To appear in IEEE VR 2019, the 26th IEEE Conference on Virtual Reality and 3D User Interfaces

Real-Time Continuous Level of Detail Rendering of Point Clouds

Markus Schütz*
TU Wien

Katharina Krösl†
TU Wien,
VRVis Forschungs-GmbH

Michael Wimmer‡
TU Wien

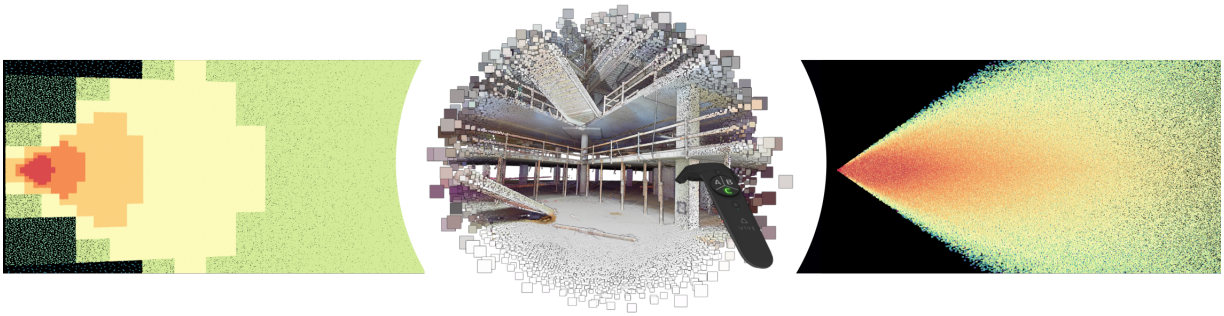


Figure 1: Left: Discrete LOD structure with sudden jumps in density and popping artifacts under motion. Middle: Screenshot of our method in virtual reality. No visible seams across different levels of detail due to a continuous reduction in density, and also continuously reduced density towards the periphery to reduce geometric complexity where less is needed. Endeavor point cloud courtesy of NVIDIA [4]. Right: Continuous transition from one LOD to another.

Abstract

Real-time rendering of large point clouds requires acceleration structures that reduce the number of points drawn on screen. State-of-the art algorithms group and render points in hierarchically organized chunks with varying extent and density, which results in sudden changes of density from one level of detail to another, as well as noticeable popping artifacts when additional chunks are blended in or out. These popping artifacts are especially noticeable at lower levels of detail, and consequently in virtual reality, where high performance requirements impose a reduction in detail.

We propose a continuous level-of-detail method that exhibits gradual rather than sudden changes in density. Our method continuously recreates a down-sampled vertex buffer from the full point cloud, based on camera orientation, position, and distance to the camera, in a point-wise rather than chunk-wise fashion and at speeds up to 17 million points per millisecond. As a result, additional details are blended in or out in a less noticeable and significantly less irritating manner as compared to the state of the art. The improved acceptance of our method was successfully evaluated in a user study.

Index Terms: Computing methodologies—Computer graphics—Rendering; Computing methodologies—Computer graphics—Graphics systems and interfaces—Virtual reality

1 Introduction

Point-cloud rendering poses a variety of challenges, such as the enormous amount of points that is necessary to represent a model in similar detail as textured meshes, as well as aliasing artifacts due to poor scan quality and the lack of mipmapping. One of the most common sources of point-based models are 3-dimensional scans of real objects, buildings and even whole countries. Smaller objects and structures may consist of a few million coloured points in millimeter resolution, whereas scans of large areas and whole countries may consist of hundreds of billions of points.

Typically, hierarchical acceleration structures are used in order to efficiently load and render these amounts of data, addressing two aspects: The reduction of data to an amount that can be rendered in real time, and out-of-core processes to load and unload data as needed, especially if the whole data set is larger than the available memory. In this paper, we will concentrate on the first aspect, the in-core rendering of data sets that fit in GPU memory, but which are still too large to be rendered in real time in VR. State-of-the art methods address this by generating level-of-detail structures that group points into chunks of various extent and density. Smaller chunks with higher density and detail are visible up to a certain distance from the viewer. As the distance to the viewer increases, the density of the chunks to be rendered decreases. Chunk-wise handling of LODs has emerged as the state of the art because a coarse-grained management of data reduces overhead on file I/O, network transfers, graphics draw calls, etc., as opposed to handling each point individually. The disadvantage, however, is that these chunks are noticeable in the rendered image, especially at lower levels of detail and during motion.

