is implemented by modifying the depth values of screen-aligned squares inside the fragment shader. The following steps are required. First, the world-space radius of a point is calculated in the vertex shader by projecting its screen-space pixel size back into the world-space. In the next step, the fragment shader calculates the depth offset of a fragment from the radius and a given weight function. Finally, the modified depth-value is written to the *gl_FragDepth* output variable.

The interpolation mode offers a trade-off between the performance of squares and circles and the quality of high-quality splats. It reduces overlaps and improves the readability of high-frequency details such as text, lines and patterns. It also avoids popping artifacts that appear when an occluding point suddenly becomes an occluded point from one frame to the next. It does not deal with noise and aliasing, however. Both are as prevalent with the nearest-neighbor-like interpolation as with the square and circle rendering modes.

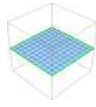



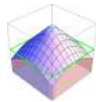



	Weight function	Shape	Result
square	0		
sphere	$\sqrt{1 - (u^2 + v^2)}$		
cone	$1 - \sqrt{(u^2 + v^2)}$		
paraboloid	$1 - (u^2 + v^2)$		

Figure 4.11: Shapes produced by different weight functions and possible results when rendering points using the respective shape. $u, v \in [-1, 1]$. Figure taken and modified from [57].

4.3 Determining Point Sizes

Determining the point size is an important factor for speed and visual quality. A low point size improves the performance and reduces occlusion between points, but it also leads to holes in the rendered images. A larger point size reduces holes between points, but it also reduces performance and increases occlusion artifacts. The following sections describe 3 different algorithms for point-size determination.

4.3.1 Fixed Screen-Space Size

A fixed screen-space point size means that the same pixel size is used for all points. This mode is trivial to implement and sufficient in many cases. The main disadvantage of this mode is that it is prone to holes at close range and overdraw when the user zooms out.

4.3.2 Fixed World-Space Size

Instead of specifying a pixel size, the size is defined as the world-space radius in this mode. This leads to larger sizes at close range and reduced sizes for distant points.

Figure 4.13 shows how the field of view relates to the slope of the view frustum. Equation 4.3 shows how the inverse of the slope of the view frustum is used to project

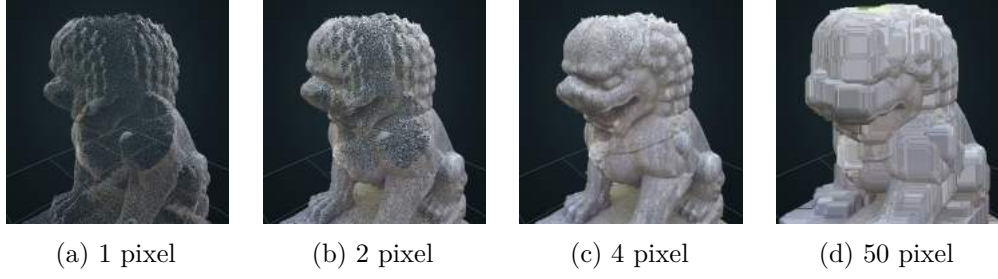


Figure 4.12: Fixed point sizes

from world-space radius to image-space pixel radius.

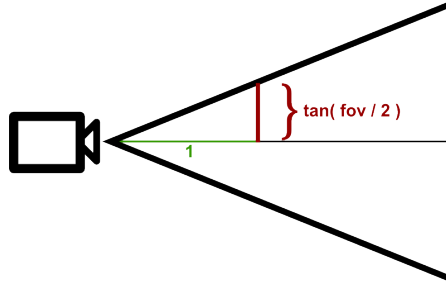


Figure 4.13: The slope of the view frustum.

$$slope = \tan\left(\frac{fov}{2}\right) \quad (4.2)$$

$$pixelRadius = \frac{screenHeight}{2} * \frac{worldRadius}{slope * depth} \quad (4.3)$$

4.3.3 Adaptive Point Sizes

The adaptive point size mode was developed for Potree to adjust point sizes to the level of detail. It hides the otherwise noticeable differences in point densities between different levels of detail.

A similar point size mode is available in Scanopy under the name "weighted point size" or "point size heuristic" [54]. The goal is the same, namely to avoid noticeable differences in point densities between different levels of detail, and to achieve a more or less closed surface representation of a point cloud. This can be done by making the point size equal to the distance between points, which is assumed to correspond to the spacing in Potree and the size of a cell in the inscribed grid in Scanopy.

The problem with setting the point size equal to the spacing is that in the hierarchical structure used in this work, higher detail nodes are rendered jointly with all their ancestors, including the root node, which means that points from higher detail nodes will be intermixed with points from lower detail nodes. For a leaf node, the point size



Figure 4.14: (a) Fixed pixel or world-space sizes lead to noticeable differences in density between different levels of detail. (b) Color-coded level of detail. (c) Point sizes have been adjusted to the level of detail to reduce holes without excessive overdraw. (d) Close-up where two different levels of detail meet with fixed point sizes. (e) With adaptive point sizes, points in the lower level of detail are enlarged, but points in the higher level of detail remain small to avoid excessive overdraw and occlusions. Whitby point cloud courtesy of GeoM [18].

can be increased up to the spacing between points, in order to close holes. However, the points in a lower detail node can not be enlarged beyond the spacing of its deepest child without occluding that node's points, as shown in Figure 4.15.

The point size heuristic of Scanopy adjusts the sizes of points in a node in a way that avoids excessive occlusions while still obtaining a closed surface. It accounts for the depth and the estimated point density of the descendants of a node and carefully adjusts the point sizes of inner nodes to close holes with as little occlusions and overdraw as possible.

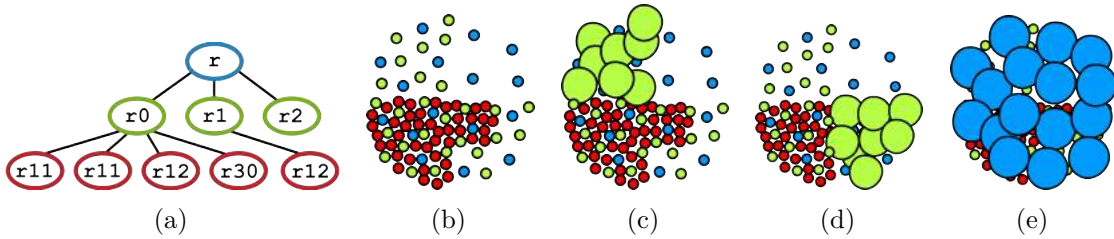


Figure 4.15: The difficulty of adjusting point sizes node-wise. (a) The hierarchy. (b) All points rendered at the same size. (c) Node r_2 is a leaf so its points can be enlarged without occluding higher details. (d) Points in r_1 can not be enlarged without occluding nodes with higher detail. (e) Points in the root can not be enlarged either without occluding higher levels of detail.

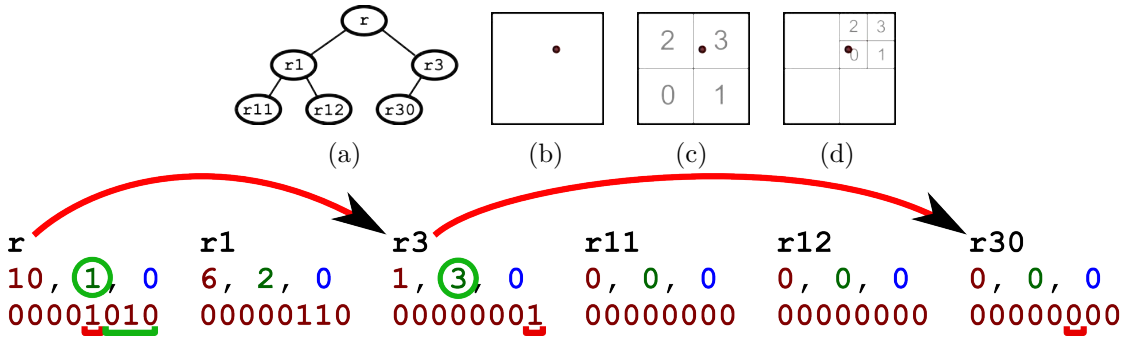
The adaptive point size mode of Potree resolves this issue by calculating the point sizes point-wise instead of node-wise. The idea is that the size of a point is adjusted to the spacing between points at a specific level of detail. Computing the level of detail of a point is the issue, since points in a single node may be part of different levels of detail. The level of detail in a specific region is equal to the highest-level visible node therein. It

is therefore obtained by traversing the octree from a point's node to the deepest visible node that encompasses that point. Visible, in this context, includes all nodes that are going to be rendered, i.e., nodes that passed the visibility determination step and whose points are loaded.

Equation 4.4 returns the world-space size of a point, given the spacing of points inside the root node and the level of detail. The resolution effectively doubles with each level and as a consequence, the point size is halved. The final pixel size is obtained by applying the projection from world-space radius to pixel size, described in the previous section.

$$worldRadius = \frac{spacingAtRoot}{2^{LOD}} \quad (4.4)$$

The octree traversal that computes the level of detail for each point is done on the GPU. The visible part of the hierarchy is passed to the shader, which then traverses the octree from the root in the direction of the currently processed point, until it reaches the deepest node that encompasses that point. At this time, for simplicity, traversal always starts from the root but it should be sufficient to start from the node that a point resides in. We will explore this in the future as part of performance improvements. Traversal stops when the deepest visible node that encompasses a point is reached. The number of nodes that were traversed is the LOD. Traversal is, theoretically, not necessary for the leaf nodes, since the LOD for all points in a leaf is the level of the leaf itself. Potree does not make an exception at this time, but we will explore this option in the future as another part of performance improvements.



(e) Encoding of the hierarchy in breadth-first order into a texture. The middle row represents the RGB texture values. The children bitset, which indicates which child nodes exist, is stored in the red channel. The green channel contains the offset to a node's first child in the texture. The blue channel is unused and filled with zeroes. The bottom row shows the binary representation of the children bitset that is stored in the red channel.

Figure 4.16: Computing the LOD of a point. (a) The hierarchy. (b+c+d) Traversing from the root to r3 and finally to r30. (e) The hierarchy, encoded into a RGB texture.

In order to pass the visible hierarchy to the vertex shader, it is stored in a 1-dimensional RGB texture in a breadth-first order. The 8 bits of the red value indicate

which of the 8 children are visible. The green channel contains the relative offset to the node's first child. These 2 properties, and the octree-size, are sufficient to traverse the octree from top to bottom. The blue channel is an empty filler to align the data to the RGB texture layout.

Figure 4.16 shows an example of a hierarchy, its encoding, and how the LOD is obtained for a specific point. An example follows that calculates the LOD for a point that may have originated from node r , $r3$, $r30$, which ultimately does not matter. What matters is the depth of the hierarchy at its location. In the first step, we find out in which of the root's children the point falls and compute its index, in this case 3. We then check the bit of the children bitset at index 3 to see whether there actually is a child node at this position or not. The bit is set to 1 so there is a child and traversal continues. Next, we need to jump in the texture to $r3$, which is the second child of the root node. The offset to the first child is already stored in the green channel. The number of 1-bits up to the index (underlined in green) stands for the number of children we have to skip to get to the child we are interested in. The same process is repeated for $r3$ to find out that we have to jump to $r30$. The offset to $r3$'s first child is 3 and since $r30$ is already the first child of $r3$, no additional child nodes have to be skipped over. Now at $r30$, the next child node that the point falls into is $r302$. However, the bit at index 2 is 0, which means that $r302$ does not exist or it is not visible. Traversal stops at $r30$, which is at octree level 2. The LOD for this point is therefore 2.

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. The texture look-ups and bit counting puts additional overhead on the vertex shader, but it also reduces the number of vertices and fragments that are required to fill holes.

As a final note, the adaptive point size does not create a truly closed surface since it is only concerned with adjusting the point sizes to the spacing of a level of detail and not to the true spacing between points. This leads to two issues with leaf nodes. The first issue are leaf nodes with a point density below the expected density at a certain octree level. In this case, choosing the spacing at the leaf node's level as the point size will not be sufficient to cover holes. Figure 4.17 shows a case where the density in the leaf node is lower than expected, even if only in one direction. The other issue are leaf nodes with a higher density than what would be expected at a certain octree level. This happens if a dense part of a point cloud falls into a small part of a node. The node is not split because a relatively small amount of points are added to it. The depth of the octree remains low in this region, which prompts the adaptive point size algorithm to increase the size of points in it, even though the points should be rendered with low point sizes due to their high density. Figure 4.18 shows an example of a point cloud with uniform density but different octree depths. Points in green have the same density as points on orange, but because the octree was not split further, are wrongly assumed to be of a lower level of detail.

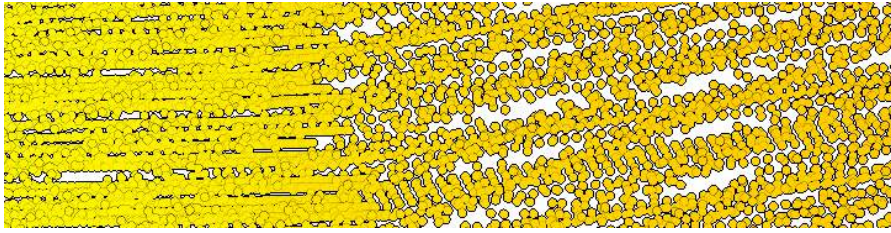


Figure 4.17: LIDAR scans often have high scan density in one direction and a lower density in the other. The level of the leaf node is not enough to conclude a meaningful point size. Apart from that, this is also a case where trying to close the gaps in both directions introduces heavy overlaps along the denser direction. The density becomes uniform in both directions in lower-resolution nodes, since they have a subsample density that is lower than the original density in both directions.

