Technical Report / Preprint: *TR 193-2-2019-4*.
In submission to a peer reviewed venue.

Progressive Real-Time Rendering of One Billion Points Without Hierarchical Acceleration Structures

Markus Schütz[1], Gottfried Mandlburger[2], Johannes Otepka[2], Michael Wimmer[1]

[1]TU Wien, Institute of Visual Computing & Human-Centered Technology
[2]TU Wien, Department of Geodesy and Geoinformation

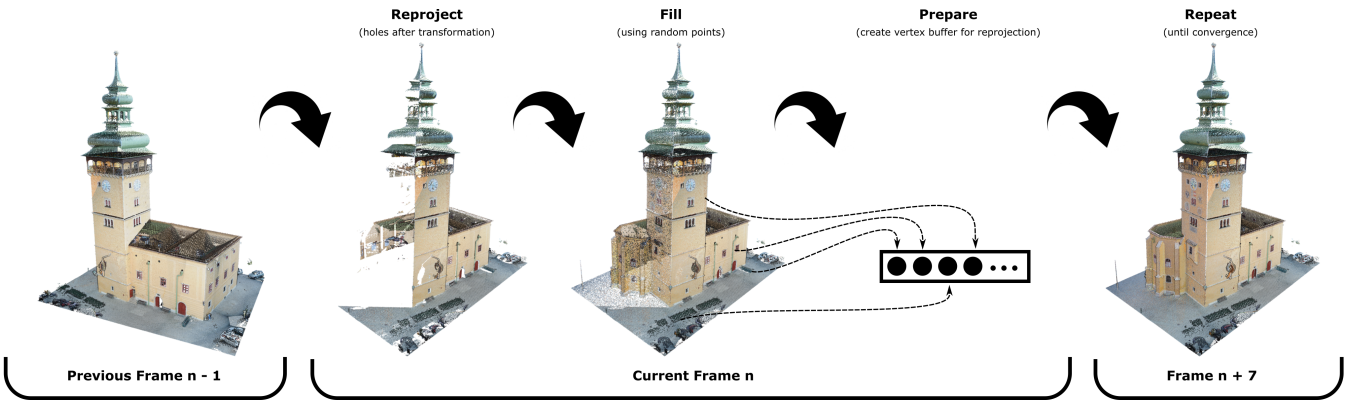


Figure 1: The progressive rendering of point clouds via reprojection and filling allows us to maintain real-time frame rates by distributing the rendering of large point clouds over multiple frames, without the need to generate acceleration structures in advance. Filling holes with randomized subsets of the full data set leads to higher quality convergence patterns, and the duration to convergence can be adjusted by the amount of random points that are rendered in the fill pass. *Retz* point cloud courtesy of *Riegl*.

Abstract

Research in rendering large point clouds traditionally focused on the generation and use of hierarchical acceleration structures that allow systems to load and render the smallest fraction of the data with the largest impact on the output. The generation of these structures is slow and time consuming, however, and therefore ill-suited for tasks such as quickly looking at scan data stored in widely used unstructured file formats, or to immediately display the results of point-cloud processing tasks.

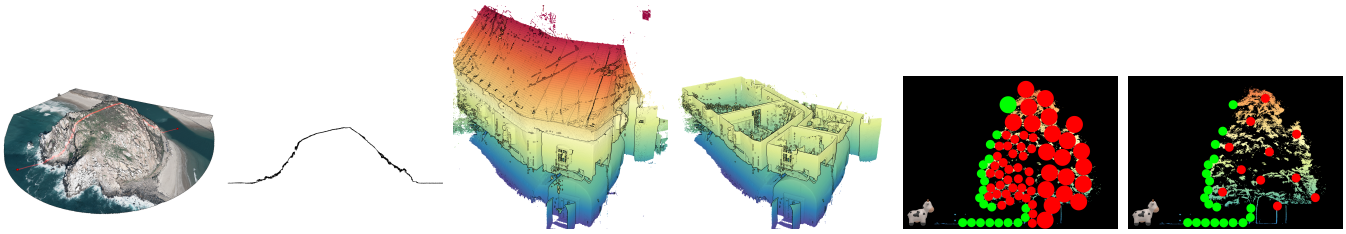
We propose a progressive method that is capable of rendering any point cloud that fits in GPU memory in real time, without the need to generate hierarchical acceleration structures in advance. Our method supports data sets with a large amount of attributes per point, achieves a load performance of up to 100 million points per second, displays already loaded data in real time while remaining data is still being loaded, and is capable of rendering up to one billion points using an on-the-fly generated shuffled vertex buffer as its data structure, instead of slow-to-generate hierarchical structures. Shuffling is done during loading in order to allow efficiently filling holes with random subsets, which leads to a higher quality convergence behavior.

CCS Concepts

• ***Computing methodologies*** → ***Rendering; Rasterization;***

1. Introduction & Problem Statement

Point clouds, i.e., 3D-models that consist of potentially colored points, are usually obtained by scanning the real world with various types of 3D-scanners or by image-based reconstruction methods [Wei16]. Unlike mesh-based models, which can represent additional detail between vertices cost-effectively with textures, basic point-cloud models represent all surface details with individual points. Points have no connectivity, which can easily lead to noticeable gaps in-between, unless they are filled with even more points or covered up by larger points. As a consequence, even seemingly small scenes are made up of millions of points, and larger



(a) Low Complexity (b) High Complexity (c) Hierarchical (d) Progressive

Figure 2: Depth-complexity denotes the amount of occluded surface layers in a given viewpoint. A high complexity results in wasteful rendering of invisible data. (a) Low-complexity models feature few occlusions – e.g. individual objects, buildings without interiors and terrain without vegetation. (b) A highly complex scan of a building with interiors. (c, d) Rendering a detailed scan of a tree. Green: Rendered points that are visible from viewpoint. Red: Rendered points that are occluded from green points. (c) Hierarchical methods without progressive approach waste their point budget on rendering occluded points. (d) Progressive rendering performance is largely unaffected by depth complexity. Occluded points are rendered, but only varying random subsets that can be rendered in real time, while previously rendered data is preserved through reprojection.

models can consist of hundreds of billions of points. Due to the lack of a standard hierarchical point-cloud file format with support for levels of detail (LOD), point clouds are distributed in sequential formats such as LAS [ASP19] and its compressed counterpart, LAZ [Ise13]. Point-cloud processing and rendering applications may build their own hierarchical structures, but this takes time, and support for a particular hierarchical format is usually limited to the application that created it.

The data structures used to render large point clouds often differ from data structures that are used to process large point clouds. Rendering requires data structures that provide quick access to varying levels of detail of the model, based on the position and direction of the viewer. Processing, on the other hand, usually requires data structures that provide quick access to all points in a certain region without considering levels of detail. The point-cloud processing framework OPALS [OP; PMOK14], for example, uses a kd-tree where all points are stored in the leaf-nodes. This provides no access to level-of-detail data, but efficient access to data within a chosen region. While OPALS can quickly modify, filter and augment all the data in a region, it cannot quickly display the results. With state-of-the-art methods, rendering the results would require lengthy preprocessing steps to generate a hierarchical structure whenever new results are generated. Several issues complicate the use of hierarchical structures for point clouds:

Number of Point Attributes. Most point clouds contain at least an XYZ coordinate and either a color value or a scalar value with various meanings. This basic format consumes at least 16 bytes per point. However, some use cases require a large amount of additional per-point attributes. Possible attributes include intensity, reflectance, classification, return number, scan angle, GPS-time, echo ratio, beam direction, surface normals, etc., which can increase the storage requirements to more than 100 bytes per point. Storing all these attributes negatively affects load-, processing- and rendering times, even if only a small amount of attributes is actually needed. Section 3.5 describes how we deal with attributes in our progressive rendering method.