Virtual Reality (VR) introduces additional issues in point-cloud rendering, such as drastically increased performance and quality requirements. The HTC Vive and Oculus Rift head-mounted displays (HMDs) both require a frame rate of 90 frames per second (FPS) per eye, for a total of 180 FPS. Render target sizes are also relatively large, with display resolutions of 1080×1200 for the HTC Vive and the Oculus Rift, and a recommended resolution that is about 1.4 times higher in each direction to account for distortion and aliasing [26]. Furthermore, anti-aliasing becomes mandatory because aliasing and other rendering artifacts are much more noticeable in VR. Therefore, the level of detail of a point cloud has to be reduced considerably in order to meet these high performance requirements. Unfortunately, this augments noticeable popping artifacts as larger chunks of points are blended in and out during motion.

The distortion effect of the lenses inside HMDs also results in wasteful rendering at the outer regions of the image. The lenses create a pincushion distortion that has to be countered by a barrel distortion before displaying the rendered image. By default, this barrel distortion is applied to the rendered image and strongly

*e-mail: mschuetz@cg.tuwien.ac.at
†e-mail: kkroesl@cg.tuwien.ac.at
‡e-mail: wimmer@cg.tuwien.ac.at

compresses outer regions. This results in an effectively reduced resolution for outer regions and thus needlessly rendered pixels [26]. NVIDIA's multi-res shading and lens-matched shading address this issue by rendering outer regions at lower resolutions [13], but this only reduces shading cost, not geometry cost as is relevant for point clouds.

Fig. 1 shows an example of a model that is particularly difficult to render using state-of-the art discrete level-of-detail (DLOD) mechanisms in VR even though it is only 86 million points: It has high depth complexity from most viewpoints due to multiple floors and additional structures like fences and wires. The commonly used octrees and kd-trees produce chunks that do not align well with arbitrarily oriented walls, pillars or stairs. Also, the sizes of the chunks are limited in granularity, which makes frustum culling, but also focusing on details towards the center, less efficient.

In this paper, we propose a continuous level-of-detail (CLOD) method for point clouds that addresses the challenges of real-time point cloud rendering in VR through the following contributions:

1. Our method eliminates chunk-wise popping artifacts prevalent in state-of-the art DLOD methods, and evaluates in a point-wise rather than chunk-wise fashion which points to render.

2. The change of detail as users move through the scene is much less noticeable due to a subtle point-wise fading approach.

3. Our method exhibits a smooth transition in density as the distance to the viewer increases, as opposed to sudden jumps in density prevalent in DLOD methods.

4. The point density is decreased away from the center of the image, in order to account for the reduced resolution after barrel distortion in VR. As a result, significantly fewer points have to be rendered in the periphery.

In addition, our implementation avoids hierarchy traversal and culling based on distance, frustum or visibility, yet manages to match the performance and exceed the perceived quality of an octree-based approach that culls nodes by frustum and distance. Rather than traversing through a tree, finding visible nodes, and then dispatching draw calls for each of them, we update a vertex buffer and render the result with a single call to glDispatchCompute and glDrawArraysIndirect per frame.

2 Related Work

Luebke et al. [15] describe a wide variety of algorithms, problems and solutions in the field of level-of-detail rendering, with a focus on meshes, and a whole chapter on LOD for terrain rendering as a special case. According to the terminology of the book, our method can be classified as a view-dependent, continuous LOD method with fidelity-based simplifications. Alternatively, methods may be classified as discrete LOD methods with either fidelity or budget-based simplification.