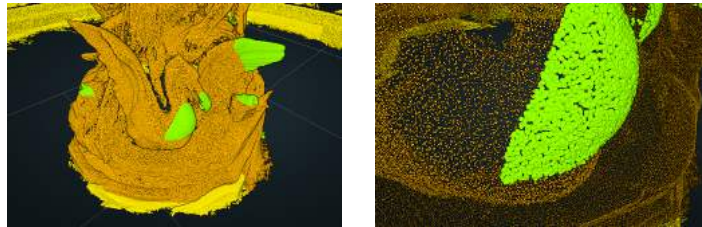


Figure 4.18: Point cloud with uniform density but different octree depths.

4.4 Eye-Dome Lighting

Illumination models are used to enhance the depth perception of a scene and to make the results look more pleasant. Without the shading provided by illumination, it becomes hard or even impossible to perceive shapes. Point cloud models with colors from photos or baked-in ambient occlusion already have a form of static illumination.

A large amount of point clouds do not contain surface normals, which are necessary for illumination models such as Phong or Blinn. Widely used point cloud formats, such as LAS, don't even have a designated normal property.

Eye-Dome Lighting (EDL) is a method that creates illuminated surfaces and outlines along silhouettes, without the need for normals [8]. The algorithm is conceptually similar to an edge-detection filter on a depth map. High differences to surrounding values lead to high responses. Surfaces that point towards the camera have relatively small differences in depth and result in low responses. The largest responses are experienced along silhouettes, which is the reason for the black outlines. Figure 4.20 shows an example of a rendering where EDL helps to perceive objects. An overview of the EDL workflow is shown in Figure 4.19.

Potree uses a variation of EDL with logarithmic depths. The response is obtained by computing differences in depth to surrounding samples, as shown in Equation 4.5. This equation sums up positive depth differences and normalizes the result to make

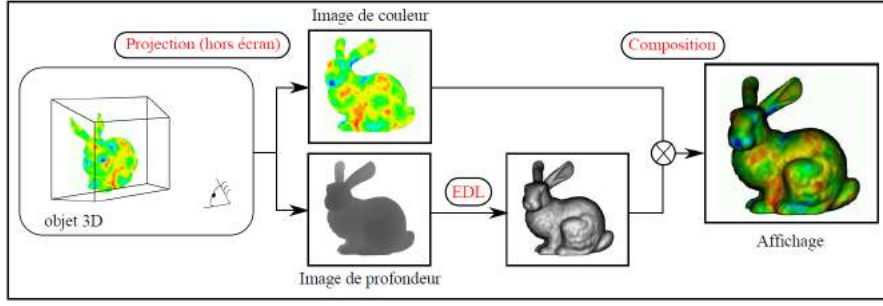


Figure 4.19: Eye-Dome Lighting workflow. First, colors and depths are rendered. The shading is then computed from the depth map and finally composed with the color values to obtain the shaded image. Image taken from *Visualisation scientifique de grands volumes de données : Pour une approche perceptive* [8].

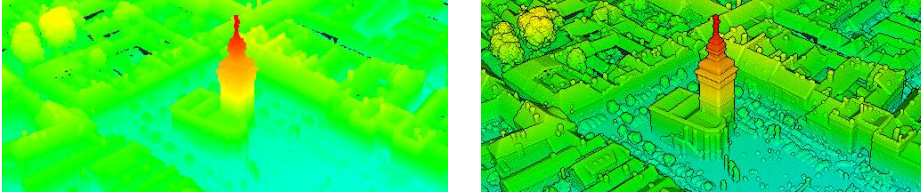


Figure 4.20: An elevation-colored point cloud without and with EDL. Retz point cloud courtesy of Riegler [50].

it independent from the number of samples. The logarithm to the base 2 results in a shading that is independent of the scale of the object. A small object close to the camera will be shaded the same way as an object with twice the size at double the distance.

The number of neighborhood samples could theoretically be increased to achieve a higher quality, but we were already satisfied with the results with four samples and the performance benefits of keeping the sample size low. By default, a 4-connected neighborhood is sampled by evaluating the depth values of the left, right, top and bottom neighbors. In addition, a radius parameter is provided that allows users to place the samples further away, e.g., 2 pixels or 3 pixels from the current pixel. Doing so increases the thickness of the outlines.

$$response = \frac{\sum_{i=1}^n \max(0, \log_2(depth) - \log_2(neighbor_i))}{n} \quad (4.5)$$

Equation 4.6 transforms the response to a shading factor. Multiplying the color value by the shading factor gives the final, EDL-shaded, result. The factor 300 is an empirically determined value for a suitable base shading strength. The *edlStrength* variable allows users to further modify the strength of the shading. A value of zero results in no shading, that is, the original color value remains unchanged. As *edlStrength* increases, the shading will grow stronger, resulting in a darker output image.

$$shade = \exp(-response * 300.0 * edlStrength) \quad (4.6)$$



Figure 4.21: Different values for EDL strength.

Figure 4.21 shows the results obtained by applying EDL with different strengths to a point cloud without colors and normals.

The implementation in Potree is based on the EDL shader source of CloudCompare [10], with some modifications.

- The linearization of the standard hyperbolic depth buffer has been removed. Instead of the hyperbolic depth-buffer values, logarithmic values to the base 2 are rendered to, and then read from the alpha component of a floating-point texture. This is mainly done because Potree has no access to WebGL’s depth buffer, but using a custom logarithmic depth buffer has some advantages as well. We directly render the non-normalized $\log_2(depth)$ into the texture, which is completely independent of the near and far clip plane. With hyperbolic depth buffers, the linearization is required to resolve this dependency. Otherwise, the shading would be affected by the clip planes. Logarithmic values also feature a more favourable distribution of precision between close and distant objects, which makes EDL in Potree less prone to staircasing artifacts due to sudden jumps in depth.
- The EDL shader of CloudCompare supports a light direction vector. In Potree, objects are always lit from the front.
- CloudCompare computes the shading at full, half and quarter resolution, and then combines the results. Potree uses a single full resolution pass, since it already produces satisfactory results.



Figure 4.22: Point cloud courtesy of WeissAG [73].

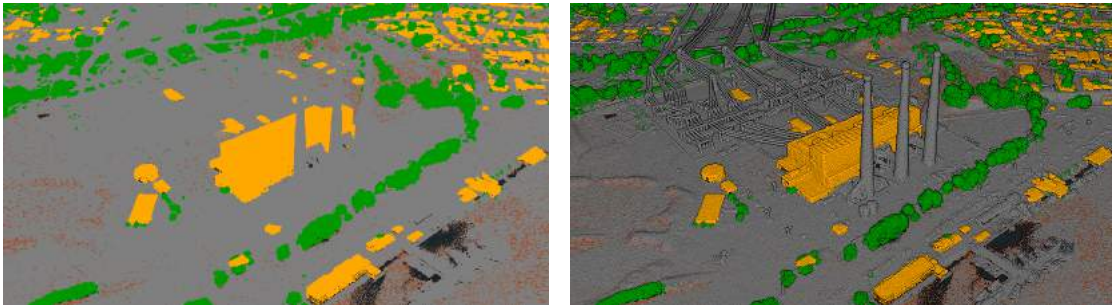
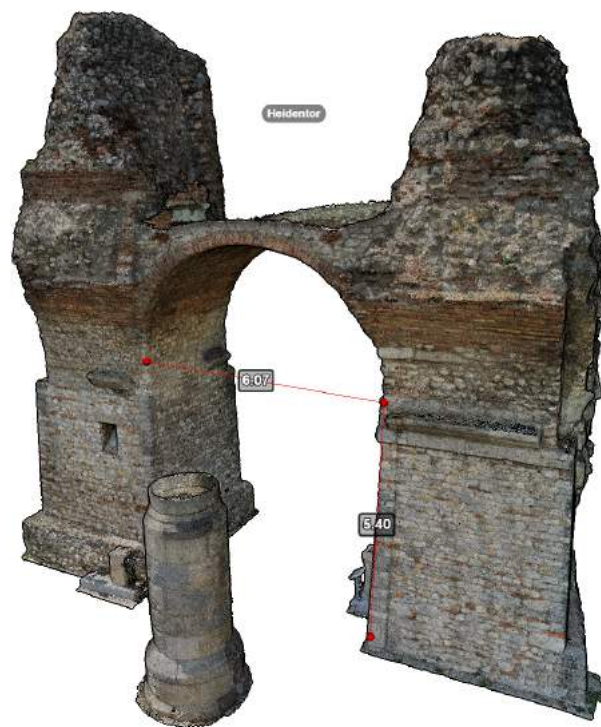


Figure 4.23: Rendering point classification without and with EDL. CA13 point cloud courtesy of OpenTopography and PG&E [42].

Implementation and Features



This chapter covers implementation details and functionality of the Potree viewer.

5.1 WebGL

WebGL [72] is a variation of OpenGL for web browsers, which is intended to provide GPU rendering capabilities to web pages on a wide variety of devices, including desktop PCs, notebooks, mobile phones and tablets. As such, it is based on the lowest common denominator, OpenGL ES 2.0, to ensure that WebGL applications will run on as many devices as possible. WebGL Extensions [71] expose additional features that are not included in the base specification of ES 2.0 and WebGL. However, they are not guaranteed to work on all WebGL conforming devices or browsers.

Some features in Potree, such as High-Quality Splatting and point interpolation, make use of WebGL Extensions like floating-point textures or modifying the depth value in fragment shaders. During setup, Potree checks if the respective extensions are available and if they are not, the features will be disabled. Due to this, High-Quality Splatting, point interpolation and Eye-Dome Lighting are usually not available on mobile devices.

Potree uses the three.js [65] rendering library to handle scene graphs and draw calls. Direct use of the WebGL API and GLSL are limited to special cases where three.js does not provide some necessary functionality. This includes GPU-based point picking and shaders for Eye-Dome Lighting and point-based rendering.

A limitation of WebGL is that it does not provide bit operations, which are required by the adaptive point size mode. The necessary bit operations, namely counting bits up to a certain index and checking if a certain bit has been set, were implemented in the custom functions `numberOfOnes` 5.1 and `isBitSet` 5.2. These arithmetic implementations of bit operations are far slower than the natively available bit operations in later versions of GLSL. In an OpenGL 4.5 implementation of this shader, they have shown to be a considerably bigger bottleneck than the texel-fetches that are also required by this method.

Listing 5.1: WebGL workaround for the GLSL 4.0 `bitCount` function.

```
float numberOfOnes(float number, float index) {
    float tmp = mod(number, pow(2.0, index + 1.0));
    float numOnes = 0.0;
    for(float i = 0.0; i < 8.0; i++) {
        if(mod(tmp, 2.0) != 0.0) {
            numOnes++;
        }
        tmp = floor(tmp / 2.0);
    }
    return numOnes;
}
```

Listing 5.2: WebGL workaround for $(\text{number} \& (1 \ll \text{index})) > 0$.

```
bool isBitSet(float number, float index){
    return mod(floor(number / pow(2.0, index)), 2.0) != 0.0;
}
```

5.2 Asynchronous and Parallel Execution

A major requirement for rendering is to keep the application responsive at all times. Tasks should not block the control flow and leave enough cycles to maintain a steady framerate.

Any Javascript code, with the exception of Web Workers, is executed in one single thread. Javascript applications do not maintain a custom main loop. Instead, they provide callbacks that are invoked by the browser when certain conditions, such as timeouts, the start of a new frame, or user input, have been met. In between the execution of callbacks, the browser will handle updates to the page and render elements. A callback should return quickly or it will prevent the browser from updating the page, effectively freezing it.

Javascript offers asynchronous functions to schedule callbacks to be invoked at a later time, or to execute specific tasks, but not code, in parallel. The popular `setTimeout` function, for example, executes a callback after a set amount of time has passed. Due to the single-threaded nature of Javascript, callbacks will run one after another, never at the same time. The asynchronous version of the `XMLHttpRequest` function tells the browser to load a resource and invoke a callback during progress or after loading has finished. The resource loading itself is done in parallel, and multiple invocations of `XMLHttpRequest` may result in multiple resources being loaded at the same time, but as with `setTimeout`, the execution of the callbacks is done one after another.

The asynchronous callback philosophy is also applied to the render loop in Javascript, as shown in Listing 5.3. The `requestAnimationFrame` function takes a callback that is invoked at the browser's own discretion. In order to keep the loop running, the callback invokes another `requestAnimationFrame` call on itself, which will be executed during the next frame. A browser decides when to invoke the callback based on a few conditions. If a tab is active, it will usually try to maintain 60 frames per second or less if that is not possible. If a tab is hidden or running in the background, the browser may not invoke the callback to reduce CPU usage and increase battery life on mobile devices.

Listing 5.3: A basic render loop. Instead of maintaining an endless loop, Javascript applications repeatedly send a request for the browser to invoke a callback before the next repaint.

```
function loop(timestamp) {
    requestAnimationFrame(loop);

    update(timestamp);
    render();
}
```

```
};
requestAnimationFrame(loop);
```

Web Workers are the exception to the usually single-threaded environment and run Javascript code parallel to the main thread. They do, however, have quite strict limitations. Code that is executed by a Worker runs in its own execution environment and has no access to elements of the main thread. It is not possible to directly change a html page or access any object of the main thread. Instead, Workers communicate with the main thread through messages. The main thread sends the Worker a message, usually with a workload to be processed, and the Worker will reply with another message, usually the result of the task or information about the progress. Messages are processed by callbacks which, as usual, will be executed sequentially with other pending callbacks.