Depth Complexity. One of the main problems of hierarchical structures for point clouds is that they do not address models with high depth complexity – Treddinick et al. [TBP16] being a notable exception. A high depth complexity means that the model consists of multiple layers of surfaces that occlude each other, as shown in Figure 2. Occluded points use up a large portion of the available point budget without contributing to the image. As a consequence, the model will either be rendered at a lower level of detail because there is no more budget left for higher levels, or the budget is increased, which reduces performance. Examples for models with high complexity include scans of buildings with their interiors, dense vegetation, and data with a high amount of noise. In case of buildings with interiors, most rooms are occluded by walls, floors and ceilings. All these occluded rooms still need to be rendered because they may be partially visible through gaps between points or holes in the scan. Adapting point sizes to close gaps is problematic because of irregular point densities, especially along and between scan lines of laser scanners, and because a lot of gaps are a result of missing data that the scanner did not obtain.

LOD Build-Up Times. Generating LOD structures requires time, and the lack of a standard format means that each application that supports LOD structures uses its own format. Two of the most widely used formats for distribution, LAS and LAZ, are non-hierarchical. Building hierarchical structures happens at rates of around 50 thousand to 1 million points per second (see [Sch14; WBB*07] and Table 2), which means it takes 300 to 6 000 seconds before we can explore a point cloud with 300 million points. With our method, we are able to fully load and render 300 million points in 3 to 10 seconds, depending on the file format. 3 seconds refer to an ideal format that matches the vertex buffer, and 10 seconds are required for the the widely used LAS format.

Our main contributions are as follows:

- We introduce a progressive method that renders point clouds that fit in memory in real time without hierarchical structures, tested with up to one billion points.
- In each frame, our progressive method fills holes by rendering

a random subset of the point cloud, which results in a relatively uniform and pleasant convergence to the full image.
- We show how to create these random points incrementally and in parallel on the GPU using a prime number based pseudo-random number generator that generates unique integer values in a given range.
- Our method allows real-time rendering of already loaded data, while remaining data is still being loaded from disk.
- It achieves disk to GPU transfer rates of up to 37M points/s or 1GB/s for the widely used LAS point cloud file format, and up to 100M points/s or 1.6GB/s for a simple binary file that matches the GPU vertex buffer format.
- It is tailored to and supports point clouds with large amounts of attributes – tested with up to 50 attributes and 107 bytes per point.

2. Related Work

Previous work related to our method includes progressive – or incremental – rendering algorithms in various domains, especially those using reprojection, but also hierarchical rendering algorithms for large point clouds. While we do not use hierarchical structures, or any other spatial acceleration structure, we consider these to be related because they are, to the best of our knowledge, the only other option to render large point clouds at rates higher than 60 frames per second.

Temporal coherence denotes the similarity of a scene or rendered image over time. Badt [Bad88] already suggested to take advantage of temporal coherence in ray tracing by reprojecting the pixels in the previous frame to the current one. This saves a considerable amount of work because most of the surfaces that are visible in the current frame were already visible in the previous frame. Walter et al. [WDP99] proposed a point-based structure for reprojection, the Render Cache, which is used to reproject previously rendered data to the new frame, and to keep track of possibly outdated regions of the image that need to be updated. The goal of the Render Cache is to maintain interactive frame rates during motion or when editing a scene in ray or path tracers, but to make sure the frame eventually converges if there is no further change to the scene or camera. Jevans [Jev92] exploits temporal coherence in object space by tracking objects that move so that only animated parts of the scene need to be retraced. In a state-of-the-art report on temporal-coherence methods from 2011, Scherzer et al. [SYM*11] discuss a wide range of algorithms that exploit coherence, with a special focus on reprojection algorithms

Most of the work on rendering large point clouds focuses on creating and rendering hierarchical level-of-detail structures. These structures have the additional advantage that they can also be used in an out-of-core fashion, but they require time to generate in advance. QSplat [RL00] was the first one to use a point-based hierarchical structure to render large meshes. It uses a hierarchy of bounding-spheres that is traversed during rendering. Sequential Point Trees [DVS03] and Layered Point Clouds [GM04] improve this concept and offer GPU-friendly structures. The former is similar to QSplats but sequentialized into a single sorted array, and the latter groups points into tree nodes of varying resolution and size. This grouping of points into a multi-resolution tree structure, where each node stores a subset or a representative model of the original model, is the key breakthrough that allowed for efficient rendering of arbitrarily large point clouds on the GPU. Further work then explores various ways to generate, modify and render hierarchical structures similar to the layered point clouds [WS06; Sch14; APS*14; WBB*07; GZPG10; Sch16; PTC17].

Tredinnick et al. [TBP16] and Ponto et al. [PTC17] proposed a progressive rendering method for point clouds that is related to ours. The previous frame is reprojected to the current one and holes are filled by rendering additional points. Their work focuses on hierarchical methods, however, and in each frame they render a different set of octree nodes within the view frustum. Even though only part of the data is rendered in each frame, the image converges to the full amount of detail after a few frames. Our method differs in that we focus on progressively rendering unstructured data for which no hierarchical structure was generated in advance, which allows us to look at unstructured data up to two orders of magnitudes faster than methods that require hierarchical structures. Another similar technique by Futterlieb et al. [FTB16] renders cached results during movement and accumulates details when the camera doesn't move.

3. Progressive Rendering

In the context of our paper, progressive rendering means that we distribute the task of rendering the full point cloud over multiple frames, instead of doing all the work in a single frame. The goal is to maintain real-time frame rates and keep the application responsive at all times. The basic idea to achieve this goal is to reproject the previous frame, since most of the previously visible points are likely to be visible again in the current frame, and then fill holes that appear due to disocclusions with randomly selected additional points to obtain a high-quality convergence behavior. Over the course of multiple frames, the result converges to an image of the full model.

In this section, we will describe the necessary data structures, how we load points, an efficient way to incrementally shuffle points during loading, the actual rendering pipeline, how we adaptively select the point budget to minimize the duration to convergence while maintaining real-time frame rates, and how to switch between different point attributes.

3.1. Data Structure

Our method employs two data structures in order to stream new attribute data to the GPU, and to quickly render a certain amount of random points in each frame.

On the CPU side, point attributes are stored in a struct-of-arrays fashion, i.e., one array stores exactly one attribute: $[RRR][GGG][BBB]$. This allows us to stream specific attributes from CPU to GPU with minimal usage of memory bandwidth, since accessing a value of an attribute array will load a whole cache line of subsequent values into the CPU cache [Dre07]. An interleaved array, on the other hand, would result in loading various different attributes of a point into the CPU cache, which is not useful if only one attribute of a point is needed.

On the GPU side, we use a shuffled interleaved vertex buffer with 16 bytes per point as our rendering data structure, which is created by (inserting points at pseudo-random locations. Due to the maximum buffer size of 2^{31} bytes on modern GPUs, the shuffled Vertex Buffer Object (VBO) may actually consist of multiple buffers – one for every $2^{31}/16 \approx 134$ million points. Shuffling is done because it reduces the problem of rendering a batch of N random points to rendering N consecutive points. Each point contains 12 bytes for XYZ coordinates, and another 4 bytes for attribute data. The attribute data may contain a single 4 byte float, or four unsigned bytes. The former is used to visualize single scalar attribute values, and the latter is used to visualize vectors of attributes, such as colors and normals. It is up to the vertex shader to interpret the data as needed. Newly loaded batches of points or new batches of attributes are not directly uploaded to the shuffled VBO. Instead, they are uploaded to a separate *Distribute* buffer that holds a single batch of 500k points. A compute shader then inserts the points or attributes to the respective shuffled location in the VBO. The *Distribute* buffer receives 16 byte XYZRGBA during the initial loading from disk, but only 4 bytes per point, i.e., just the attribute data, when switching to a new attribute. Finally, the *Reproject* buffer contains all the points that are visible at the end of a frame. In addition to position and attribute data, it also stores the index of that point inside the shuffled VBO, which is needed during reprojection to write the point indices along with point colors to the framebuffer.