Terrain rendering is a particularly popular field for LOD rendering since terrain models are usually meshes with a lot of polygons covering a large area, with most of the polygons being either outside the view frustum, or so far away from the camera that fine details are not visible. Also, commonly used height-mapped terrains provide additional opportunities for optimizations that are not available to meshes of arbitrary complexity. Lindstrom et al. [14], Sander and Mitchell [20], and Schneider and Westermann [23] describe methods to render terrain with smooth transitions between levels of detail. The process of smoothly transforming the geometry of a mesh from one LOD to another is referred to as geomorphing, and it helps to avoid popping artifacts in terrain rendering as users move through the scene. This is not exclusive to terrain, however, and methods such as *Progressive Meshes* describe geomorphing techniques for meshes of arbitrary complexity [12]. Scherzer and Wimmer also proposed an image-space technique that achieves a smooth transition by interpolating between renderings of different LODs, rather than adjusting the geometry [22].

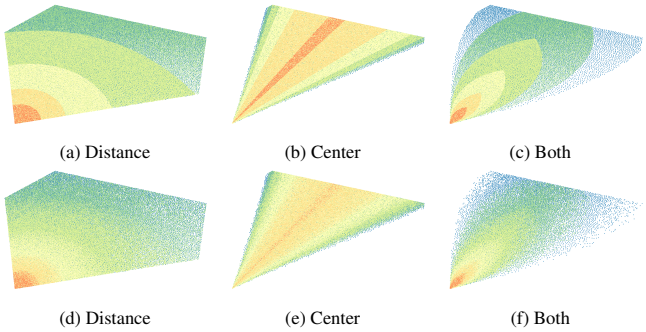
Surfels [18] and QSplat [19] were the first proposed multiresolution methods for the point-based rendering of meshes. Surfels create an octree-based data structure from the original mesh and focus on high-quality rendering. QSplat, on the other hand, uses a bounding-sphere hierarchy and has its focus on the interactive and progressive rendering of large point clouds.

Following the advances of GPUs, Dachsbacher et al. [7] proposed sequential point trees, a GPU-friendly approach that first generates a point tree hierarchy similar to QSplat, and then sequentializes it into an array, ordered by hierarchy level. The sequential array can be quickly rendered by the GPU, and the level of detail can be reduced by rendering the first x points of the hierarchically sorted points. Gobbetti and Marton [10] were the first to use an LOD approach that groups points into a hierarchy of chunks where each chunk contains points that are added to the points contained in the parent. Chunk-based rendering is very efficient for GPUs and has become the de-facto standard for displaying large point clouds. Subsequent research introduced various improvements and alternatives on how those chunks are generated, their size, shape, and hierarchical properties. In particular, Wimmer and Scheiblauer [28] introduced a nested octree system where each node of the outer octree contains an inner octree, which is a memory-optimized version of a sequential point tree [7]. These inner octrees constitute the chunks mentioned above, and are used to sample the points that each outer octree node contains. They later simplified this to sampling points on regular grids for each node to allow for efficient addition and removal of points [21]. Schütz [25] uses the same structure, but selects points following a Poisson-disk sample distribution to improve the appearance of the subsamples [6]. Another structure for addition and removal of points was previously proposed by Wand et al. [27], which stores the original points in its leaf nodes, and representative points in inner nodes. Goswami proposed a multi-way kd-tree structure that achieves a better balance in the number of points in each node and reduces unnecessary branching [11].

Futterlieb et al. [9] propose a point-based rendering approach that preserves details from the previous frame if the camera does not move and then adds additional detail. They also reduce popping artifacts through an image-based LOD-blending method of multiple render targets, whereas our method avoids popping by point-wise filtering and size adjustments. Discher et al. [8] built a virtual reality point-cloud renderer with a kd-tree as its level-of-detail structure, hidden-mesh rendering optimization to skip fragments that are not visible on the HMD, and a node-wise drawing order from front to back to exploit early fragment testing.

Many of these LOD schemes are able to handle arbitrarily large point clouds in theory. Entwine [1] and Massive-PotreeConverter [16] are two implementations that put this into practice, being able to create an LOD structure for point clouds of more than 600 billion points.

All of the discussed methods either suffer from artifacts due to chunked LOD representations, or do not scale well enough to render point clouds in VR. Work in the related field of molecular visualization already covers the continuous level of detail for molecules, and a smooth transition between different LODs of a molecule and the transition from an abstract molecule shape to its atoms. Le Muzic et al. [17] developed the first tool, CellView, that is capable of visualizing scenes with large amounts of molecules and up to 15 billion atoms at 60 Hz. Aside from accelerating the rendering process, their level-of-detail method also serves to reduce the visual clutter and display smooth abstract molecular shapes, rather than clusters of atoms. This is possible due to the specific structure and semantics of atoms and molecules.