The difference between asynchronous functions and Web Workers is that asynchronous functions may do specific things, such as loading a file, in parallel, but any Javascript code that is provided to the function will be executed in the main thread. Web Workers, on the other hand, do execute Javascript code in parallel.

Potree uses asynchronous functions to load files from a remote location, and Web-Workers are used to parse and prepare the loaded data in a parallel thread, as shown in Figure 5.1.

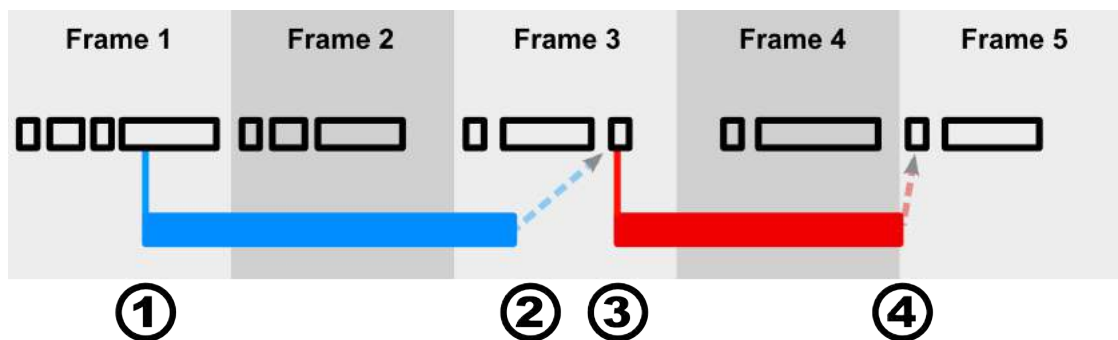


Figure 5.1: Timeline in Potree. Black rectangles: Tasks and callbacks that run inside the main thread, including key and mouse events. The biggest task in each frame is the update and render callback. Blue: Parallel loading of a resource by the browser. Red: Parallel execution of Javascript code by a Web Worker. (1) Start loading data for a node with XMLHttpRequest. (2) Data has been loaded. The finish callback is scheduled to run inside the main thread. (3) The finish callback is executed inside the main thread and it spawns a Web Worker that prepares the loaded data in a parallel thread. (4) The Web Worker finishes and the result will be handled by a callback in the main thread, which sends the loaded data off to three.js. The node is now ready to be rendered.

5.3 Tools and Interaction

5.3.1 Point Picking

Point picking is required by various navigation and interaction operations such as zooming to a point or creating measurements. Mouse-point-intersections can be calculated either on the CPU or on the GPU. Both options were evaluated and are described in the following sections.

On the CPU

The CPU approach iterates through all points inside nodes that are intersected by the mouse. For the intersection test, points are treated as disks with a certain radius.

Advantages of this approach are that its implementation requires less complexity, it does not require potentially slow GPU-stalling operations and it may be implemented in an asynchronous fashion, although we did not explore the latter. Disadvantages are that it requires additional effort to account for projected pixel sizes and visibility of points as calculated by vertex and fragment shaders.

On the GPU

This approach does point picking by rendering the region around the mouse on the GPU, and evaluating the result on the CPU. All nodes that are intersected by the mouse will be rendered but instead of colors, point indices are written to the output. A small window around the mouse is fetched from the GPU to extract the index of the point closest to the mouse.

A unique index is assigned to each rendered point. It consists of the sequence number of the point within its node and the sequence number of the node itself. The point's sequence number is 3 bytes and the nodes sequence number 1 byte. Due to these size constraints, point picking works for nodes with up to around 16 million points each and for up to 255 nodes at once. The node sequence number starts at 1 so that an index value of 0 can safely be used for empty regions. These limits are not enforced in Potree, as both are unlikely to be reached. The octree generation creates nodes with far less than 16 million points and the culling of all nodes that are not intersected by the mouse significantly reduces the amount of nodes that will be rendered during picking. The latter case might occur in rare situations and could be avoided by ignoring nodes with larger volume. Due to the lack of *gl_VertexID* in WebGL, the point sequence numbers are stored in an additional vertex attribute array. The node sequence number is passed as a uniform value. The fragment shader writes the former into the RGB channel and the latter into the alpha channel.

After the points have been rendered into a frame buffer, a small window, for example 17x17 pixels, around the mouse pointer location is read back to the CPU using `readPixels()`. Reading a single pixel is not sufficient because there may be holes between points, in which case the pick function should snap to the closest point within the window. Zero values indicate empty regions. The value closest to the center that is larger than zero is

the index of the point that is closest to the mouse pointer location. This index is then split into the node and point sequence numbers which are used to retrieve the actual point attributes, most importantly its coordinates.

The main advantage of this approach is that it does point picking based on the same point sizes and occlusions that are seen by the user. The disadvantage is that the use of `readPixels()` requires the CPU to wait until the GPU has finished rendering, which can cause significant performance drops. In order to reduce performance losses, only those nodes which are intersected by the mouse pointer are rendered. In practice, this method still performs faster than the CPU approach we explored, even though `readPixels` synchronizes CPU and GPU.

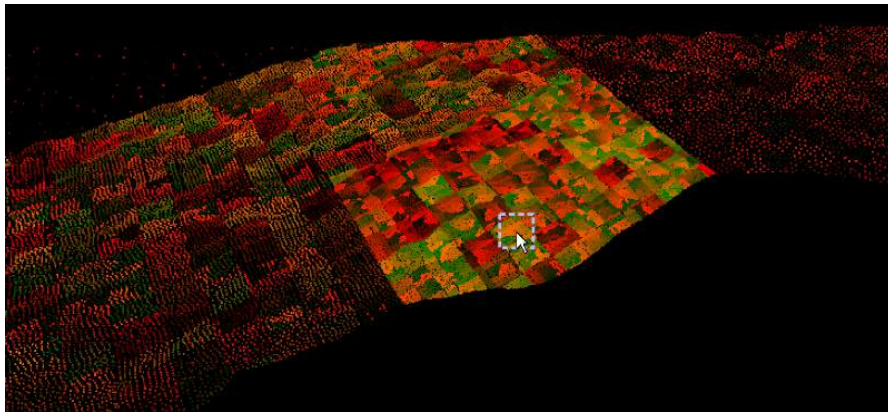


Figure 5.2: Output of the pick render pass. The point cloud is denser near the mouse pointer because nodes that do not intersect the mouse are not rendered. Patches are noticeable because the point sequence number, which is stored in the RGB channels, starts at zero in each node.

5.3.2 Navigation

Different tasks require different tools. The same is true for navigation; no single navigation mode is suitable in all situations.

Three navigation modes were implemented in Potree: `OrbitControls`, `FirstPersonControls` and `EarthControls`.

OrbitControls is a set of controls which let users orbit around a target or pivot, similar to how satellites orbit the earth. The difference to trackball controls is that the object remains upright, i.e., orbit controls feel like moving around and under or above the object without touching it, whereas trackball controls feel like rotating the object itself in any possible way.

The distance to the pivot can be changed using the mouse wheel. The amount of change increases with the distance to the target, thus making it independent of the object scale. It may be used for planet-sized objects as well as small statues. Double clicking on any point will zoom the camera towards that point and change the target to

the point position. Dragging while holding the right mouse button will pan the screen which, in essence, translates camera as well as target by the exact same value.

OrbitControls are intuitive, easy to use and yet powerful at the same time, which is the reason they were chosen as the default mode in Potree.

FirstPersonControls or **FlightControls** give users a fly- or walk-through-like experience. This is useful for navigating through enclosed spaces or vast landscapes. Panning (right mouse button) and click-to-zoom work exactly the same as with **OrbitControls**. Dragging the left mouse button rotates the camera and allows the user to look into all directions. The W, A, S, D, or alternatively the arrow keys, move the camera forwards, to the left, backwards or to the right.

EarthControls provide a navigation mode similar to the one in Google Earth [21]. The left mouse button allows users to drag and drop the object along a ground plane with the height of the targeted point. By dragging the right mouse button, the user rotates around the clicked location. Finally, the scroll wheel can be used to zoom towards the targeted position. This mode is designed to quickly and precisely navigate vast, open landscapes.

5.3.3 Clip-Boxes

Clip-boxes allow the user to focus on a particular area of interest, as shown in Figure 5.3, by highlighting points inside the box or by clipping points outside of it. They are useful to cull away points which would otherwise distract or occlude.

A popular use case for clip-boxes are interiors of buildings. Without clipping areas, the user would have to navigate inside rooms in order to be able to make sense of the data. Clip-boxes allow users to cull away walls, ceilings or whole parts of buildings and analyze their interiors from the outside, as shown in Figure 5.4.



Figure 5.3: Using a clip-box to focus on a single object. Retz point cloud courtesy of Riegler [50].

For both, highlighting and clipping, the vertex shader checks if a point is inside one of the boxes. It then either modifies its color (highlighting) or discards it (clipping).

Algorithm 5.1 shows how the vertex shader handles rendering of clip-boxes. The amount of clip-boxes and their inverse world matrices are passed to the shader. For each vertex, the shader loops through all boxes and transforms the points to the clip-box

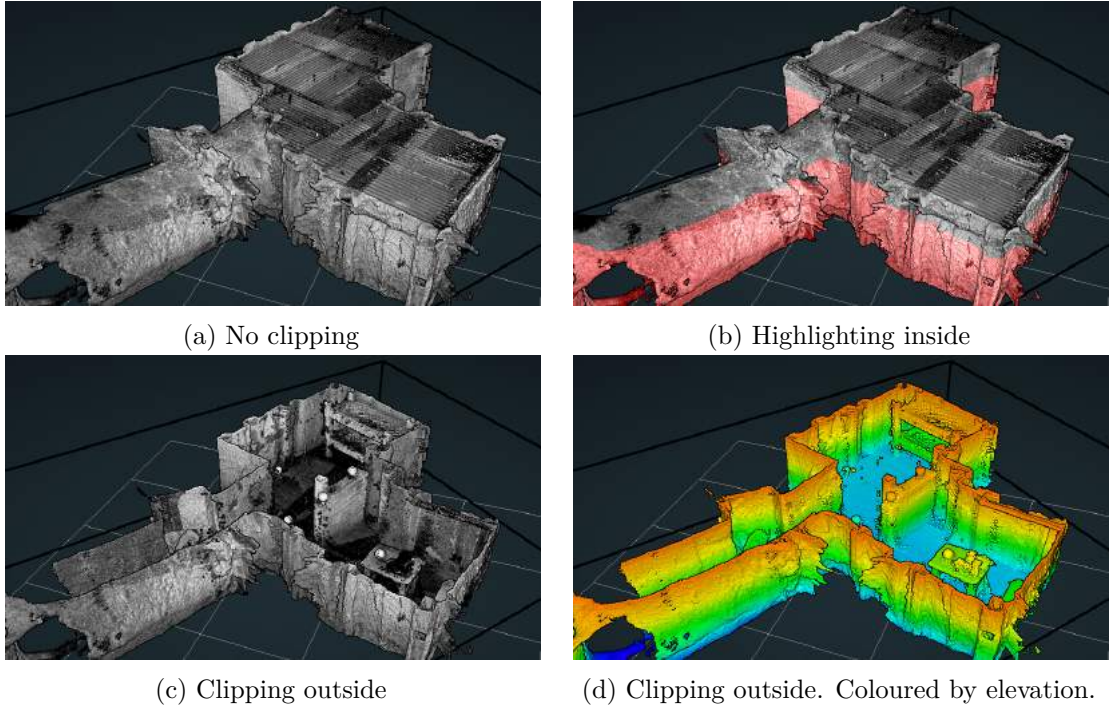


Figure 5.4: Using a clip-box to reveal the interior. Point cloud courtesy of Ogle, Tucker and Hicks [38]

object-space. In object-space, all clip-boxes have a width, height and depth of 1 and are located at the origin. A point is therefore inside the clip-boxes if all its coordinate components are larger than -0.5 and smaller than 0.5. Depending on the clip-mode, the point is now either highlighted if it is inside this interval or discarded if it is outside. Discarding a vertex is not directly supported by WebGL. In order to discard it, its coordinate is set to any value outside of the clip region.

Algorithm 5.1: render profile 3D

```

1 insideAny = false;
2 foreach clip_box do
3   transform point to clip box object space;
4   inside = -0.5 <= point.xyz <= 0.5;
5   insideAny = insideAny || inside;
6 end
7 if insideAny AND highlight_inside then
8   highlight point - increase red channel;
9 else if !insideAny AND clip_outside then
10  discard point;

```

5.3.4 Measurement

Potree offers distance, area and angle measurement tools. Although they display different information, they work very similarly and are therefore implemented in the same MeasuringTool class. Each measure consists of an array of vertices, and some flags that indicate how they will be displayed. A distance measure, for example, will have distance labels activated but angle labels and closing edge deactivated. An angle measure, on the other hand, will have a closing edge and angle labels activated but no distance label. Area measurements usually have distance labels, area labels and closing edge activated. It is, however, also possible to define any other combination of flags.

The measurement tools use the previously described point picking method to allow real-time drag and drop of the vertices. Due to the GPU picking, the drop-off location is exactly the point that is currently hovered or the closest one around it.

Area measurements are done with respect to the ground plane. Differences in height are ignored. Results are displayed without units, since Potree does not make any assumption about the coordinate units.