3.2. Loading

One of the objectives of our method is that intermediate results are shown in real time while remaining data is being loaded. In order to achieve this, files are loaded and transformed to GPU-ready buffers in parallel, and the task of the main thread is simplified to sending batches that are ready to the GPU. Figure 3 illustrates this process in a time line. The load thread is dedicated to reading binary data in batches of 500k points from disk. Three additional parser threads transform the binary batches and separate the interleaved point data into one array per attribute, which are then appended to the struct-of-arrays structure in main memory. During the start of the next frame, the main thread sends the XYZRGBA attribute of all batches that were fully loaded and parsed in the previous frame to the GPU. The composite XYZRGBA array is a special case that gets assembled by the parser threads after all other attributes are stored in separate arrays, because this is the initial data that we send to the GPU.

3.3. Incremental Parallel Shuffling

Rendering randomly selected points improves the perceived visual quality during convergence to the final image, compared to rendering points in their original and potentially sorted order. Points are shuffled during loading so that we can efficiently render N random points by rendering a subset of N consecutive points from the vertex buffer. Since we want to display the points with our progressive method while additional points are still being loaded from disk, we need to use a shuffling method that is capable of incrementally shuffling points as they become available. The Fisher-Yates shuffle [FY38; Dur64] is a well-known shuffle algorithm that is unbiased (each permutation is equally likely), has an optimal complexity of

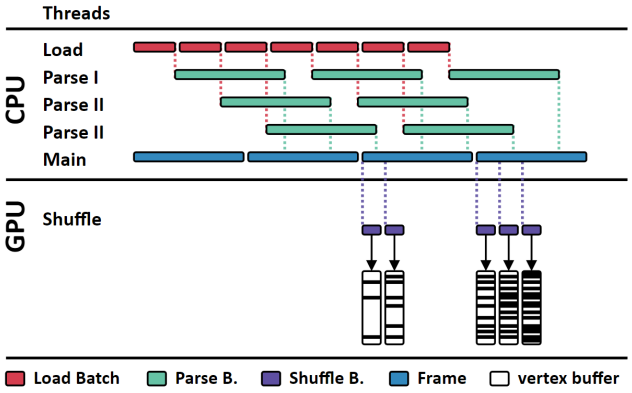


Figure 3: The load pipeline. One thread is dedicated to reading batches of binary data from disk. Three threads are used to transform the loaded binary batches into the structure-of-arrays memory layout. At the start of each frame, the main thread uploads fully parsed batches to the GPU and executes a compute shader that moves each point to its shuffled location in the vertex buffer.

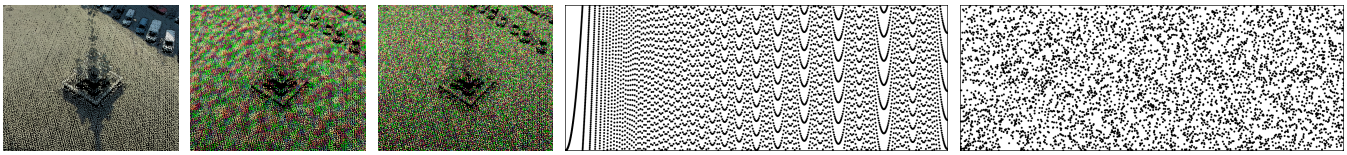
O(n), and can operate incrementally. However, it is not parallelizable because each step of the loop depends on the result of the previous step, and it cannot keep up with the speed at which points are read with sequential file I/O from an SSD.

Random number generators are an essential component of many shuffle algorithms, including the aforementioned Fisher-Yates shuffle, and prime numbers are the basis of some generators with useful properties. Blum Blum Shub [BBS82], for example, is a random-number generator in the domain of cryptology that uses the input $M = P \cdot Q$, where P and Q are primes that are congruent to 3 (mod 4). M is then used to generate a sequence of pseudo-random bits. Similarly, a single prime P that is congruent to 3 (mod 4) can be used to generate a sequence of unique integers in the interval $[0, P)$ [Pre12]. This type of pseudo-random number generator essentially provides a permutation of input to target indices that we use to move points from their original position to their position inside a shuffled array. The target indices are computed as:

$$permute(i) = \begin{cases} i^2 \bmod P, & \text{if } i \leq \frac{P}{2} \\ P - i^2 \bmod P, & \text{if } i < P \\ i, & \text{otherwise} \end{cases} \quad (1)$$

$$targetIndex(i) = permute(permute(i)) \quad (2)$$

Equation 1 maps each number in $[0, 1, 2, ..., P-1]$ to another number in the same range without producing duplicates. Due to the condition that P must be a prime and $P \equiv 3 \,(\mathrm{mod}\, 4)$, we can only shuffle points with indices in interval $[0, P)$. For any given number of points N, we compute the next smaller prime $P \leq N$ that satisfies the condition, permute all the points from index 0 to P, and leave the remaining points at their original positions. The number of unshuffled points is negligible because the gap between consecutive



(a) (b) (c) (d) $f(i) = permute(i)$ (e) $f(i) = permute(permute(i))$

Figure 4: (a) Points colored by RGB. (b+c) Points colored by indices. (b+d) One pass of Equation 1 produces noticeable patterns in the mapping of input indices to target indices – some patches of points preserve locality after the shuffle. (c+e) Applying Equation 1 twice results in a sufficiently random permutation of input to target indices.

primes is small. The largest gap between two consecutive primes $P \equiv 3 \,(\mathrm{mod}\, 4)$ for up to 500 million points is 532, between primes 184 007 671 and 184 008 203. This means that for up to 500 million points, at most 532 may not be shuffled. Trying to shuffle them as well results in extra work with insignificant improvement. Alternatively, one could find the next larger prime, shuffle the entire data set, and leave a negligible amount of vertex buffer elements empty. The big advantage of the prime number-based method over other methods like the Fisher-Yates shuffle is that it can be applied to each input index i individually, without depending on the state from previous calculations and with no collisions. It is therefore inherently parallelizable and can be implemented in a compute shader on the GPU without synchronization between threads.

A disadvantage of the prime number-based method is the relatively low quality of the permutation after only one pass, which manifests as noticeable patterns, as shown in Figure 4. Since Equation 1 is bijective – each element of the input set $[0, 1, 2, ..., P-1]$ maps to exactly one distinct element of the same set – we can simply apply it multiple times and still obtain the same number of unique target indices. Applying it a second time results in a randomness that is not necessarily of high quality, but sufficiently random for our progressive rendering method. With "not high-quality" we mean that there are certain patterns, and some random numbers may be predictable from previous random numbers. For example, Equation 1 is monotonically increasing for the first $\sqrt[2]{P}$ numbers and Equation 2 is monotonically increasing for the first $\sqrt[4]{P}$ numbers. The former is immediately obvious in Figure 4 (d), and the same patterns are visible repeatedly throughout the function graph. The latter is not noticeable in Figure 4 (e). As long as patterns aren't immediately obvious visually, we consider a random number generator sufficiently random for our method.

3.4. Rendering Pipeline

The progressive rendering method reprojects the previous frame to the current frame, and then fills in missing data by rendering a certain number of random points. Over the course of multiple frames, the result will converge to the same image that we would get by rendering all points at once, not accounting for render order and z-fighting issues. This method is realized in three render passes:

1. **Reproject**: Render all the points that were visible in the previous frame, reprojected to the current frame.
2. **Fill**: Render a batch of random points to fill holes. This is done efficiently by rendering subsets of a shuffled vertex buffer.
3. **Prepare**: Create a new vertex buffer from all points that are visible in the rendered image. This vertex buffer will be used in pass one of the next frame.