(a) Distance (b) Center (c) Both

(d) Distance (e) Center (f) Both

Figure 2: Point density reduction by distance to camera, distance to the center of the screen, and both. Top-row: Filtering based on the level of a point inside the hierarchy, with sudden drops in density. Bottom-row: Continuous results with dither-like patterns after adding a random value between 0 and 1 to the level of each point. Colors represent the level attribute of a point.

3 Continuous Level of Detail

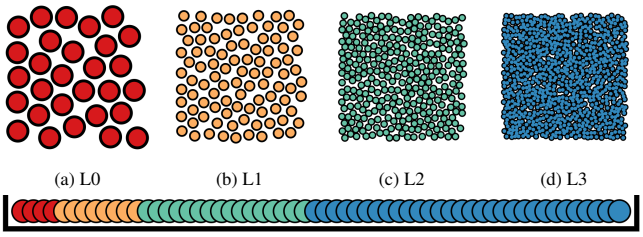
The basic idea behind our continuous level-of-detail method is to repeatedly create a reduced low-resolution version of the full point cloud at runtime, based on the current camera orientation and position, and with a gradual reduction in density as the distance to the camera increases. Our current implementation skips common optimizations such as frustum culling or hierarchical traversal. Instead, it iterates through all points of the data set to identify the points that should be rendered and stores the relevant points in a new vertex buffer. This reduced model is created over the course of a few frames – fast enough to give the impression that the model is always up-to-date, yet slow enough so as not to take too much performance away from the actual rendering process. We are able to create an updated version of the reduced model every 5 to 6 frames on a GTX 1080 by allocating around 1.1 milliseconds to the reduction step for point clouds up to 104 million points.

The main aspect that makes our method continuous is a runtime randomization of state-of-the-art discrete structures. These discrete structures organize points in level 0, 1, and so on. Adding random numbers between zero to one to the level of each point in such a discrete hierarchy then allows us to filter on a continuous scale rather than integer intervals. At a distance of 9.3 meters, we may want to display points up to level 2.3, for example. The results exhibit a continuous smooth transition in density with dither-like patterns, as shown in Fig. 2.

3.1 Data Structure

When designing a chunk-based LOD system for points, one has to decide whether data from higher (more detailed) levels should be added to the data from lower levels (corresponding to memory optimized sequential point trees [28]), or replace it (corresponding to the original sequential point trees [7], and similar to the data structure by Wand et al. [27]). The advantage of the additive approach is that it does not require additional memory and we do not have to take care of removing lower levels before displaying a higher level. The advantage of replacing lower levels is that each level can store and display representative points for the current level without data from another level inbetween.

We chose to primarily implement an additive scheme for performance reasons. However, instead of storing the points in a hierarchy as in the original schemes, our CLOD data structure is a single flat array, with the hierarchy level stored as a point attribute, and the structure is evaluated in a point-wise fashion on the GPU rather than a chunk-wise fashion on the CPU.



(a) L0 (b) L1 (c) L2 (d) L3

(e) All points stored in a single vertex buffer, preferably but not necessarily ordered.

Figure 3: Our CLOD structure is a series of subsamples of the original point cloud that is then flattened into a single array.

Points are subsampled by enforcing a level-dependent spacing between points, given by

$$spacing_{level} = \frac{rootSpacing}{2^{level}} \tag{1}$$

For example, level 0 contains points with a spacing of 1 meter, level 1 points with a spacing of 0.5 meters, and so on. Each point is assigned to exactly one level, and merging all levels results in the original point cloud.

To reduce aliasing at lower levels, we borrow the idea of averaging from the replacement scheme: points that are assigned to lower-level nodes keep their original position but have their original color value replaced by an average color over the area they represent. This is not an entirely accurate solution since points with averages over different radii will be intermixed due to the additive LOD scheme, but it is significantly better for the visual quality than aliasing effects from keeping the original color values. Overblurring occurs but remains a minor issue because in any given view, points with the correct averaging radii from high LODs far outnumber points with larger averaging radii from lower LODs.

