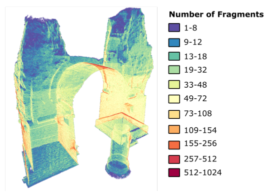
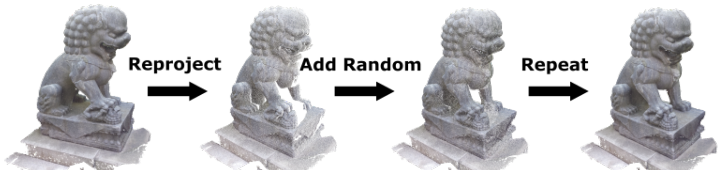
Progressive Real-Time Rendering of Unprocessed Point Clouds

Markus Schuetz
TU Wien
mschuetz@cg.tuwien.ac.at

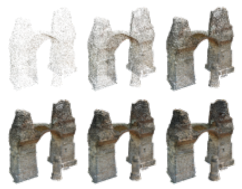
Michael Wimmer
TU Wien
wimmer@cg.tuwien.ac.at



(a) Fragments per Pixel



(b) Pipeline



(c) Uniform progression

Figure 1

ABSTRACT

Rendering tens of millions of points in real time usually requires either high-end graphics cards, or the use of spatial acceleration structures. We introduce a method to progressively display as many points as the GPU memory can hold in real time by reprojecting what was visible and randomly adding additional points to uniformly converge towards the full result within a few frames.

Our method heavily limits the number of points that have to be rendered each frame and it converges quickly and in a visually pleasing way, which makes it suitable even for notebooks with low-end GPUs. The data structure consists of a randomly shuffled array of points that is incrementally generated on-the-fly while points are being loaded.

Due to this, it can be used to directly view point clouds in common sequential formats such as LAS or LAZ while they are being loaded and without the need to generate spatial acceleration structures in advance, as long as the data fits into GPU memory.

CCS CONCEPTS

• **Computing methodologies → Rasterization**;

KEYWORDS

point based rendering, point cloud, LIDAR

ACM Reference Format:
Markus Schuetz and Michael Wimmer. 2018. Progressive Real-Time Rendering of Unprocessed Point Clouds. In *Proceedings of SIGGRAPH '18 Posters.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3230744.3230816

1 INTRODUCTION

Point clouds are commonly obtained by scanning the real world through various devices and methods, such as laser scanners and photogrammetry.

One of the difficulties of point-based models is that a significantly larger number of vertices is required to achieve a similar level of detail as polygon models. Textures can be used to represent surface features in between the vertices of a polygon, but with point clouds, all the features are represented with colored points. As a consequence, even small point-cloud models tend to consist of tens of millions to hundreds of millions of points.

Another issue of point-based rendering is the overdraw due to the large amount of overlapping fragments, as depicted in Figure 1a. When a user moves away from a textured triangle, the number of fragments it produces is reduced. However, if a user moves away from a point cloud, the number of fragments remains the same but with many of them projected onto the same pixel.

2 RELATED WORK

Futterlieb et al. developed a method that accumulates detail when the camera is still and creates a new vertex buffer from visible points in discrete intervalls, in order to preserve the accumulated details when the camera moves again [Futterlieb et al. 2016]. Our method differs in that we create an index buffer every frame, instead of a vertex buffer in discrete intervalls.

Similar to our approach, Ponto et al. reprojects every frame to the next, but they add nodes of a hierarchical structure, instead [Ponto et al. 2017]. As such, it converges faster but in non-uniform way, and it requires a hierarchical structure.

3 METHOD

The basic idea of our method is to reduce the amount of points that are drawn each frame by only rendering points that were visible in the previous frame, plus a random set of additional points to fill gaps that appear after transformations. This is done in three passes:

(1) Reproject previous frame.
(2) Add random points.
(3) Compute Dynamic Index Buffer (IBO) from visible points.

Reproject Previous Frame: The previous frame is reprojected by rendering the vertex buffer object (VBO) with an index buffer that contains only the indices of points that were visible in the previous frame. This index buffer is generated directly on the GPU through a screen-space compute shader in the third pass of the previous frame. We use glDrawElementsIndirect to draw the GPU-generated index buffer and index buffer arguments without the need to send them to the CPU first.

Add random points: During transformations, gaps will appear because previously occluded areas become visible and points that occupied adjacent pixels in the previous frame may not be adjacent in the current frame anymore. These gaps are filled by adding a different set of random points each frame.