Figure 6 shows the ratio of time spent on passes of a standard brute-force approach, a basic fixed fill-budget approach, and an adaptive fill-budget approach.

The idea of reprojection is that most of the data that was visible in the previous frame will also be visible in the current frame, so it may make sense to reuse it. However, during movements, previously occluded parts of the surface and parts that were outside of the frustum may become visible, but since this data is missing from the previous frame, holes and empty regions will appear, as shown in Figure 1. The *Fill* pass attempts to fill missing areas by adding random points. We chose to fill using randomly selected points because it results in a relatively uniform convergence to the final result over the whole image with no apparent pattern, and because it looks relatively pleasant compared to filling with sorted chunks of points. If points are in some way sorted or structured, it will result in unpleasant flickering artifacts during motion because in each frame, parts of the image will fully converge, while other parts will see no progress at all until later frames.

Depending on hardware capabilities, we render between $S = [1M, 30M]$ random points each frame during the *Fill* pass. The selection of random points is achieved by rendering subsets of the shuffled vertex buffer. In the first frame, points from $[0, S)$ are rendered, then in the next frame $[S, 2 \cdot S)$ and so on. Once we have looped through all the points in the vertex buffer, we repeat again from the beginning. Without camera motion, the image converges to the full result after looping through all points once. During motion, the whole buffer must be repeatedly looped through in order to keep filling new holes.

The third and final pass – *Prepare* – creates the *Reproject* vertex buffer out of all currently visible points, which is then used in the *Reproject* pass of the next frame. Note that instead of reprojecting the points directly from the framebuffer, which would lead to inaccuracies, we identify the original point projected to a pixel and reproject it from its original coordinates. This requires that the *Reproject* and *Fill* passes both write point indices into an additional index-color attachment on the framebuffer. A compute shader iterates over all pixels, reads the point indices from the index-color attachment, and copies the respective points from the shuffled vertex buffer into the *Reproject* vertex buffer. In addition to XYZ and the 4-byte attribute value, all points in the *Reproject* buffer also store the shuffle point index, which is needed by the *Reproject* pass

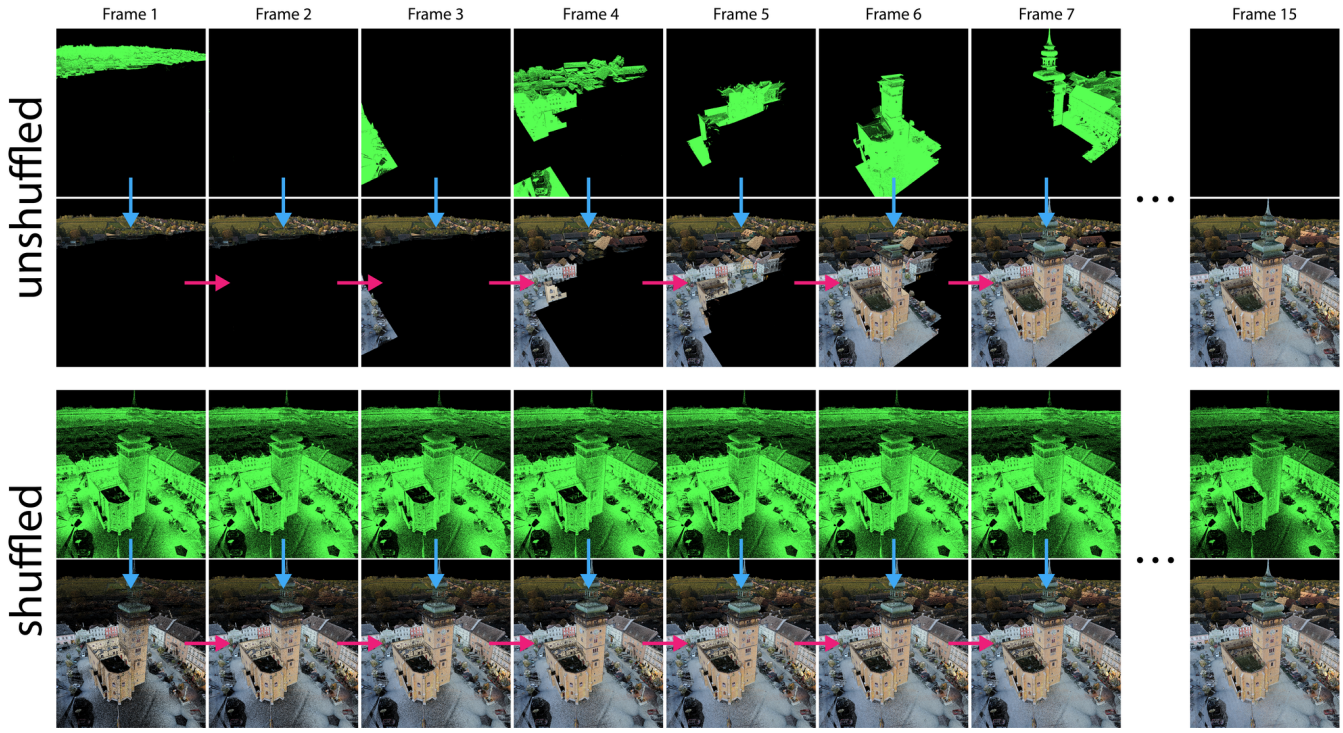


Figure 5: Convergence behavior of unshuffled and shuffled point clouds. First and third rows: Varying subsets of 10 million points that the *Fill* pass will render in that frame. Second and fourth rows: The image that is displayed to the user after reprojecting the previous frame to the current one, and filling missing data with the selected subsets. The unshuffled version renders localized batches, and sometimes no data at all if the subset is completely outside the view frustum. The shuffled version quickly covers the entire screen. Both versions converge to the same image after 15 frames.

to write the correct index of a point inside the shuffled vertex buffer into the index-color attachment.

3.4.1. Adaptive Fill Budget

A basic implementation of the *Fill* pass renders a fixed amount of random points to fill gaps, e.g., 1 million points. To improve convergence times, we want to render as many points as possible in each frame while maintaining real-time frame rates. Since render times vary greatly depending on viewpoints, we cannot reliably use past frames to estimate the fill budget for the current frame. Instead, we measure render timings of the current frame directly on the GPU and then estimate the number of additional points we can render. If 60fps are desired, the frame must be fully rendered within $\frac{1}{60} = 16.6$ms. To estimate an adaptive fill budget, we measure the time since the beginning of the frame, and the time it took to render the first 1 million points. From this, we compute the number of rendered points per millisecond, which we then multiply by the milliseconds we have left to finish the frame. Due to the margin of error of this estimate and time consumed by additional render passes and GPU tasks, we suggest to assume the available time to be well below 16.6ms, e.g., around 10ms.

An advantage of the adaptive fill budget is that it implicitly accounts for points that are outside the view frustum. While rendering the first 1 million points, points that are outside the view frustum take less time to render and the adaptive budget will be correspondingly larger. For the subsequently rendered remaining points, due to the randomization, roughly the same percentage will be outside the view frustum. The first step essentially provides a representative percentage of the amount of points that will be outside the view frustum during the second step. Due to this, the adaptive budget can vary from 20 million points per frame in viewpoints where all points are within the view frustum, up to 100 million points for close-up viewpoints within the point cloud, when a large portion of points is outside the view frustum (Measured on an RTX 2080 TI).

Implementations of an adaptive budget have to take care to avoid CPU-GPU sync-points. In OpenGL, we suggest to use timer queries that write timestamps directly into GPU buffers, and use compute shaders to estimate the remaining budget directly on the GPU. The compute shader then writes the estimated number of points we can render in the remaining time into another buffer that is used as an argument to an indirect draw call.