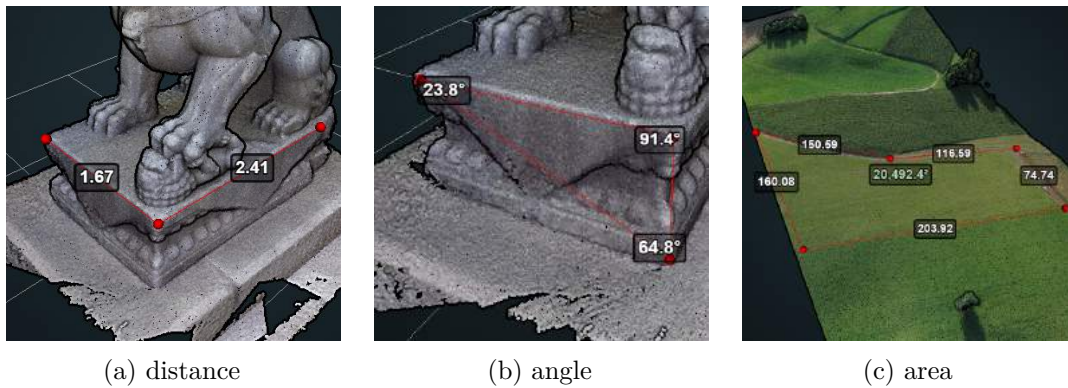


Figure 5.5: Different measurement tools. (c) Cutout of a highway construction point cloud courtesy of sigeom sa [60].

Measurements are rendered without depth-testing, after the scene has been rendered and post-processed. They are therefore not occluded by point clouds and unaffected by Eye-Dome Lighting.

The texture of a label is updated by drawing into a two-dimensional canvas element. two-dimensional canvas elements are part of the html standard and can be inserted into a html page like any other element. The canvas API offers functionality to draw text, lines and splines into the element. For the measurement labels, canvas elements are drawn to and their content sent to the GPU, without adding them to the web page.

5.3.5 Height Profile

A height profile, or elevation profile, consists of a polyline where each edge, or segment, has a width and an infinite height. Points that are within the boundary of a segment

are part of the profile. Figure 5.6 shows an example of a profile with 3 vertices in a three-dimensional and an ortographic two-dimensional view.

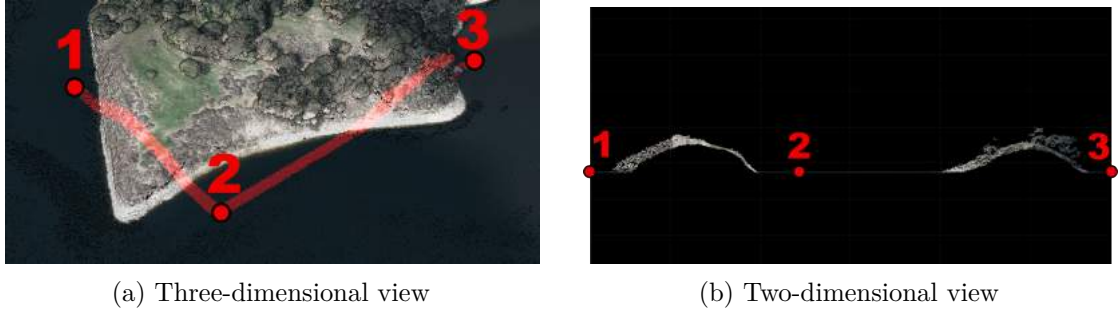


Figure 5.6: A profile that consists of 3 vertices that span 2 segments. (a) By default points inside the profile are highlighted. (b) In the two-dimensional view, parallel projections of each segment are placed next to each other. Point cloud courtesy of Open Topography and PG&E[42].

Height profiles allow users to obtain cutouts of the data and measure and analyze without beeing blocked or distracted by surroundings. Distance and other measurement tools, as well as navigational tools, can be constrained to points within the profile by switching to the clip-outside mode.

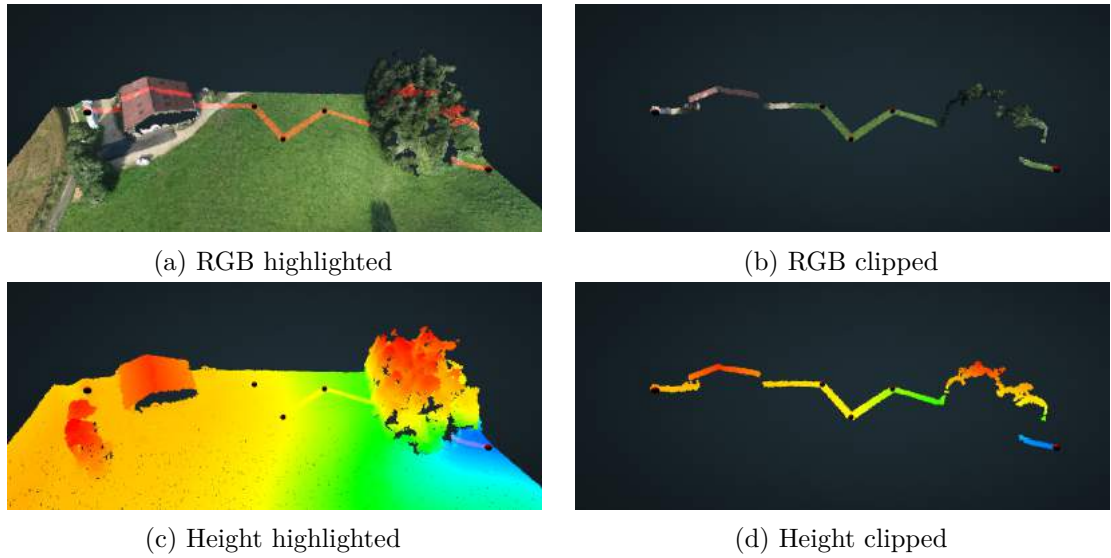


Figure 5.7: The same height profile rendered in 2 different color and clip modes. Point cloud courtesy of sigecom sa[60]

Retrieving Points in a Profile

The `Potree.PointCloudOctree.getPointsInProfile()` function returns all points inside the given profile. It comes in 2 versions. A synchronous one, which immediately returns a result for nodes in memory, and an asynchronous one, which creates a `ProfileRequest`. The request frequently invokes callbacks whenever new points have been loaded. To keep the application responsive, only a small number of nodes are loaded and processed each frame. The request can be stopped at any time by calling `request.cancel()`. Cancelling is useful for dense point clouds, when even small profiles may contain hundreds of thousands or millions of points.

Unlike three-dimensional rendering, which does a screen-projected-size-first order to traverse nodes, elevation profile retrieval does a level-order traversal. The reasoning behind this is that the two-dimensional representation of the elevation profiles employ orthographic projections, which map lower-level nodes to larger screen-projected sizes anyway. Level-order ensures that the result converges in an even fashion towards the final profile. For the two-dimensional representation, this means that elevation-profile requests can be canceled with minimal or even negligible impact on the result after a certain number of points were fetched. For a 500x300 pixel canvas, a threshold of 20.000 points works well in practice. Additional points will mostly occlude or be occluded by previously fetched points.

Rendering a Height Profile

Two different options for the rendering of height profiles were implemented. Profile segments may be rendered using clip-boxes with infinite height in the three-dimensional view or projected and drawn in an orthogonal two-dimensional view.

The three-dimensional view provides the same rendering options as clip-boxes. Points inside the profile may be highlighted, points outside the profile clipped or clipping may be disabled. Measurement tools and navigation are constrained to the visible points, allowing users to work on the area of interest without being obstructed or distracted by surroundings.

The two-dimensional view uses the asynchronous version of `getPointsInProfile(profile, callback)`. The callback's `onProgress` function continuously draws new points into the two-dimensional canvas element as they are streamed in. If the profile is modified, either by moving a vertex or changing its width, the current request is cancelled and a new one will be created. The request is also cancelled after a certain number of points have been drawn. Additional points are unlikely to improve the result and stopping the request reduces unnecessary computational overhead. Due to its asynchronous nature, the user can continue using the viewer while the two-dimensional view is gradually refined. The 2D method aligns all profile segments along the x-axis resulting in side-by-side orthogonal projections of all segments, as shown in Figure 5.8.

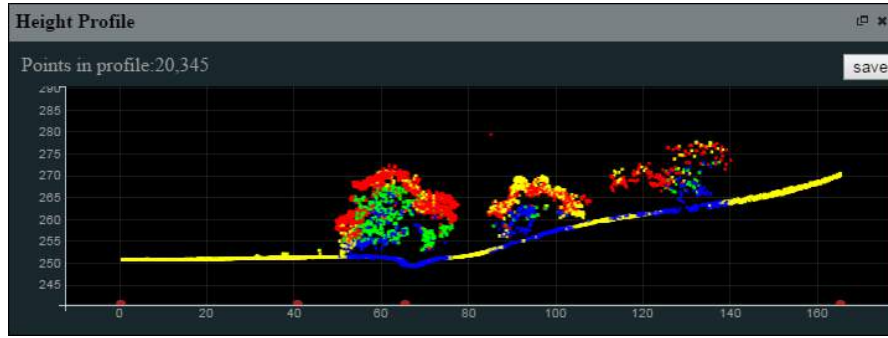


Figure 5.8: 2D projected profile. Start and end of each segment are indicated by red dot at the bottom line.

Issues

Each segment is treated as a box with a start, end, infinite height and a width. As each segment is treated individually, overlapping parts will return the same points multiple times. This issue is prevalent at adjacent nodes with sharp angles, as shown in Figure 5.9.



Figure 5.9: (a) Adjacent segments overlap and return the same points twice. Possible improved boundaries to avoid intersections are suggested in (b) and (c). CA13 point cloud courtesy of Open Topography and PG&E [42]

Potree provides the option to save the points in a profile in a LAS file. The effects of the point threshold is much more noticable in this case. Zooming into the data will expose differences in point density because some regions were loaded up to a higher level of detail than others, as shown in Figure 5.10. At this time, a complete profile without threshold is not feasible for a fully client-side point cloud viewer, considering that a profile may contain millions or even billions of points.

5.3.6 Annotations

Aside from presenting raw three-dimensional models, content providers may also want to highlight and provide information about specific points of interest. Annotations enable developers to insert text labels, move the camera towards a pre-defined location and show short descriptions. Annotations are implemented using HTML elements in order to take advantage of their extensive functionality.

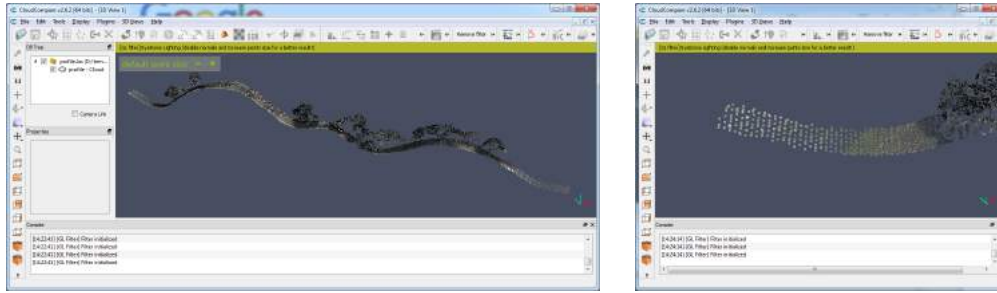


Figure 5.10: Exported profile, opened in the CloudCompare point cloud viewer. The point threshold works when viewing the data at a certain distance. Close-up views reveal density differences as further processing has been cancelled after reaching the threshold.



Figure 5.11: Three Annotations on the lion model.

Listing 5.4 shows how an annotation with all available features is added to the scene. Annotation and camera position have to be specified. Clicking on the annotation will move the user to the camera position. The other attributes are optional. If no target is specified, the annotation position is used as the target. The title defines the text of the label. A sequential number starting from one will be used if this attribute is omitted. Descriptions are shown when the mouse moves over the annotation. Descriptions may consist of text as well as HTML elements.

Listing 5.4: Adding an annotation

```
viewer.addAnnotation(annotationPosition, {
    "cameraTarget": target,
    "cameraPosition": cameraPosition,
    "title": "Lion",
    "description": "Description <b>can include html tags</b>"
})
```

});

Clicking on an annotation will move the camera to the user-specified position. Instant change of position and direction may lead to disorientation and confusion, as users lose the context of how they got to their destination. The movement towards the specified target and camera position is therefore implemented through a smoothly interpolated animation path. An animation duration of 800ms has proven to be short enough to swiftly reach the target, but long enough to grasp how the viewer got to the destination.

Putting the mouse over an annotation displays the optionally defined description. The description may contain HTML tags, which allows developers to include standard text as well as hyper-links, images and more, as shown in Figure 5.12.



Figure 5.12: Descriptions with links and images.

5.4 Georeferencing

Georeferencing is the process of linking two-dimensional or three-dimensional coordinates to real-world locations. This is essential for various applications, such as creating and displaying maps, creating scenes representing the real world, doing measurements and more.

On a globe, locations may be specified through angular coordinates, namely latitude and longitude, which give the angles in north-south and east-west direction. Another option are cartesian coordinates in the earth-centered, earth-fixed (ECEF) coordinate system. In this system, the origin is at the center of mass of the earth and coordinate units are in meters. A third option are localized map-projections, where small patches of the earth are taken and projected, so that the real-world ground is aligned with the virtual scene ground.

The shape of the earth approximately resembles an oblate spheroid. Any projection of a spheroid on a plane will cause a certain amount of distortion. The commonly used Web Mercator projection, also known as EPSG:3857, shows increasingly larger distortions towards the poles. Countries close to the pole appear bigger than they are, in relation to those near the equator, as shown in Figure 5.13.