Random points are selected by looping through and rendering subranges of a shuffled VBO. In the first frame, points in range [0, numRandom) are rendered, then in the second frame, points within range [numRandom, 2 * numRandom), and so on. After the end of the VBO is reached, this process starts from the beginning.

Without camera movement, this method converges to the same result as if all points were rendered at once. It takes *numPoints / numRandom* frames to converge. For example, if 1 million points are added each frame and the point cloud contains 26 million points, then it takes 26 frames until this method converges.

We decided to add random points because it results in a uniform progression to the final result. Adding points that are sorted in some way has shown to be faster in our benchmarks, likely because of improved cache behaviour of texture writes, but it results in very noticeable and unpleasant flickering.

Compute Dynamic IBO from visible points: This last pass is a preperation for the next frame. A compute shader dynamically creates an IBO of all visible points by running over each pixel in the point-index color attachment and storing the indices in a buffer. It also generates the arguments for the next draw call of the Reproject-Pass, mainly the numer of indices that are stored in the dynamic IBO, directly on the GPU so that a roundtrip to, and syncing with, the CPU is avoided.

3.1 Data Stucture

The data structure for our method is a single randomly shuffled VBO, which allows us to add random points to a frame by rendering a range of vertices from the VBO.

Since one of the goals of this method is that it can be used while a file is being loaded or while a scan is still in progress, this VBO has to be kept in random order over all the points it contains, even when a new batch of points is added. Shuffling just the newly added batch of points is not sufficient, but shuffling all previously added points along with it would significantly decrease performance as the total number of points increases. To avoid this, we incrementally build a randomly shuffled VBO by adding new points to random locations inside the VBO using a compute shader. If there is a collision with a previously added point, we move the existing point towards the end of the VBO before inserting the new point to its target location.

Table 1: Timings for Heidentor (26M points), Retz (120M points) and Candi Sari (250M points), in milliseconds. To exploit faster GPUs, 1M, 2M and 3M points were added each frame on the 940 MX, 1060 GTX and 1080 GTX, respectively. Even though Candi Sari contains more points, it renders faster because views of the temple tend to cover fewer pixels.

Model	GPU	Reproject	Add	Build IBO	Total
Heidentor	1080 GTX	0.312	1.804	0.038	2.154
	1060 GTX	0.415	2.913	0.086	3.414
	940 MX	2.293	8.649	0.339	11.281
Retz	1080 GTX	1.697	1.126	0.069	2.892
	1060 GTX	1.523	4.002	0.117	5.642
Candi Sari	1080 GTX	0.976	1.716	0.052	2.744

4 PERFORMANCE

The maximum number of points that are rendered each frame is limited to *numPixels + numRandom*, with 0.5 to 10 million as suitable values for *numRandom*, depending on the performance of the GPU.

Table 1 shows benchmark results on different GPUs. We were able to push up to 120M points to the 1060 GTX, and 250M points to the 1080 GTX before the GPU started to render from main memory instead of GPU memory.

It takes *numPoints / numRandom* frames until the result converges. At a framerate of 300fps for Candi Sari on a 1080 GTX, it takes about a third of a second until it converges, with *numRandom* set to 3M.

5 CONCLUSIONS AND FUTURE WORK

We have shown a method that can be used to progressively render any point cloud that fits on GPU memory in real time using a shuffled array as data structure, and how to incrementally build this shuffled array while the points are being loaded.

In the future, we would like to explore ways to handle point clouds that are larger than GPU memory, for example by asynchronously streaming points from CPU memory, which is usually larger than GPU memory.

ACKNOWLEDGMENTS

The authors would like to thank the Ludwig Boltzmann Institute for Archaeological Prospection and Virtual Archaeology for providing the Heidentor data set, *RIEGL* Laser Measurement Systems for providing the data set of the town of Retz, and the Institute of History of Art, Building Archaeology and Restoration at TU Wien for providing the Candi Sari data set.

REFERENCES

Jörg Futterlieb, Christian Teutsch, and Dirk Berndt. 2016. Smooth visualization of large point clouds. *IADIS International Journal on Computer Science and Information Systems* 11, 2 (2016), 146–158. http://publica.fraunhofer.de/dokumente/N-453338.html

Kevin Ponto, Ross Tredinnick, and Gail Casper. 2017. Simulating the experience of home environments. In *2017 International Conference on Virtual Rehabilitation (ICVR)*. 1–9. https://doi.org/10.1109/ICVR.2017.8007521