3.4.2. Convergence

Progressive point-cloud rendering methods distribute the rendering process over multiple frames until the result converges to the final image. We can calculate the number of frames until convergence, but the actual time to convergence has to be measured. We can also differentiate convergence behaviour that progresses in lo-

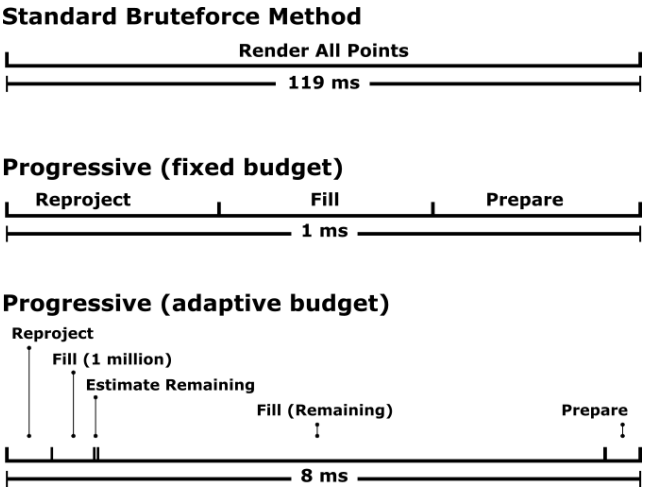


Figure 6: Proportion of time spent on the passes of three rendering approaches for 302M points. The brute-force approach significantly exceeds the limit of 16.6ms. The fixed fill budget approach with a budget of 1M points is well below the limit but has little progress per frame, and only 33% of the time is spent on the *Fill* pass that drives progress. The adaptive fill budget renders a fixed amount of points first, and then an estimated additional amount that can be rendered in the remaining time based on the time it took to render the fixed amount. It spends 90% of the rendering time on filling holes and progressing towards convergence.

calized batches, or uniformly over the whole model, as shown in Figure 5. The advantage of convergence in localized batches is that the GPU often (but not always, see Table 4) renders vertices faster if they are sorted by locality. The disadvantage, however, is that it results in severe flickering artifacts since some regions converge immediately, and others don't converge at all until later frames. Rendering randomly selected points may be slower, but it results in a uniform and pleasant convergence over the whole model.

When motion stops, the framebuffer converges after all points were rendered at least once during the *Fill* pass, i.e., after $\frac{\#points}{budget}$ frames. With a fixed fill budget of 1 million points, it will take 100 frames to finish rendering 100 million points. The actual time until convergence isn't easily predictable, since it varies between GPUs, number of points in view-frustum, number of overlapping points, etc. The rate of time spent on the *Reproject*, *Fill* and *Prepare* passes also affects time to convergence, since only the *Fill* pass keeps progressing further whereas the other two passes consume time without driving progress. If all three passes consume the same amount of time, we end up with one third of the available performance spent on progressing to convergence. However, if 0.19 ms are spent on reprojection, another 0.34 ms on preparing a new vertex buffer, and 3.34 ms on filling holes, then we are utilizing $3.34/(0.19+3.34+0.34) \approx 86\%$ of the render time towards progressing to convergence while still maintaining real-time frame rates (timings taken from Table 3).

3.5. Streaming Point Attributes

Our data sets consist of point clouds with hundreds of millions of points, and up to 50 different attributes for up to 107 bytes per point. Figure 7 shows various attributes that a point cloud can contain. Assuming 10GB of GPU memory and 107 bytes per point, we could store at most $\frac{10*1024^3}{107} = 100M$ points on the GPU. In most cases, we only need the coordinates plus one to 4 attributes at any time, so the remaining attributes consume memory unnecessarily. In addition to that, our rendering pipeline is also strongly affected by memory bandwidth because vertex buffers are recomputed each frame. The more bytes a vertex has, the slower it will be to generate a new vertex buffer. Due to this, we only keep 16 bytes per point in GPU memory, comprising of $3 \cdot 4 = 12$ bytes for the XYZ coordinates and another 4 bytes encoding one to four attributes. This allows us to store up to $\frac{10*1024^3}{16} = 671M$ points in 10GB of GPU memory, although the actual amount will be lower since other parts of the application, as well as other applications and the OS, also require some GPU memory.

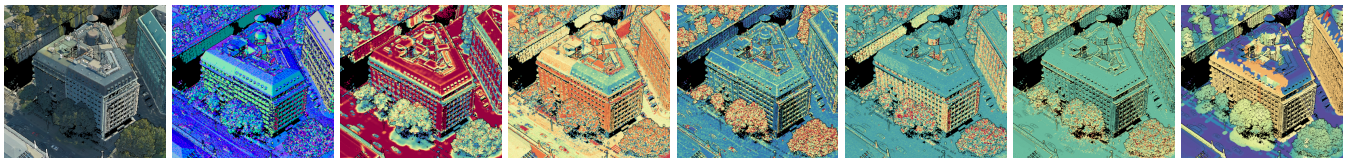
In order to be able to visualize all available attributes, we keep them in main memory and stream them to the GPU once a user asks to see another attribute. At the start of each frame, the main thread sends multiple batches of attributes to the GPU. A compute shader distributes the vertex attribute data to the respective vertices with the same pseudo-random target index computation that is used during the initial loading step, thereby overriding the previous vertex attribute data. Streaming a new attribute from CPU to GPU happens at rates of 300 to 800 million points per second on an RTX 2080 TI, depending on whether the attribute is a single scalar or a vector of up to 4 values. For the *Vienna* data set with 124 million points, switching attributes takes 0.155 to 0.356 seconds. While a new attribute is streamed, the application remains interactive, but the amount of attribute batches that are uploaded to the GPU in each frame increases frame times up to 200ms. However, implementations may place a limit on how many batches they will upload in each frame in order to maintain real-time frame rates.

4. Evaluation

In this section, we provide background information on the widely used LAS point-cloud file format that we use, followed by an introduction to our test data sets, and conclude with an evaluation of load and rendering performances.

4.1. LAS File Format

Two of the most widely used binary point-cloud file formats are LAS [ASP19] and its compressed counterpart, LAZ [Ise13]. We use the LAS format due to its compatibility with a wide range of applications, and because the simple but strict file format makes it easy to develop custom file loaders. The LAS format stores points in an interleaved fashion, i.e., each point is stored one after the other. Points consists of a set of attributes, and various predefined point data record formats describe which combination of attributes are stored. Some attributes, such as XYZ, intensity, return number and classification are present in all available formats, whether they are needed or not. Others are only present in specific formats, such as RGB in formats 2 and 3, or GPS-time in 1 and 3. Since version



(a) RGB (b) Normal (c) Echo Ratio (d) Amplitude (e) Sphericity (f) Deviation (g) Return Nr. (h) vRange

Figure 7: Various attributes of a point cloud. Each attribute increases the bytes per point, which in turn increases memory requirements. We keep only one attribute at a time in GPU memory to maximize the number of points we can store. New attributes are streamed to GPU and replace old ones on demand. *Vienna* point cloud courtesy of *Riegl*.

1.4, the spec also allows a standardized definition of custom extra attributes in addition to the fixed set of attributes. Our point-cloud application makes heavy use of these extra attributes, and some of our data sets use 50 attributes that require up to 107 bytes per point. The large amount of information for each individual point leads to challenges such as increased memory requirements and memory bandwidth usage.

4.2. Data Sets

Screenshots and descriptions of our test data sets are provided in Figure 8 and Table 1. Vienna and Morro Bay were captured with airborne laser scanners that provide a relatively uniform but low density over a large area. Retz was scanned with a combination of airborne and terrestrial laser scanning. The former provides a low-density model of the town and the surrounding area, and the latter augments it with higher detail at the center of the town.