The order of the points inside the array is not important because our in-core method repeatedly iterates over all points to produce a downsampled version at runtime. However, we still recommend to order them from lowest level to highest level because the method can be applied to any subset of the data, and ordering points from lowest level to highest level allows us to display a coarse version of the whole model while increasingly higher levels of detail are still being loaded.

3.2 Build-Up

The build-up step for our CLOD structure uses the publicly available PotreeConverter [24] [25] to organize points into an octree, and custom scripts to average colors and flatten the hierarchy into an array.

The PotreeConverter creates an octree out of a point cloud that can be used to stream and render only relevant chunks of points up to a certain level of detail. It also selects points based on the spacing between points as given by Equation 1, which we need for our CLOD simplification algorithm .The result exhibits two issues, however, that we addressed with additional custom scripts.

The first issue is aliasing as discussed above, as points in lower-resolution octree levels store the color from a single input point, rather than the average over the area they represent. We address this issue by computing the arithmetic mean of the color of a point in $level_n$, and the colors of all points at $level_{n+1}$ within the distance defined by *spacing* at level n.

The second issue is the large amount of relatively sparsely populated nodes that are generated, and then stored in separate files. Each node consists of around 100 to 10,000 points. Handling a large amount of small files results in I/O overhead that is unnecessary since our method does not deal with individual tiles, but rather all

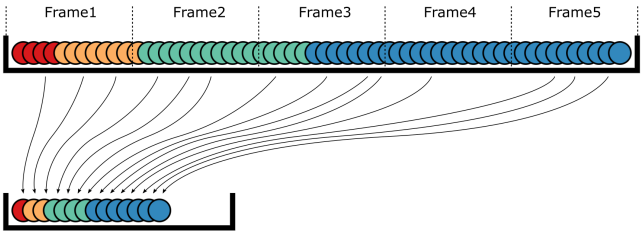


Figure 4: Over the course of a few frames, the compute shader iterates through all points of the full point cloud and copies the subset that will be rendered into a separate vertex buffer.

points as a single large blob. We therefore concatenate all the octree nodes into a single file. The only hierarchical information that is kept is the octree level of a point, which is stored as a byte inside the alpha component of the point's color.

The result of the build-up step is an array of points where each point contains position, an average color over the area it represents, and its level within the hierarchy. Each point requires 16 bytes: 12 bytes for position, 3 bytes for color, and 1 byte for the hierarchy level.

3.3 Rendering

The rendering process consists of two steps:

1. Reduce: Repeatedly recreating a reduced vertex buffer over the course of a few frames, based on view-frustum and a CLOD-factor.

2. Draw: Rendering the most recent reduced vertex buffer.

3.3.1 Reduce

The reduce step creates a new vertex buffer by iterating through all points in the full point cloud and copying those that match the desired level of detail in the target buffer, as shown in Fig. 4.

We want to obtain a vertex buffer with a specific target spacing between points, depending on the distance to the camera. As a baseline, we would like a point spacing of 1 millimeter at a distance of 1 meter. The baseline is adjustable by a CLOD-factor and multiplied by the distance. This definition for the baseline makes it independent of the field of view and the resolution of the target devices, so that users will get to see the same points for the same viewpoint, except for additional points in the periphery with higher fields of view.

For desktop use, this desired minimum spacing (millimeters) between points at any given distance to the viewer (meters) is computed as

$$targetSpacingDesktop = \frac{distance \cdot CLOD}{1000} \tag{2}$$

In order to account for lens distortions and the resulting reduction in resolution in outer regions of the image, we compute the desired spacing in VR as

$$targetSpacingVR = \frac{targetSpacingDesktop}{max(1 - a \cdot dc, minDensity)} \tag{3}$$

The denominator reduces the density in the periphery, which leads to a significantly reduced workload for the vertex shader. *dc* specifies the distance to the center in *normalized device coordinate* space, without depth. *dc* is zero at the center and one at the border of the ellipse inscribed within the screen. *a* is used to adjust how fast the density decreases and *minDensity* specifies a lower limit on the reduction in percent. We suggest values of 0.5 for *a* and 0.3 for *minDensity*.