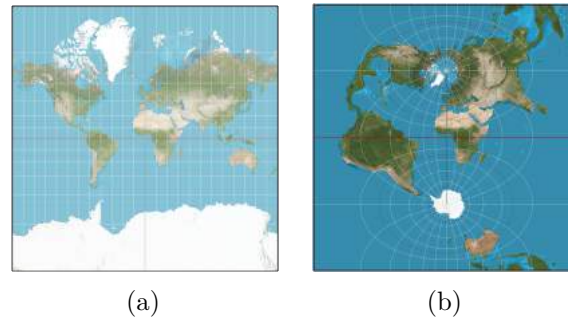


Figure 5.13: Distortions caused by map projections. (a) makes Greenland appear as big as Africa whereas (b) makes Greenland appear much smaller. Images taken from Wikipedia Transverse Mercator Projection [74]

In order to reduce distortion errors, some projections are only valid for certain regions, as shown in Figure 5.14. Within the specified extents, distortions are small enough such that measurements produce useful results. The Swiss coordinate system, CH1903 / LV03 or EPSG:21781, is defined within the boundaries of Switzerland. Other countries also have their own spatial reference systems. The Universal Transverse Mercator coordinate system (UTM) is a world-wide system that splits Earth in 60 zones around the equator. Each zone has a width of 6 degrees to reduce east-west distortion. Zones are then projected cylindrically to reduce north-south distortions. The distortion increases as coordinates get closer to the border of each zone. Surveyed data may not always fall into the extent of a single map projection. In such cases, a data set may be split and different projections used for each tile. Alternatively, if a certain amount of distortion is acceptable, the same projection may be used for the whole data set, even though part of the data is outside the extent.

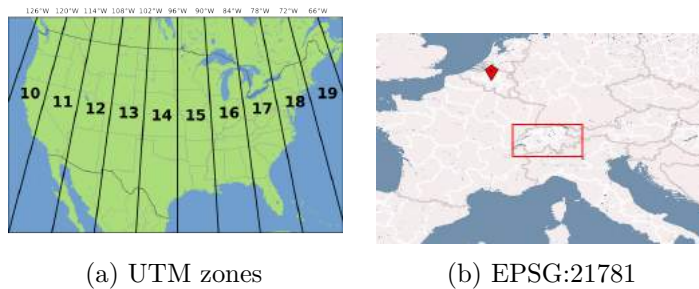


Figure 5.14: (a) UTM zones in the USA. Image courtesy of Chris Murphy [36] (b) A coordinate system for switzerland. Screenshot taken from spatialreference.org [16]

Potree offers basic support for georeferencing. During conversion, the user has to specify the spatial reference system of the data set in proj4 format, which is then stored in the metadata of the converted data set. The coordinates themselves are left untouched. If a point cloud with a spatial reference system is loaded by the Potree Viewer, it will

create a two-dimensional map overlay that displays the extent of the point cloud as well as measurements that were added to the three-dimensional scene. The map overlay was created with the `openlayers3` library and uses `OpenStreetMap` as its map provider. The spatial reference system of the map is Web Mercator. The `proj4js` library allows transforming between the spatial reference systems of point cloud and map.



Figure 5.15: Single precision floating-point numbers do not have enough precision to store many types of georeferenced coordinates. Point cloud courtesy of sigecom sa [60].

An issue with georeferenced coordinates are their large values. The minimum of the extent of the Swiss coordinate system, for example, is $[485869.5728, 76443.1884]$. The precision of floating-point numbers, however, decreases as values get larger. This issue can manifest itself as jitter during movements or stripes, as coordinates on one axis are rounded to the closest value that the floating-point number is able to represent, as shown in Figure 5.15.

This issue has been solved as follows:

- For calculation purposes, the converter handles coordinates as double precision numbers.
- For storage, 32bit fixed-point numbers are used. 64bit double values require too much disk space and 32bit floating point values are not precise enough for large models. A more detailed description of the format is given in Section 5.5.
- For rendering, floating point coordinates are used to exploit the GPU's proficiency with floating point operations, but the coordinates in object space are kept close to the origin, $(0,0,0)$, to preserve precision. The world matrix of the point cloud would usually be used to translate the point cloud from its object space to its georeferenced position in the world-space. The view matrix then turns the position of the camera to the new coordinate system origin. The world-view transformed point cloud coordinates are then relative to the nearby camera origin, rather than the very distant world origin. The translational parts of both matrices effectively cancel each other out. However, for this to work, both matrices would have to use double precision to be able to store the huge translational parts and to ensure an accurate result of the matrix multiplication. Since the version of `three.js` we use only offers single precision floating-point values, we can not rely on this technique.

As a workaround, we do not transform the point cloud into a georeferenced space at all. Instead, the first point cloud remains at the origin and subsequently loaded point clouds are placed relative to the first one. If georeferenced coordinates are required, e.g., to synchronize the scene with a map overlay, coordinates can be transformed from the local scene coordinate system to georeferenced coordinates by a custom and more precise transformation. A downside of this workaround is that the coordinate system origin remains near the first point cloud and any other model that is placed too far away from it will still suffer from precision issues.

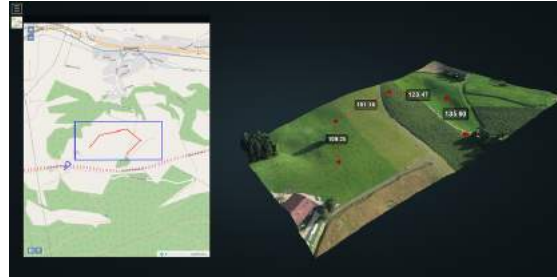


Figure 5.16: Point cloud with map overlay. The map shows the extent of the point cloud, the position of the camera and the distance measurements. Point cloud courtesy of sigecom sa [60].

5.5 Data storage

Each octree node contains a point cloud model that is stored in its own file on disk. Potree offers a choice of 3 different point cloud file formats to choose from. The default is a custom binary format that is based on the format of Scanopy, with the difference that coordinates are stored as fixed-point numbers instead of floats. The other two supported formats are LAS and LAZ.

5.5.1 The Potree Format

Potree stores point after point and then attribute after attribute. A metadata file specifies which attributes are available. The only mandatory attribute are the Euclidean coordinates of a point. Optional attributes are RGBA, intensity and classification. A list of attributes and their data types is shown in Table 5.1.

The format is based on the file format of Scanopy, with the main difference that floating-point coordinates are quantized to, and stored as fixed-point values.

Coordinate Quantization

Floating-point values are not a suitable data type to store coordinates. Double precision floating-point values have a high precision but require 64bit, which is very inefficient in

Table 5.1: Point Attributes

Attribute	Format	Bytes
XYZ	3 * uint32	12
RGBA	4 * uchar	4
intensity	short	2
classification	uchar	1

terms of disk space usage. Single precision floating-point values require only 32 bits, but they do not have enough precision for many large models. This is because even though floating-point values have a large range, they still have a very limited amount of digits. It makes them more precise for small values, where those digits are free to be used for the fractional part, but less precise for large values.

Coordinate quantization, in this context, means to transform the coordinate values from a floating-point representation to a fixed-point representation. Fixed-point values have a uniform precision over the whole coordinate range. The available bits can be traded off between the coordinate range and the number of fractional digits. With 32 bits, it is possible to store 2^{32} , just over 4.294 billion, unique values. If we assume millimeters as units, we end up with the ability to store a model with an extent of 4294 kilometers at millimeter precision.

Fixed-point values are stored in 32bit unsigned integers, because no native fixed-point data type exists. The fractional part of a fixed-point value is preserved by scaling the value up by a factor of 10^{digits} .

Equation 5.1 shows how a coordinate is quantized from a float or double value to an integer with a precision given by *scale*. A scale of 0.001 moves the decimal point 3 digits to the right, effectively transforming the units from meters to millimeters. The remaining fractional part is lost during the conversion to an integer. Subtracting the minimum of a node’s bounding box ensures that quantized values start from zero. Doing so also avoids out-of-bounds errors, which will happen when already large coordinates grow even larger through a division by a scale lower than 1.

$$x_{\text{quantized}} = \frac{x - \text{boxMin}}{\text{scale}} \quad (5.1)$$

During loading, the reverse is done to transform back from a fixed-point representation to a floating-point representation.

Compression of quantized coordinates

Compression of quantized coordinates is currently under development and not part of the public Potree repository, yet.

Quantization lends itself well to a simple form of compression. After quantization, we get integers in a range between zero and $\text{boxWidth} / \text{scale}$. This range is usually a small fraction of the whole 32 bit range. The idea of compression through quantization is to store a number in as many bits as necessary, not more.

Equation 5.2 gives the number of bits required to store quantized coordinates. According to this equation, 19 bits are enough to store a model with a size of 400 meters at millimeter precision, a reduction of about 40%.

$$bits = ceiling(log_2(\frac{boxWidth}{scale} + 1)) \quad (5.2)$$

Additional savings are obtained by quantizing not to the bounding box of the whole point cloud, but the bounding box of each node. With each level, the bounding box size is halved. Cutting the value space in half reduces the amount of required bits by exactly 1 bit per axis, for a total of 3 bits per three-dimensional coordinate. At octree level 8, coordinates require 3 bytes less than coordinates at level 0.

5.5.2 The LAS and LAZ Formats

LAS is a widely used point cloud format, which is supported by most point cloud applications. It also stores coordinates as fixed-point integers. Instead of specifying each desired point attribute, users decide between pre-defined record formats with a fixed collection of attributes. Due to this, users are forced to store attributes even if they are not in use, which leads to increased file sizes.

Passing the *-output-format LAS* flag to the PotreeConverter will save the contents of each node in the LAS format.

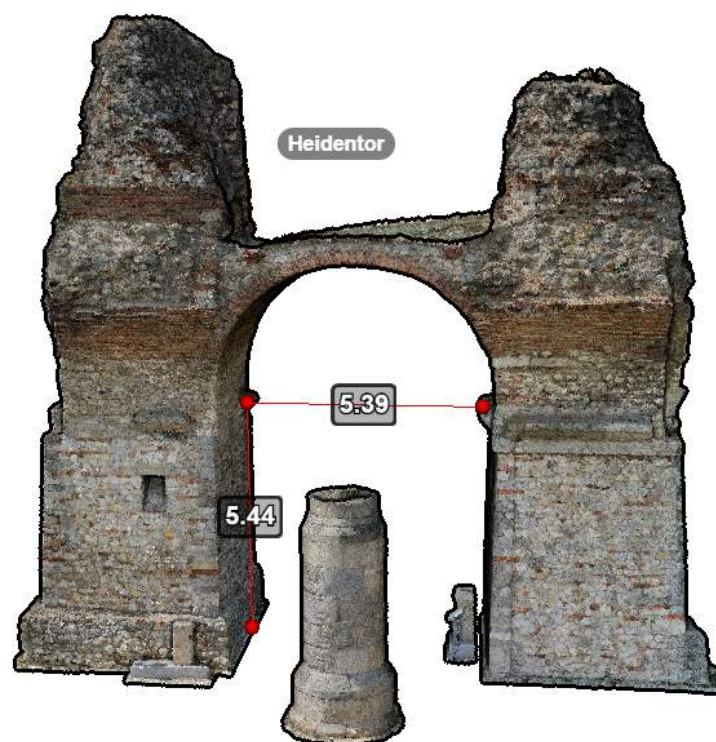
A compressed version of LAS, the LAZ format, was created by Martin Isenburg [49] to provide a storage-efficient way of saving the contents of LAS files. LAZ files tend to be about 70 to 90 percent smaller than LAS.

Passing the *-output-format LAZ* flag to the PotreeConverter will save the contents of each node in the compressed LAZ format. The reduced data size speeds up transfers between server and clients. The Javascript version of the LAZ decoder, ported by Uday Verma and Howard Butler using emscripten [14], is used to decompress the contents on the Javascript client. Multiple Web Workers are then used to decompress multiple LAZ files in parallel.

Decompression is a computationally expensive task, and due to the nature of the code generated by emscripten, each instance of the decoder requires a few hundred MB of memory. Using LAZ compression is therefore less suited for mobile devices with a low amount of memory, or machines with very fast connections where transfer of larger files is faster than transfer and decompression of smaller files. Multi-core devices with 4GB RAM or more will greatly benefit from the faster transfer of compressed files, however, especially if network speed is low.

CHAPTER 6

Results



This chapter contains performance evaluations of the converter and viewer, applications of potree by third parties, and our own renderings of point clouds that were provided by third parties.

6.1 Performance

This section lists performance evaluations of the converter and the viewer.

Table 6.1: Test Systems

system	Memory	CPU	GPU	Disk
Notebook	16GB	i7, 2.3GHz	GTX 860M	1TB, 5400rpm
SSD	16GB	i7, 2.3GHz	-	500GB, SSD SATA 3
Server	32GB	i7, 3.40GHz	-	3TB, 7200rpm
S4 Active	2GB	-	Adreno 320	-
S7 Edge	4GB	-	Adreno 530	-
iPhone 6S	2GB	-	PowerVR GT7600	-

Table 6.1 shows the list of test machines. The converter performance was tested on the *Notebook*, *SSD* and *Server* systems. *Notebook* and *SSD* are identical, except for the disk drive. The disk used by the *SSD* system has a read performance of 540MB/s and a write performance of 520MB/s, according to its specification. The viewer performance was tested on the *Notebook* system and on three mobile devices, a Samsung Galaxy S4 Active, a Samsung Galaxy S7 Edge and an iPhone 6S.

The Potree converter performance is limited by disk speed and CPU. Table 6.2 shows that the *SSD* system outperforms the *Notebook* system, even though it is identical, except for the disk. The *Server* system, with a significantly higher CPU performance but an expected disk performance between *Notebook* and *SSD*, outperforms both.

The disk speed bottleneck stems from the large amount of data that has to be read and written. Due to the out of core-scheme, points may be read from disk and written back to disk multiple times, thus further increasing the amount of I/O operations.