4.3. Performance

This section lists and discusses performance results for loading unstructured data from LAS files, how to achieve 100M points per second by loading from an optimal file format, performance of the rendering pipeline, and a comparison to hierarchical structures.

The following test systems are used throughout the paper:

- **2080**: A desktop system with an AMD Ryzen 2700X CPU, an NVIDIA RTX 2080 TI with 11GB GPU memory, a 1TB Samsung 970 PRO SSD, and 32GB RAM.
- **1660**: A notebook system with an Intel i7-9750H CPU, an NVIDIA GTX 1660 TI Max-Q with 6GB GPU memory, a 256GB SSD, and 16GB RAM.
- **940**: A notebook system with an Intel Core i7-7500U CPU, an NVIDIA 940MX with 2GB GPU memory, a 1TB SATA HDD, and 16GB RAM.

In addition to that, we confirmed the claim that our method works for up to one billion points within an expected margin of performance during temporary access to a system with an NVIDIA RTX Titan with 24GB GPU memory. The data set is fully loaded in around 25 seconds, and an average adaptive fill budget of around 20 million points per frame is rendered in real time. However, specific benchmark numbers are limited to the aforementioned systems that we had unrestricted access to.

4.3.1. Loading LAS Files

In this section, we discuss the time it takes to *Load* LAS files from disk, and the time it takes to fully parse and *Upload* the data to the GPU, as shown in Table 2. Loading from disk and uploading to the GPU happen in parallel, as shown in Figure 3. The timings for the latter are therefore the total time of doing both. The *Upload* column shows that parsing and uploading finishes tenths of a second after the last batch of binary data was loaded from disk. For these benchmarks, we deactivated OS-level file caching under Microsoft Windows by calling CreateFile with the FILE_FLAG_NO_BUFFERING flag on the LAS file, before loading it with the standard C FILE API using fopen.

4.3.2. Loading 100 Million Points Per Second

The evaluated LAS load performance is limited in bandwidth for multiple reasons: First, all LAS formats store various attributes that may not actually be needed. Second, the interleaved (Array-of-Structures) layout needs to be transformed to a Structure-of-Arrays layout during loading to allow for efficient switching between attributes. And third, attributes are encoded in a way that is not directly useful for rendering, e.g., RGB values are stored in 2 bytes each, and coordinates are stored as 32-bit fixed-precision integers in order to maximize the precision of 32-bit values while avoiding the additional disk-space cost of 64-bit data types. During loading, the coordinates are transformed to a floating-point type, usually double values for accurate processing, or origin-centered floats for rendering. An ideal file format with respect to low loading times would need few bytes per point and require little to no transformation of the attribute values.

We are able to achieve load performances of up to 100 million points per second by limiting ourselves to XYZ and RGBA values, storing points in the same format on disk as we use in the GPU vertex buffers, and directly transferring buffers from disk to GPU without any modifications. Coordinates are stored as single-precision floating-point values and colors as unsigned bytes. Each point requires 16 bytes. The resulting transfer rate from disk to GPU is 1.6GB/s. These numbers also include the times for shuffling the transferred data. The achievable read performance of the utilized SSD is 2.5 GB/s according to the UserBenchmark test suite. This puts the disk-to-GPU performance of our implementation (1.6GB/s) at 64% of the theoretically achievable disk-to-RAM performance (2.5GB/s).

(a) Heidentor (b) Retz (c) Arbegen (d) Vienna (e) Morro Bay

Figure 8: Data sets used in this paper. The same viewpoints were used in the rendering benchmarks in Table 3.

Data Set	File Size	#points	#attributes	bpp	pt / m^2	km^2	Acquisition Method	Sensor
Heidentor	0.7 GB	25.8 M	12	26	-	-	Photogrammetry	Nikon D300 + Photoscan
Retz	4.9 GB	145.5 M	13	34	56.43	2.58	ALS + TLS	RIEGL LMS-Q1560 + VZ-400i
Arbegen	6.7 GB	258.0 M	12	26	-	-	TLS	Zoller + Fröhlich Imager 5006h
Vienna	29.6 GB	276.7 M	50	107	57.52	4.81	ALS	RIEGL VQ-1560i
Morro Bay	13.8 GB	407.0 M	13	34	22.06	18.45	ALS	Leica ALS70 + Optech Orion

Table 1: Key parameters of used data sets. ALS: Airborne Laser Scanning. TLS: Terrestrial Laser Scanning. bpp: bytes per point.

4.3.3. Comparison to Hierarchical Structures

One of our claims is that our method allows users to quickly look at large point-cloud data that would otherwise require a relatively slow generation of hierarchical structures. In order to put this claim into perspective, we show how long it takes for state-of-the-art converters Potree [Sch16], Entwine [ENT], and Arena4D [A4D] to generate these acceleration structures out of our test data in Table 2.

We acknowledge that this is not an entirely fair comparison because these converters spend time reading data, writing it back to disk and potentially reading it again, whereas our progressive method does not have to write data back to disk. However, the data structures required to generate hierarchical acceleration structures also need additional memory during conversion, so the converters would not be able to hold as many points in memory as our progressive method during the conversion without flushing data back to disk.

4.3.4. Rendering

Table 3 shows the performance of our progressive approach, including the time it takes to render a single frame and the time it takes until the image converges. We also compare these timings to a brute-force approach where all the points are rendered in each frame. For the brute-force approach, shuffling is deactivated for two reasons: First, because brute-force approaches usually render the data in their original order. And second, because in the majority of cases we tested, rendering shuffled data was slower than rendering the same points in their original order that exhibited a certain amount of locality between consecutively stored points. A notable exception is the *Morro Bay* data set, where the shuffled vertex buffer renders faster from the chosen viewpoint in Figure 8, with almost all 407 million points located inside the view frustum. Once the user zooms in, the situation reverses and rendering the shuffled data becomes slower again. Table 4 illustrates these differences in performance. This difference in rendering times of shuffled and unshuffled buffers contributes to the fact that the convergence times for our progressive method are usually significantly higher than the rendering times of the brute-force approach, except for *Morro Bay* where the duration to convergence can be lower than the time needed to render the unshuffled data set with the brute-force approach.

A notable observation is made in Table 3 regarding the *Vienna* data set on system *1660*. If all 277 million points are rendered, the *Prepare* pass takes almost seven times as long as when only the first 230 million points are rendered. Similarly, the brute-force method requires more than double the time if all 277 million points are rendered, instead of only the first 230 million points. This is because at some point, the GPU will allocate shared system memory instead of dedicated GPU memory to our buffers. Average rendering times for the *Fill* pass do not change much because we render small subsets at a time and because most of the data is still rendered from GPU memory – only a subset that did not fit is rendered from system memory. The *Prepare* pass, on the other hand, slows down drastically because it now has to also read randomly shuffled point data from shared system memory as well. The total rendering time still remains below 16.6ms, which means even data sets that do not fit in GPU memory can be rendered in real time, but the time to convergence increases by a multiple.

Regarding depth complexity, we tested various viewpoints inside and outside the *Arbegen* point cloud. This data set consists of selected scan positions of the interior and parts of the exterior of a house with multiple rooms, an attic, a cellar and a hallway to the cellar. With the octree system of *Potree*, the point budget has to be increased to values of around 20 to 25 million points per frame to obtain a resolution of 1 point per 2x2 pixels, or as high as 55 million points to obtain a resolution of 1 point per pixel. The reason for this is because points that are hidden behind floors, ceilings, walls and noise have to be rendered due to the lack of occlusion culling. As a result, occluded points consume the majority of the render time, without contributing to the image. In the worst case, 55 million points were rendered but at most 2 million points were visible at a time on a 2 megapixel screen. Our progressive method, on the other hand, converges to the full image with a point budget as low as 1 million points per frame, plus the amount of points that are reprojected.

Model	Points	File Size	bpp	Hierarchical			Progressive			
				Converter	Duration	Points/s	Format	Load	Upload	Points/s
Heidentor	25.8 M	0.7 GB	26	Potree	36 s	0.72 M	LAS	0.56 s	0.70 s	37.18 M
				Entwine	40 s	0.60 M	VBO	0.30 s	0.41 s	63.48 M
				Arena4D	43 s	0.65 M				
Vienna	276.7 M	29.6 GB	107	Potree	611 s	0.45 M	LAS	32.75 s	32.92 s	8.40 M
				Entwine	301 s	0.92 M	VBO	3.36 s	3.36 s	82.34 M
				Arena4D	306 s	0.90 M				
Morro Bay	407.0 M	13.8 GB	34	Potree	776 s	0.52 M	LAS	14.31 s	14.36 s	28.35 M
				Entwine	564 s	0.72 M	VBO	4.05 s	4.05 s	100.45 M
				Arena4D	391 s	1.04 M				

Table 2: Load Performance. Time needed to create a hierarchical LOD structure in advance, compared to the time needed to directly load and render the non-hierarchical data with our progressive method. *Points/s*: Number of points per second processed. *LAS*: Load, parse and upload LAS files. *VBO*: Load and upload files in the same format as the vertex buffer. All benchmarks are done on system *2080* and its NVMe SSD.

Model	Points	System	MSAA	Brute-force	Progressive					
					Budget	Reproject	Fill	Prepare	Total	Converges In (ms)
Heidentor	25.8 M	2080	1x	6.87	1 M	0.10	0.33	0.15	0.71	18.3
		2080	1x	6.87	10 M	0.10	3.19	0.15	3.56	9.2
		1660	1x	10.10	1 M	0.20	0.96	0.29	1.68	43.5
		940	1x	44.02	1 M	1.27	8.39	2.67	12.72	328.7
Vienna	276.7 M	2080	1x	84.84	1 M	0.35	0.45	0.62	1.54	427.2
	276.7 M	2080	1x	84.84	10 M	0.35	4.49	0.72	5.68	157.1
	276.7 M	1660	1x	233.30	1 M	0.72	1.72	9.04	11.71	3 239.8
	230.0 M	1660	1x	107.81	1 M	0.72	1.84	1.35	4.31	992.0
Morro Bay	407.0 M	2080	1x	295.92	1 M	0.19	0.34	0.31	0.94	384.2
		2080	1x	295.92	10 M	0.19	3.34	0.34	3.99	162.3
		2080	4x	-	10 M	1.09	8.20	0.92	10.35	421.2
		2080	16x	-	10 M	4.01	19.39	3.09	26.60	1 082.5

Table 3: Rendering performance. (Brute-force) Time to render all points in a single frame. (Progressive) Time spent on the passes and the total time of a progressively rendered frame. (Budget) Number of points rendered in the *Fill* pass. All timings in milliseconds.

Model	Brute-force	Progressive	Shuffled?
Heidentor	9.69 ms	3.56 ms	yes
	6.87 ms	2.62 ms	no
Vienna	143.27 ms	5.68 ms	yes
	84.84 ms	4.43 ms	no
Morro Bay	150.97 ms	3.99 ms	yes
	295.92 ms	7.59 ms	no

Table 4: Performance difference of rendering shuffled and unshuffled vertex buffers. Progressive method rendered with 1x MSAA and a budget of 10M points per frame.

4.3.5. Virtual Reality

Our progressive method is able to maintain 2×90 frames per second in different viewpoints required by the HTC VIVE, at a resolution of 1448×1608 per eye on an RTX 2080 TI with 4xMultisample anti-aliasing (MSAA) and a fixed fill budget of 3 million points per frame after the data was fully loaded. For the *Vienna* data set, the frame rate targets are also achieved during loading if the fixed fill budget is lowered to one million points. In VR, the entire progressive rendering pipeline is executed twice, once for each eye. Since the frame rate is locked at 90fps and the fill-budget at 3 million points, the image converges at a rate of 180 million points per second. The number of points of the model affects time to convergence, but it does not affect rendering performance because we render at most $numReprojected + fillbudget$ points per eye in a frame. Since the pose of the head-mounted display always changes from frame to frame – even if it sits on a table, due to tracking noise – the result will closely approach but never truly reach convergence.

5. Limitations, Discussion and Future Work

In this section, we list and discuss some of the limitations of our current approach.

- Our method is currently in-core only. The complete data set must fit into CPU memory, and the position data and chosen attribute must fit into GPU memory. The GPU needs to store 16 bytes per point, but the CPU needs to store all the attributes to enable fast switching of attributes. However, implementations can also choose to keep attribute data in separate files on disk to quickly stream them to the GPU without the need to hold them in RAM.
- Shuffled vertex buffers as used in our method often render significantly slower than vertex buffers that are ordered by locality. The GPU may render 10 million ordered points in the same time as 7 million shuffled points. One could use non-shuffled buffers to reduce the duration to convergence, but the resulting flickering artifacts severely reduce visual quality. However, Table 4 also shows that there are cases where rendering shuffled buffers is faster.
- Our progressive method is developed for data without spatial acceleration structures. However, not having acceleration structures increases the duration to convergence since we cannot use frustum culling or culling by LOD to reduce the amount of points to the most viable candidates. Future work may explore the possibility of creating simple acceleration structures during loading, or afterwards in parallel to improve performance and quality during runtime.
- We currently do not offer a cost-effective method for quality improvements. MSAA works, but effectively acts as costly supersampling. The *Prepare* pass automatically treats each MSAA sample in the rendered image as if it was a separate pixel.
- Regions with higher point density will progress quicker than regions with low density, because the random subsets rendered by the *Fill* pass will also have a higher density in these regions.
- If points occupy more than one pixel or MSAA sample, then the *Prepare* pass will add them multiple times into the dynamically generated vertex buffer for reprojection. We tested an alternative implementation of the *Prepare* pass that operates on 2x2 frame buffer samples (= 4 pixels with no MSAA, 1 pixel with 4xMSAA) and only adds those points that are unique within that 2x2 sample grid. This form of approximate prevention of duplicates increased performance up to 10% for point sizes of 2x2 pixels and 4xMSAA. No performance improvements and sometimes performance losses of up to 5% were observed with smaller point sizes or without MSAA, because the duplicate prevention cost roughly as much time in the *Prepare* pass as it saves in the *Reproject* pass.

As part of future work, we would like to investigate suitable anti-aliasing strategies that are targeted specifically at this progressive approach.

6. Conclusion

We have shown a method that can render any point cloud that fits in GPU memory in real time without the need to generate acceleration structures in advance. This is achieved by distributing the task of rendering a large data set over the course of multiple frames by reprojecting the previous frame to preserve already rendered details, and then adding a random subset of points to drive progress until convergence. We believe that it has the potential to replace the brute-force rendering approach (all points in each frame) used in point cloud renderers that do not support LOD rendering, but also some LOD approaches as long as the point cloud fits into memory.

Progressive rendering allows us to efficiently render point clouds with high depth complexity, which hierarchical structures alone do not handle well. We believe that progressive rendering methods, such as ours for non-hierarchical data and Tredinnick and Ponto et al. [TBP16; PTC17] for hierarchical data, will prove to be essential to render increasingly complex scan data in real time.

Code and videos are available at github.com/m-schuetz/skye and www.cg.tuwien.ac.at/research/publications/2019/schuetz-2019-PPC/.

7. Acknowledgements

The authors with 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 sets of Vienna and the town of Retz, *PG&E* and *Open Topography* for funding and hosting the data set of San Simeon, including the Morro Bay area, and *TU Wien, Institute of History of Art, Building Archaeology and Restoration* for providing the Kirchenburg Arbegen (Figure 2 (b)) data set.