A large part of the CPU bottleneck comes from a single-threaded implementation of the octree build-up, even though it is a potentially highly parallelizable task. Currently, each point is processed one after another. A second thread exists to read points from the input, while the build-up thread processes points that have already been read. Multi-threaded build-up methods will be investigated in the future. Projects like Entwine [26] by Hobu, and a modified Massive-PotreeConverter [34] by the NLeSC, have already shown promising results in multi-threaded and distributed build-up methods.

The performance of the renderer is mostly affected by the GPU and partially by the CPU. It has been evaluated in frames per second, to get an impression of the overall performance, and duration in milliseconds to measure the execution time of specific rendering tasks on the GPU.

We assume at least 60FPS to be a flawless real-time performance, 30FPS to be an acceptable real-time performance, and anything between 10FPS and 30FPS to be

Table 6.2: PotreeConverter performance showing conversion duration and points per second (pps) for different data sets and test machines.

data set	points	Notebook		SSD		Server	
		duration	pps	duration	pps	duration	pps
CA13	17.7B	-	-	-	-	16h	305k
Whitby	7.7M	12s	642k	10.6s	726k	6s	1,247k
Eclepens	68.7M	437s	157k	-	-	290s	236k
Subseamanifold	26.9M	-	-	-	-	57s	467k
Heidentor	25.8M	92s	280k	38.5s	670k	20s	1,275k
Chowilla	692.3M	-	-	-	-	3,484s	198k
Dechen Cave	40.0M	167s	240k	95s	421k	115s	347k
Lion	4M	5.9s	670k	5.3s	754k	3.5s	1,125k

interactive. Frame rates below 10FPS are undesirable and should be avoided by reducing the point budget, or by disabling features such as EDL and adaptive point sizes.

In order to achieve 60 frames per second, each frame must finish within $1 / 60 = 16.6$ milliseconds. The actually available time to render objects is usually lower, in practice, for reasons such as running the application inside a browser environment, which acts as an additional layer to the GPU, inefficiencies in the communication between CPU and GPU, or poor GPU utilization.

Figure 6.1 shows a performance comparison of Potree on different devices. The goal of this performance evaluation is to show that Potree is able to achieve real-time or interactive rendering performance in real-world use cases on desktop and mobile devices. The point cloud is therefore rendered into the body of a maximized browser, on each device. Some results are capped at 60 frames per seconds, because browsers automatically enable vsync. For the *Notebook* system, we were able to turn vsync off, and achieve higher frame rates, by starting Chrome with the command-line option `-disable-gpu-vsync -disable-d3d11`.

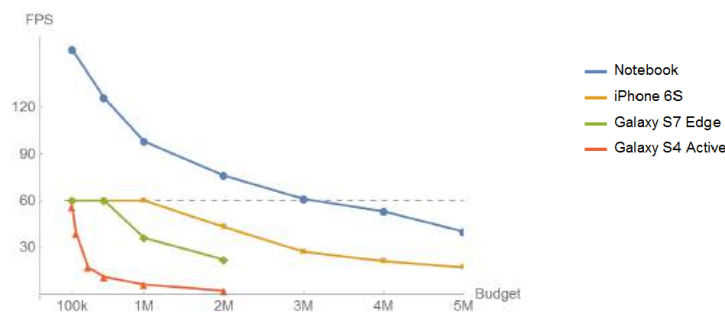


Figure 6.1: Frames per second for the Matterhorn data set with different point budgets.

Figure 6.2 shows the time it took to render a point cloud with different point budgets on the *Notebook* system. The relation between point budget and duration is mostly

linear, with a small deviation between a budget of 0.5 and 1 million points, which happens when the GPU automatically adjusts the clock speed to the higher workload. Durations were measured with the WebGL *EXT_disjoint_timer_query* extension, by invoking the three.js render calls in between the start and end of a query. At a point budget of three million, rendering the whole scene takes about 15.2 milliseconds, with the majority of 15.1 milliseconds attributed to the rendering of the point cloud. The dashed line represents the 16.6 milliseconds mark, which has to be undercut in order to maintain a frame rate of 60 frames per second.

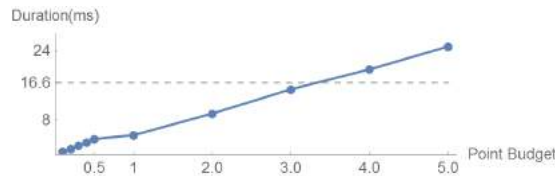


Figure 6.2: Time to render the point cloud with different point budgets. Tested on the *Notebook* system.

EDL is a post-processing shader which renders a quad over the whole page. As a result, one fragment shader is executed for each pixel. Figure 6.3 shows how the performance scales with the resolution of the canvas and the number of neighborhood samples. The point budget has no significant effect on the duration of the EDL pass, because by the time it starts, the point cloud has already finished rendering.

On the *Notebook* system, the EDL pass takes 0.35 milliseconds to render into a full-page canvas element with 1.8 megapixel, and with the default of 4 neighborhood samples. That is about 2.1% of the 16.6 milliseconds render time that is available to maintain a frame rate of 60 frames per second.

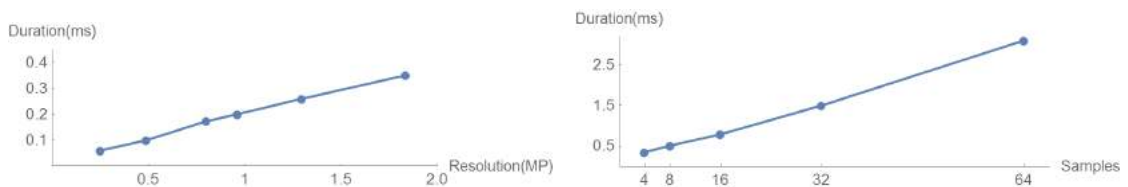


Figure 6.3: Eye-Dome Lighting performance linearly scales with the screen resolution and the number of neighborhood samples.

Figure 6.4 shows a comparison of performance between the fixed and adaptive point size modes. All the points are rendered with a size of 1 pixel for this test, even in adaptive mode, since we are interested in the cost of calculating the size, without the impact of rendering points at different sizes. Adaptive sizes are significantly slower at the same point budget. Their advantage is, however, that they reduce the number of points that have to be rendered to achieve satisfying results in the first place.

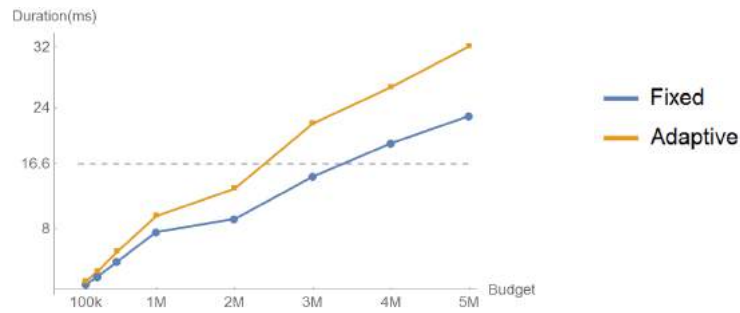


Figure 6.4: The time it takes to render a point cloud with fixed or adaptive point sizes at various point budgets.

6.2 Applications

This section introduces some uses of Potree by third parties.

laspublish

Rapidlasso recently integrated Potree into its lastools package [27] as a new module called laspublish. The lastools package is a widely used toolset for analysis, evaluation and processing of point clouds.

OpenSFM

OpenSFM is a master thesis project by Matthias Adorjan with the aim to create a "free and fully accessible structure-from-motion system, based on the idea of collaborative projects like OpenStreetMap" [1]. It combines Potree and the Cesium globe renderer, which allows users to explore georeferenced point clouds on a globe in a web browser. Users can upload new data sets that will then be available on the globe, after a processing step on the server backend.

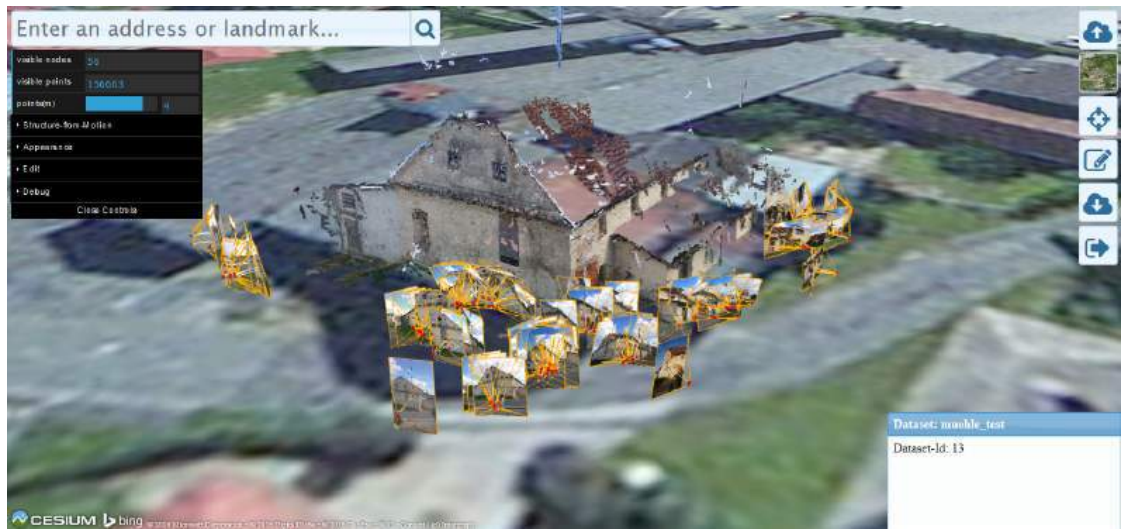


Figure 6.5: Screenshot of OpenSFM, showing a point cloud and the images that were used to create it. Image taken from OpenSFM master thesis [1]

Via Appia



Figure 6.6: Screenshots of the Via Appia viewer.

The “Mapping the Via Appia in 3D” was a project by the Netherlands eScience Center [37] with the aim of “Creating a virtual 4D reconstruction of one of the earliest and strategically most important roads of the Roman world” [32]. The data set is not available for view online due to copyright issues. The full source code for their framework, using Potree for the visualization of point clouds, has been published on github [39].

The Netherlands AHN2 data

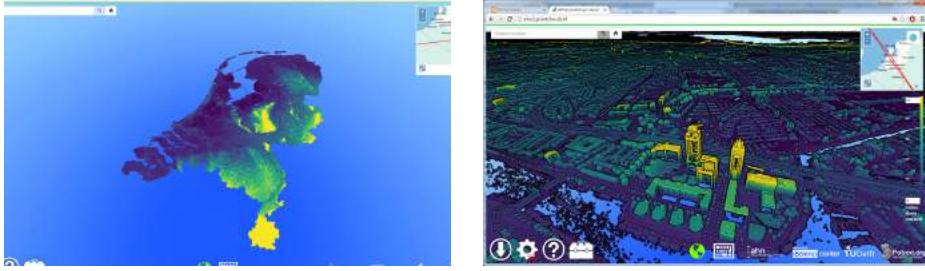


Figure 6.7: Screenshots of the AHN2 viewer.

The Actueel Hoogtebestand Nederland (AHN) is a program by the Netherlands to survey the whole country. The second scan campaign, AHN2, resulted in a data set consisting of 640 billion points, which was made publicly available as open data. This data set requires 7.68 terabyte disk space, at 12 byte per point for uncompressed coordinates.

The Netherlands Escience Center [37] processed this data and released a web viewer, based on Potree, that allows the public to view this data set online. [2] Their work was based on an older and slower version of the PotreeConverter. According to their calculations, processing the full dataset would have taken at least 100 days. In order to speed this process up, the Netherlands Escience Center developed a parallel conversion process, the Massive-PotreeConverter [34], which tiles the whole data set into multiple smaller ones, processes each tile with a separate PotreeConverter instance, and finally merges the result into a single, large octree. The final octree consist of 597 billion points, partitioned into just over 38 million compressed laz files.

6.2.1 Sagrada and Eixample

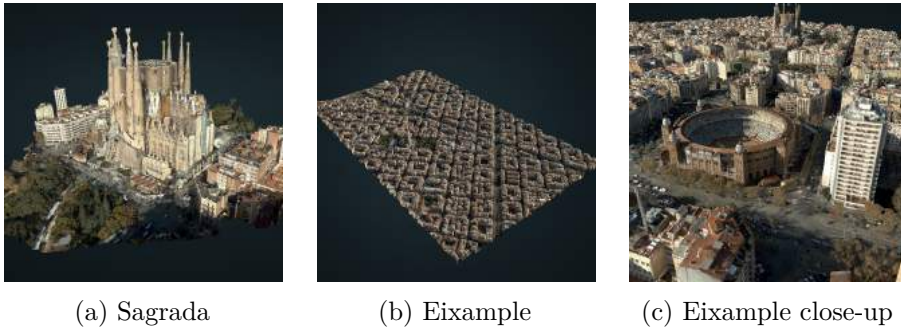


Figure 6.8: Screenshots of the Temple Expiatori de La Sagrada Família and the Example district, Barcelona.

The *Temple Expiatori de La Sagrada Família* is a church in the district of Eixample, Barcelona [53]. Point cloud visualizations of Eixample and the church, using Potree 1.2,

were published by the *Institut Cartografic i Geologic de Catalunya*(ICGC) on their web page [6].

6.2.2 PotreeViewer by SITN

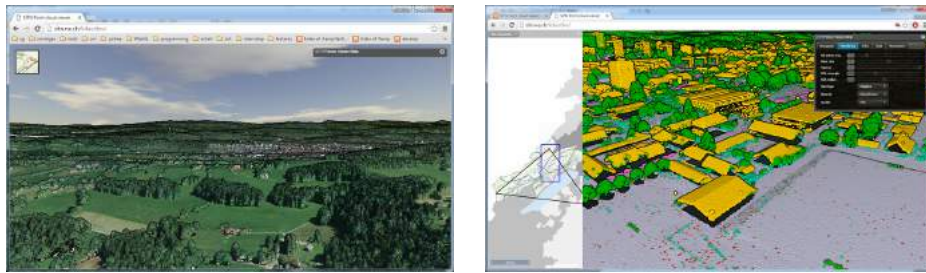


Figure 6.9: Screenshots of the Neuchâtel point cloud with RGB and classification color modes.

Based on Potree 1.3, SITN developed and published their own Potree Viewer user interface to display a point cloud of Neuchâtel, Switzerland [46]. One of its main features is a map overlay and a two-dimensional height profile view, both of which were later merged into Potree 1.4. Still unique to the SITN Potree Viewer is the wide range of map overlay sources, in addition to OpenStreetMap.

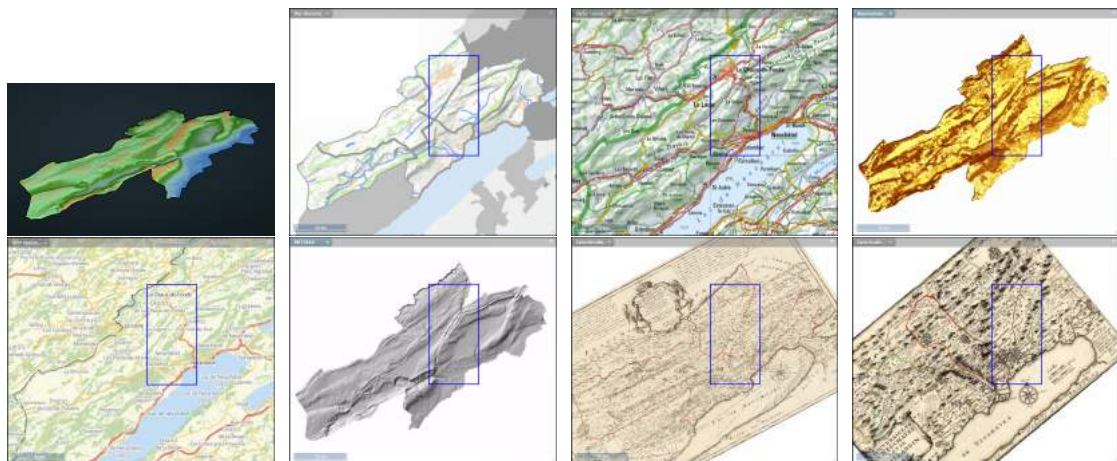


Figure 6.10: The point cloud and some of the various map overlays you can chose from.

6.2.3 ARA - Airborne Research Australia



Figure 6.11: Screenshots taken from pages uploaded by ARA. [4] [5]

Airborne Research Australia, ARA [3], is a research institute at Flinders University, Australia. ARA started to offer most of their data sets for free to the scientific community and they recently published some of their data sets online, using Potree and laspublish.

Potree as a client for Entwine



Figure 6.12: Screenshot taken from <http://potree.entwine.io> [25].

Hobu and NLeSC have developed a modified version of Potree, which supports Entwine as an alternative to the PotreeConverter, and Greyhound as an alternative to a file server. Instead of loading an octree node from a file server, this modified version sends a request to Greyhound for points within a volume at a specific level of detail.

A showcase for data sets that are processed by entwine, served by greyhound and rendered by Potree, is available at <http://potree.entwine.io> [25].

6.3 Showcase

In this section, we present screenshots of point clouds rendered with Potree.



Figure 6.14: Point clouds of the building site of NVIDIA’s new headquarter at two different times. Point clouds courtesy of NVIDIA [15].

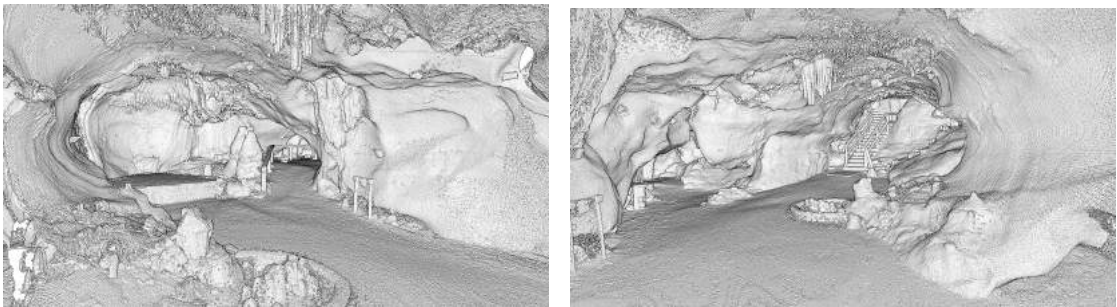


Figure 6.15: This point cloud of the Dechen cave [12] in Iserlohn consists of 40 million points. Dechen cave point cloud courtesy of Hämmerle et.al., who captured it as part of a research project [22].



Figure 6.13: Harvest4D [23] was an EU funded project, with Potree as a part of it, that dealt with the acquisition and rendering of 3D models of the real world. The screenshots depict a point cloud of the Arènes de Lutèce.



Figure 6.16: Westend Palais point cloud courtesy of Faro [17].



Figure 6.17: This point cloud of the Matterhorn was created by senseFly[58], Drone Adventures[13], and Pix4D[43]. The photogrammetry software of Pix4D was used to compute a three-dimensional model out of 2188 images of the Matterhorn. The result is a point cloud that consists of about 300 million points. Matterhorn point cloud courtesy of Pix4D.



(a) Heidendor



(b) Dover Mill ruins

Figure 6.18: Uses of Potree for archaeology. (a) The Heidendor in Carnuntum, Austria. Point cloud courtesy of the LBI ArchPro [31]. (b) Dover Mill ruin point cloud courtesy of Prologue Systems, LLC [48].



(a) Lion Head

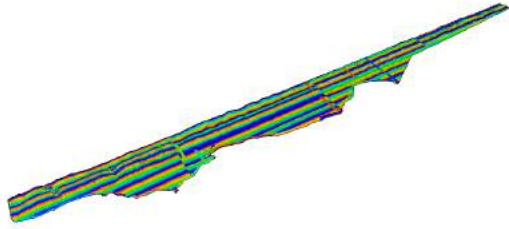


(b) Grave

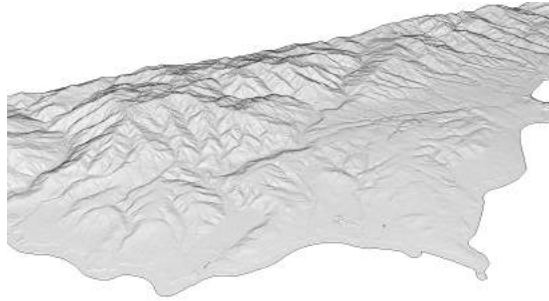
Figure 6.19: Uses of Potree for archaeology. (a) Lion head point cloud courtesy of Simone Garagnani [30]. (b) Grave point cloud courtesy of Prof. Heinz Runne, Paul Banse and Philippe Kluge [51].



Figure 6.20: This point cloud of Retz, Austria, consists of a combination of an airborne laser scan over the whole area, and terrestrial laser scans at the town center. Retz point cloud courtesy of Riegl [50].



(a) Flight lines



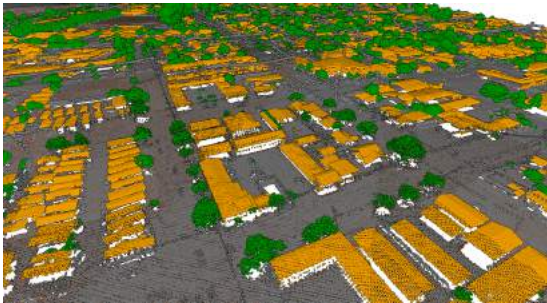
(b) San Simeon area with EDL



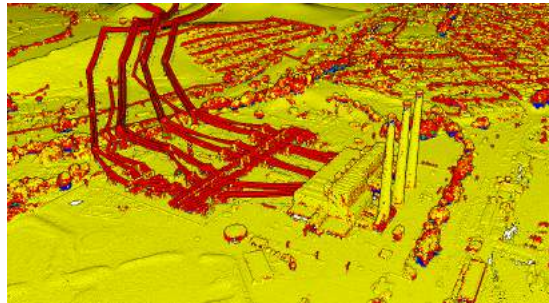
(c) Hearst Castle



(d) Morro Rock



(e) Classifications of Morro Bay area



(f) Return numbers of Morro Bay area

Figure 6.21: Screenshots of an airborne laser scanning campaign, with the short name CA13_SAN_SIM, of the San Luis Obispo County coast, including the San Simeon and Morro Bay areas. The data set consists of 17.7 billion points, and includes RGB colors, intensity, classification, return number, and flight source. CA13 point cloud courtesy of Open Topography and PG&E [42]

Conclusion and Future Work

We presented a state-of-the-art point cloud viewer that is able to render point clouds with hundreds of billions of points in real time in standard web browsers, by utilizing a slightly modified version of the modifiable nested octree (MNO) structure [56]. This octree structure, which contains a low-resolution subsample in its root node and gradually higher-resolution subsamples in higher-level nodes, allows Potree to render distant regions at a lower level of detail, thereby reducing the amount of points that have to be loaded and rendered. Regions that are outside the view-frustum are ignored altogether. We extended the original MNO structure such that the octree hierarchy itself is stored in another octree, the hierarchy-octree, which allows to load the hierarchy on-demand as well. This became necessary because millions of nodes are required to store larger point clouds, and storing the whole hierarchy in a single blob led to increased initial load times. We also decided to replace subsampling on a grid with Poisson-disk subsampling. Poisson-disk subsamples are evenly spaced subsamples with a minimum distance between points, and they exhibit more naturally looking and pleasant patterns.

We have also shown a new adaptive point-sizing method for octree structures, which adjusts sizes point-wise to the level of detail. Closed surfaces are obtained by setting the point size equal to the spacing between points at a certain level of detail. There are two settings where this mode is especially beneficial. It fills in the initially large gaps while points are streamed in, and it also fills in gaps when rendering at a low point budget.

To achieve a better visual quality, we implemented state-of-the-art methods such as Eye-Dome-Lighting [8], which provides illumination for point clouds without normals, and high-quality surface splatting [70] with weighted screen-aligned circles [55]. In addition to that, we also added a nearest-neighbor-like interpolation shader that creates renderings similar to Voronoi diagrams by drawing points as screen-aligned paraboloids rather than circles [57]. This shader provides a trade-off between the high quality of surface splatting and the high performance of basic screen-aligned circles.

The viewer is accompanied by a converter that builds-up the octree structure out of a point cloud. The converter has been tested with point clouds up to 17.7 billion points.

Thanks to the NLeSC, who built a modified and parallel version of the converter [33], we were able to test the viewer with a data set of 597 billion points.

The source code of the viewer, Potree, is available at:

<https://github.com/potree/potree>

The source code of the converter is available at:

<https://github.com/potree/PotreeConverter>

Future work includes

1. Generating low-resolution nodes in the octree introduces aliasing in a way similar to downsizing an image without applying a filter. These aliasing artifacts can be reduced by averaging colors instead of taking the first pick. This approach essentially produces the equivalent of a mip map for point clouds.
2. Low-level performance improvements from draw-call and render-order optimizations. At this time we let three.js handle most of the rendering. We found that significant improvements are possible by rendering front-to-back and by reducing the number of uniform parameter updates.
3. Storing multiple nodes in a single file. HTTP range requests can be used to stream specific nodes by reading only parts of a file. Currently, each node is stored in its own file, which makes some operations, such as deletion or copying of the point cloud files, relatively slow.
4. Improvements to the adaptive point sizing. At this time, nodes are not further split by the converter unless they contain enough points. This lets the adaptive point-size mode wrongly assume a low level of detail, thus increasing the point size.
5. Improved Poisson-disk sampling. As of now, the Poisson-disk sampling is only done within each node separately. Combinations of nodes do not uphold the Poisson-disk property and therefore exhibit increased density on borders of same level nodes or intersections with different level nodes. Another issue is that the sample quality relies on the ordering of the input. Badly arranged input sets produce Poisson-disk samples with holes or small cuts.
6. Additional sampling modes like two-dimensional grid, voxel grid, random choice, etc. Some of them may significantly improve build-up times.
7. WebVR is an actively developed API, which allows developers to create virtual reality applications in web browsers. Implementing WebVR in Potree will allow users to view and interact with point clouds in VR.