This research was enabled by the Doctoral College Computational Design (DCCD) of the Center for Geometry and Computational Design (GCD) at TU Wien.

References

[A4D] *Arena4D*. http://veesus.com/. Accessed 2019.10.02. URL: http://veesus.com/ 9.

[APS*14] ARIKAN, MURAT, PREINER, REINHOLD, SCHEIBLAUER, CLAUS, et al. "Large-Scale Point-Cloud Visualization through Localized Textured Surface Reconstruction". *IEEE Transactions on Visualization & Computer Graphics* 20.9 (Sept. 2014), 1280–1292. ISSN: 1077-2626. URL: https://www.cg.tuwien.ac.at/research/publications/2014/arikan-2014-pcvis/ 3.

[ASP19] ASPRS. *LAS Specification 1.4 - R14*. Rev. 14. The American Society for Photogrammetry & Remote Sensing (ASPRS). Mar. 2019 2, 7.

[Bad88] BADT JR, SIG. "Two Algorithms for Taking Advantage of Temporal Coherence in Ray Tracing". *The Visual Computer* 4 (May 1988), 123–132. DOI: 10.1007/BF01908895 3.

[BBS82] BLUM, LENORE, BLUM, MANUEL, and SHUB, MIKE. "Comparison of Two Pseudo-Random Number Generators". *Advances in Cryptology: Proceedings of CRYPTO '82*. Plenum, 1982, 61–78 4.

[Dre07] DREPPER, ULRICH. "What Every Programmer Should Know About Memory". (2007) 3.

[Dur64] DURSTENFELD, RICHARD. "Algorithm 235: Random Permutation". *Commun. ACM* 7.7 (July 1964), 420–. ISSN: 0001-0782. DOI: 10.1145/364520.364540. URL: http://doi.acm.org/10.1145/364520.364540 4.

[DVS03] DACHSBACHER, CARSTEN, VOGELGSANG, CHRISTIAN, and STAMMINGER, MARC. "Sequential point trees". *ACM Transactions on Graphics* 22 (July 2003), 657. DOI: 10.1145/1201775.882321 3.

[ENT] *Entwine*. https://entwine.io/. Accessed 2019.10.02. URL: https://entwine.io/ 9.

[FTB16] FUTTERLIEB, JÖRG, TEUTSCH, CHRISTIAN, and BERNDT, DIRK. "Smooth visualization of large point clouds". *IADIS International Journal on Computer Science and Information Systems* 11.2 (2016), 146–158. URL: http://publica.fraunhofer.de/dokumente/N-453338.html 3.

[FY38] FISHER, RONALD and YATES, FRANK. *Statistical Tables for Biological, Agricultural and Medical Research*. 1938 4.

[GM04] GOBBETTI, ENRICO and MARTON, FABIO. "Layered Point Clouds: A Simple and Efficient Multiresolution Structure for Distributing and Rendering Gigantic Point-sampled Models". *Comput. Graph.* 28.6 (Dec. 2004), 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 3.

[GZPG10] GOSWAMI, P., ZHANG, Y., PAJAROLA, R., and GOBBETTI, E. "High Quality Interactive Rendering of Massive Point Models Using Multi-way kd-Trees". *2010 18th Pacific Conference on Computer Graphics and Applications*. Sept. 2010, 93–100. DOI: 10.1109/PacificGraphics.2010.20 3.

[Ise13] ISENBURG, MARTIN. "LASzip: lossless compression of LiDAR data". *Photogrammetric Engineering & Remote Sensing* 79 (2013). DOI: 10.14358/PERS.79.2.209 2, 7.

[Jev92] JEVANS, DAVID A. "Object Space Temporal Coherence for Ray Tracing". *Proceedings of the Conference on Graphics Interface '92*. Vancouver, British Columbia, Canada: Morgan Kaufmann Publishers Inc., 1992, 176–183. ISBN: 0-9695338-1-0. URL: http://dl.acm.org/citation.cfm?id=155294.155315 3.

[OP] *OPALS - Orientation and Processing of Airborne Laser Scanning data*. https://opals.geo.tuwien.ac.at/html/stable/index.html. Accessed 2019.06.26. URL: https://opals.geo.tuwien.ac.at/html/stable/index.html 2.

[PMOK14] PFEIFER, N., MANDLBURGER, G., OTEPKA, J., and KAREL, W. "OPALS - A framework for Airborne Laser Scanning data analysis". *Computers, Environment and Urban Systems* 45 (2014), 125–136. ISSN: 0198-9715. DOI: https://doi.org/10.1016/j.compenvurbsys.2013.11.002. URL: http://www.sciencedirect.com/science/article/pii/S0198971513001051 2.

[Pre12] PRESHING, JEFF. *How to Generate a Sequence of Unique Random Integers*. Accessed 2019.09.16. 2012. URL: https://preshing.com/20121224/how-to-generate-a-sequence-of-unique-random-integers/ 4.

[PTC17] PONTO, KEVIN, TREDINNICK, ROSS, and CASPER, GAIL. "Simulating the experience of home environments". *2017 International Conference on Virtual Rehabilitation (ICVR)*. June 2017, 1–9. DOI: 10.1109/ICVR.2017.8007521 3, 11.

[RL00] RUSINKIEWICZ, SZYMON and LEVOY, MARC. "QSplat: A Multiresolution Point Rendering System for Large Meshes". *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, 343–352. ISBN: 1-58113-208-5. DOI: 10.1145/344779.344940. URL: http://dx.doi.org/10.1145/344779.344940 3.

[Sch14] SCHEIBLAUER, CLAUS. "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: https://www.cg.tuwien.ac.at/research/publications/2014/scheiblauer-thesis/ 2, 3.

[Sch16] SCHÜTZ, MARKUS. "Potree: Rendering Large Point Clouds in Web Browsers". MA thesis. Austria: TU Wien, 2016. URL: https://www.cg.tuwien.ac.at/research/publications/2016/SCHUETZ-2016-POT/ 3, 9.

[SYM*11] SCHERZER, DANIEL, YANG, LEI, MATTAUSCH, OLIVER, et al. "A Survey on Temporal Coherence Methods in Real-Time Rendering". Jan. 2011 3.

[TBP16] TREDINNICK, R., BROECKER, M., and PONTO, K. "Progressive feedback point cloud rendering for virtual reality display". *2016 IEEE Virtual Reality (VR)*. Mar. 2016, 301–302. DOI: 10.1109/VR.2016.7504773 2, 3, 11.

[WBB*07] WAND, MICHAEL, BERNER, ALEXANDER, BOKELOH, MARTIN, et al. "Interactive Editing of Large Point Clouds". *SPBG*. 2007 2, 3.

[WDP99] WALTER, BRUCE, DRETTAKIS, GEORGE, and PARKER, STEVEN. "Interactive Rendering using the Render Cache". *Rendering Techniques' 99*. Ed. by LISCHINSKI, DANI and LARSON, GREG WARD. Vienna: Springer Vienna, 1999, 19–30. ISBN: 978-3-7091-6809-7 3.

[Wei16] WEINMANN, MARTIN. *Reconstruction and Analysis of 3D Scenes: From Irregularly Distributed 3D Points to Object Classes*. Springer International Publishing, 2016. ISBN: 978-3-319-29244-1. DOI: 10.1007/978-3-319-29246-5 1.

[WS06] WIMMER, MICHAEL and SCHEIBLAUER, CLAUS. "Instant Points: Fast Rendering of Unprocessed Point Clouds". *Proceedings Symposium on Point-Based Graphics 2006*. Eurographics. Boston, USA: Eurographics Association, July 2006, 129–136. ISBN: 3-90567-332-0. DOI: 10.2312/SPBG/SPBG06/129-136. URL: https://www.cg.tuwien.ac.at/research/publications/2006/WIMMER-2006-IP/ 3.