Bibliography

- [1] Matthias Adorjan. “The OpenSFM Database”. MA thesis. Favoritenstrasse 9-11/186, A-1040 Vienna, Austria: Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2015. URL: <https://www.cg.tuwien.ac.at/research/publications/2015/Adorjan-2015/>.
- [2] *AHN2 online viewer*. Accessed: 2015-10-22. URL: <http://ahn2.pointclouds.nl/>.
- [3] *Airborne Research Australia*. Accessed: 2016-01-27. URL: <http://www.airborneresearch.org.au/>.
- [4] Airborne Research Australia. *Adelaide Hills*. Accessed: 2016-01-27. URL: <http://www.airborneresearch.com.au/potree/ironbank1/examples/ironbank1.html>.
- [5] Airborne Research Australia. *Part of an area in South Australia*. Accessed: 2016-01-27. URL: http://www.airborneresearch.com.au/potree/examples/0107MH_02bcd_square_RGB.html.
- [6] *Basilica and Expiatory Church of the Holy Family and Eixample neighbourhood in 3D*. Accessed: 2015-10-31. URL: http://betaportal.icgc.cat/wordpress/sagrada_familia_eixample_3d/.
- [7] Mario Botsch et al. “High-quality Surface Splatting on Today’s GPUs”. In: *Proceedings of the Second Eurographics / IEEE VGTC Conference on Point-Based Graphics*. SPBG’05. New York, USA: Eurographics Association, 2005, pp. 17–24. ISBN: 3-905673-20-7. DOI: 10.2312/SPBG/SPBG05/017-024. URL: <http://dx.doi.org/10.2312/SPBG/SPBG05/017-024>.
- [8] Christian Boucheny. “Visualisation scientifique de grands volumes de données : Pour une approche perceptive”. In: (2009).
- [9] Robert Bridson. “Fast Poisson Disk Sampling in Arbitrary Dimensions”. In: *ACM SIGGRAPH 2007 Sketches*. SIGGRAPH ’07. San Diego, California: ACM, 2007. ISBN: 978-1-4503-4726-6. DOI: 10.1145/1278780.1278807. URL: <http://doi.acm.org/10.1145/1278780.1278807>.
- [10] *CloudCompare*. Accessed: 2016-01-26. URL: <http://www.danielgm.net/cc/>.

- [11] Robert L. Cook. “Stochastic Sampling in Computer Graphics”. In: *ACM Trans. Graph.* 5.1 (Jan. 1986), pp. 51–72. ISSN: 0730-0301. DOI: 10.1145/7529.8927. URL: <http://doi.acm.org/10.1145/7529.8927>.
- [12] *Dechenhöhle - Tropfsteinhöhle und deutsches Höhlenmuseum Iserlohn*. Accessed: 2016-09-08. URL: <http://www.dechenhoehle.de/>.
- [13] *Drone Adventures*. Accessed: 2016-09-08. URL: <http://droneadventures.org/matterhorn>.
- [14] *Emscripten C/C++ to javascript transpiler*. Accessed: 2015-10-31. URL: <https://github.com/kripken/emscripten>.
- [15] *Endeavor, NVIDIA’s new headquarter under construction*. Accessed: 2016-09-01. URL: <http://endeavor.nvidia.com/>.
- [16] *EPSG:21781 on spatialreference.org*. Accessed: 2016-08-18. URL: <http://spatialreference.org/ref/epsg/21781/>.
- [17] *Faro*. Accessed: 2016-09-08. URL: <http://www.faro.com/>.
- [18] *GeoM, Intelligent Geospatial Solutions*. Accessed: 2016-09-12. URL: <http://www.geomlidar.com/>.
- [19] *Geoverse*. Accessed: 2016-01-26. URL: <http://www.euclideon.com/products/geoverse/>.
- [20] Enrico Gobbetti and Fabio Marton. “Layered Point Clouds: A Simple and Efficient Multiresolution Structure for Distributing and Rendering Gigantic Point-sampled Models”. In: *Comput. Graph.* 28.6 (Dec. 2004), pp. 815–826. ISSN: 0097-8493. DOI: 10.1016/j.cag.2004.08.010. URL: <http://dx.doi.org/10.1016/j.cag.2004.08.010>.
- [21] *Google Earth*. Accessed: 2016-01-26. URL: <https://www.google.com/earth/>.
- [22] M. Hämmerle et al. “Comparison of Kinect and Terrestrial LiDAR Capturing Natural Karst Cave 3D Objects”. In: *IEEE Geoscience and Remote Sensing Letters* 11.11 (2014), pp. 1896–1900. DOI: <http://dx.doi.org/10.1109/LGRS.2014.2313599>. URL: <http://dx.doi.org/10.1594/PANGAEA.830567>.
- [23] *Harvest4D*. Accessed: 2016-01-26. URL: <https://harvest4d.org/>.
- [24] Gabor T. Herman. “Discrete Multidimensional Jordan Surfaces”. In: *CVGIP: Graph. Models Image Process.* 54.6 (Nov. 1992), pp. 507–515. ISSN: 1049-9652. DOI: 10.1016/1049-9652(92)90070-E. URL: [http://dx.doi.org/10.1016/1049-9652\(92\)90070-E](http://dx.doi.org/10.1016/1049-9652(92)90070-E).
- [25] Hobu. *Entwine and Potree examples*. Accessed: 2016-09-07. URL: <http://potree.entwine.io/>.
- [26] Hobu. *Entwine Point Cloud Indexing Engine*. Accessed: 2016-09-07. URL: <https://entwine.io/>.
- [27] *lastools*. Accessed: 2016-01-27. URL: <http://rapidlasso.com/lastools/>.

- [28] Kurt Leimer. “External Sorting Of Point Clouds”. In: (2013).
- [29] Marc Levoy and Turner Whitted. “The Use of Points as a Display Primitive”. In: (Jan. 1985). URL: <https://graphics.stanford.edu/papers/points/>.
- [30] *Lion head by Simone Garagnani*. Accessed: 2016-09-08. URL: <http://www.tcproject.net/>.
- [31] *Ludwig Boltzmann Institute for Archaeological Prespection and Virtual Archaeology (LBI ArchPro)*. Accessed: 2016-09-08. URL: <http://archpro.lbg.ac.at/>.
- [32] *Mapping the Via Appia in 3D*. Accessed: 2015-10-22. URL: <https://www.esciencecenter.nl/project/mapping-the-via-appia-in-3d>.
- [33] Oscar Martinez-Rubi et al. “Taming the beast: Free and open-source massive point cloud web visualization”. In: (2015). DOI: 10.13140/RG.2.1.1731.4326/1.
- [34] *Massive-PotreeConverter*. Accessed: 2015-10-22. URL: <https://github.com/NLeSC/Massive-PotreeConverter>.
- [35] *Mirage Technologies*. Accessed: 2015-10-22. URL: <http://www.mirage-tech.com/www/>.
- [36] Chris Murphy. *UTM zones in USA*. Accessed: 2016-08-18. URL: <https://commons.wikimedia.org/wiki/File:Utm-zones-USA.svg>.
- [37] *Netherlands eScience Center (NLeSC)*. Accessed: 2015-10-22. URL: <https://www.esciencecenter.nl/>.
- [38] Todd Ogle, Thomas Tucker, and David Hicks. *World War I Cave at German galleries from Vauquois Hill*. 2014.
- [39] *PattyVis*. Accessed: 2016-01-05. URL: <https://github.com/NLeSC/PattyVis>.
- [40] *Performance improvement by Yin Fei*. Accessed: 2016-01-24. URL: <https://github.com/potree/PotreeConverter/issues/43#issuecomment-111017203>.
- [41] Hanspeter Pfister et al. “Surfels: Surface Elements As Rendering Primitives”. In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 335–342. ISBN: 1-58113-208-5. DOI: 10.1145/344779.344936. URL: <http://dx.doi.org/10.1145/344779.344936>.
- [42] *PG&E Diablo Canyon Power Plant (DCPP): San Simeon, CA Central Coast*. Accessed: 2016-08-16. URL: <http://opentopo.sdsc.edu/lidarDataset?opentopoID=OTLAS.032013.26910.2>.
- [43] *PIX4D Photogrammetry Software*. Accessed: 2016-09-08. URL: <https://pix4d.com/>.
- [44] *Plasio*. Accessed: 2015-10-22. URL: <http://plas.io/>.
- [45] *PointCloudViz Server*. Accessed: 2015-10-22. URL: <http://server.pointcloudviz.com/>.

- [46] *PotreeViewer by SITN*. Accessed: 2015-10-31. URL: <http://sitn.ne.ch/lidar/>.
- [47] Reinhold Preiner, Stefan Jeschke, and Michael Wimmer. “Auto Splats: Dynamic Point Cloud Visualization on the GPU”. In: *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*. Ed. by H. Childs and T. Kuhlen. Eurographics Association 2012. Cagliari, May 2012, pp. 139–148. ISBN: 978-3-905674-35-4. URL: http://www.cg.tuwien.ac.at/research/publications/2012/preiner_2012_AS/.
- [48] *Prologue Systems*. Accessed: 2016-09-08. URL: <http://www.prologuesystems.com/>.
- [49] *rapidlasso*. Accessed: 2016-01-26. URL: <http://rapidlasso.com/>.
- [50] *Riegl*. Accessed: 2016-01-26. URL: <http://riegl.com/>.
- [51] Heinz Runne, Paul Banse, and Philippe Kluge. *Grave point cloud*. Accessed: 2016-09-08. URL: <http://www.hs-anhalt.de/>.
- [52] Szymon Rusinkiewicz and Marc Levoy. “QSplat: A Multiresolution Point Rendering System for Large Meshes”. In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 343–352. ISBN: 1-58113-208-5. DOI: 10.1145/344779.344940. URL: <http://dx.doi.org/10.1145/344779.344940>.
- [53] *Sagrada Familia*. Accessed: 2016-01-04. URL: https://en.wikipedia.org/wiki/Sagrada_Fam%C3%ADlia.
- [54] Claus Scheiblauer. “Interactions with Gigantic Point Clouds”. PhD thesis. Favoritenstrasse 9-11/186, A-1040 Vienna, Austria: Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2014. URL: <http://www.cg.tuwien.ac.at/research/publications/2014/scheiblauer-thesis/>.
- [55] Claus Scheiblauer and Michael Pregeßbauer. “Consolidated Visualization of Enormous 3D Scan Point Clouds with Scanopy”. In: *Proceedings of the 16th International Conference on Cultural Heritage and New Technologies*. Vienna, Austria, Nov. 2011, pp. 242–247. ISBN: 978-3-200-02740-4. URL: <https://www.cg.tuwien.ac.at/research/publications/2011/scheiblauer-2011-chnt/>.
- [56] Claus Scheiblauer and Michael Wimmer. “Out-of-Core Selection and Editing of Huge Point Clouds”. In: *Computers & Graphics* 35.2 (Apr. 2011), pp. 342–351. ISSN: 0097-8493. URL: <https://www.cg.tuwien.ac.at/research/publications/2011/scheiblauer-2011-cag/>.

- [57] Markus Schuetz and Michael Wimmer. “High-Quality Point Based Rendering Using Fast Single Pass Interpolation”. In: *Proceedings of Digital Heritage 2015 Short Papers*. Granada, Spain, Sept. 2015, pp. –. URL: <https://www.cg.tuwien.ac.at/research/publications/2015/SCHUETZ-2015-HQP/>.
- [58] *senseFly*. Accessed: 2016-09-08. URL: <https://www.sensefly.com/user-cases/mapping-the-matterhorn.html>.
- [59] *ShareLIDAR*. Accessed: 2015-10-22. URL: <http://www.sharelidar.com/>.
- [60] *Sigeom Sa*. Accessed: 2016-01-26. URL: <http://sigeom.ch/>.
- [61] *Sketchfab*. Accessed: 2016-01-26. URL: <http://sketchfab.com/>.
- [62] Jason Stoker. *The 3D Elevation Program: Overview*. Accessed: 2016-08-30. URL: http://dels.nas.edu/resources/static-assets/besr/miscellaneous/MS/2015/3DEP_Stoker.pdf.
- [63] Larry J. Sugarbaker et al. “The 3D Elevation Program InitiativeA Call for Action”. In: (2014). Accessed: 2016-08-30. DOI: 10.3133/cir1399. URL: <http://dx.doi.org/10.3133/cir1399>.
- [64] *surface and edge*. Accessed: 2015-11-5. URL: <http://www.surfaceandedge.com/>.
- [65] *three.js*. Accessed: 2016-08-18. URL: <http://threejs.org/>.
- [66] *udWeb Demo*. Accessed: 2015-10-22. URL: http://udserver.euclideon.com/demo/html5_viewer.html.
- [67] *Veesus*. Accessed: 2015-10-22. URL: <http://www.veesus.com/veesus/index.php>.
- [68] *Voxel Quest*. Accessed: 2016-09-14. URL: <http://www.voxelquest.com/>.
- [69] W. Wagner et al. “Radiometric Calibration of Full-Waveform Small-Footprint Airborne Laser Scanners”. In: *ISPRS Archives XXXVII* (July 2008), pp. 163–168.
- [70] Michael Wand et al. “Interactive Editing of Large Point Clouds”. In: *Eurographics Symposium on Point-Based Graphics*. Ed. by M. Botsch et al. The Eurographics Association, 2007. ISBN: 978-3-905673-51-7. DOI: 10.2312/SPBG/SPBG07/037-045.
- [71] *WebGL Extensions*. Accessed: 2016-08-18. URL: <https://www.khronos.org/registry/webgl/extensions/>.
- [72] *WebGL Specification*. Accessed: 2016-08-18. URL: <https://www.khronos.org/registry/webgl/specs/1.0/>.
- [73] *Weiss AG*. Accessed: 2015-11-5. URL: <http://www.weiss-ag.org/>.
- [74] *Wikipedia - Transverse Mercator Projection*. Accessed: 2016-01-26. URL: https://en.wikipedia.org/wiki/Transverse_Mercator_projection.

- [75] Michael Wimmer and Claus Scheiblauer. “Instant Points”. In: *Proceedings Symposium on Point-Based Graphics 2006*. Eurographics. Boston, USA: Eurographics Association, July 2006, pp. 129–136. ISBN: 3-90567-332-0. URL: <http://www.cg.tuwien.ac.at/research/publications/2006/WIMMER-2006-IP/>.
- [76] Matthias Zwicker et al. “Surface Splatting”. In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '01. New York, NY, USA: ACM, 2001, pp. 371–378. ISBN: 1-58113-374-X. DOI: 10.1145/383259.383300. URL: <http://doi.acm.org/10.1145/383259.383300>.