We also tried to reduce the density from the center based on a Gaussian function but found no significant improvement. We therefore settled with the simpler Equation 3. Fig. 2 shows the result of the reduction for Equation 2 and Equation 3.

In VR, the reduce step is executed once per frame for a single HMD-centered frustum that covers both eyes, and the same reduced model is then rendered for both eyes. During quick motions, it will be noticeable that the currently rendered model is missing points outside of this view frustum because it takes 5 to 6 frames to produce an updated model. To alleviate this issue, the reduce shader clips against an extended frustum by projecting a point to normalized device coordinates, and then clipping the x and y axes against a range of $[-2,2]$ instead of $[-1,1]$. We suggest to set *minDensity* to around 0.3 to capture additional points in the extended frustum in a low but sufficient density to account for quick head movements.

The reduce operation is implemented as a compute shader that iterates over all points and stores the points that pass our CLOD requirements in a new buffer, as shown in Fig. 4. The requirements are evaluated by first clipping against the extended frustum, and then comparing the spacing of the currently processed point to the target spacing at that point's location. If the spacing of that point is smaller than the targeted spacing, the point will be discarded because it represents a higher level of detail than necessary. The spacing of a point is obtained from its level in the hierarchy by applying Equation 1. The spacing can alternatively be interpreted as the amount of space that this point occupies for itself. If the targeted spacing is large, only points that occupy a respectively large space should be visible.

At this stage, all the points are still classified in discrete integer levels, which continues to produce sharp drops in density. In order to produce smooth transitions, we add a random factor between 0 to 1 to the level of the point. This pseudo-random factor is different for each point, but remains the same for each point over time.

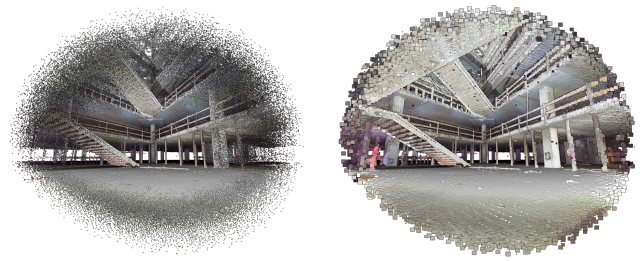
The randomization of the level of a point also randomizes the spacing, the claimed minimum distance to another point at this hierarchy level. Since we add to the level, the reported spacing is reduced. Some points of the same hierarchy level are now more likely to fail the targeted spacing requirements than others, which leads to a smooth falloff in density.

3.3.2 Draw

The draw step renders the most recent fully reduced vertex buffer, as created by the reduce step. Since the vertex buffers are created directly on the GPU, we use the *gl.drawArraysIndirect* command to render the data without the need to send the vertex buffer, or the number of points it contains, back to the CPU.

Point sizes are set to *targetSpacing* (millimeters) in order to fill the holes that appear due to the reduce step, as shown in Fig. 5b. Points with a lower spacing are discarded in the reduce step, and resizing the remaining points makes up for the reduced density. However, this only deals with holes that would be caused by our CLOD method. It does not deal with holes due to insufficient or varying sample density in the original point cloud. We suggest to additionally specify a minimum point size in millimeters, based on the sample density of the 3D scanner, to avoid huge gaps between points upon closer inspection by the user.

At this stage, moving through the scene still exhibits irritating popping artifacts of individual points if points accepted by the reduce shader immediately appear with their full world-space size of *targetSpacing*. Using *targetSpacing* as the world-space point size results in pixel sizes that are independent of the distance to the viewer. As users get closer to a point, the *targetSpacing* at this point's location, and therefore that point's world-space size, shrinks, but its resulting pixel size remains the same. This means that depending on the CLOD factor, newly accepted points pop in at a certain pixel size (e.g., 5 pixels). To improve from point-wise popping to point-wise



(a) Fixed size of 2 pixels (b) Point sizes adjusted by density

(c) Identifying points in blend-range (red) (d) Reducing size of points in blend-range

Figure 5: Result of the reduction step and application of a blend-threshold to fade-in additional detail.

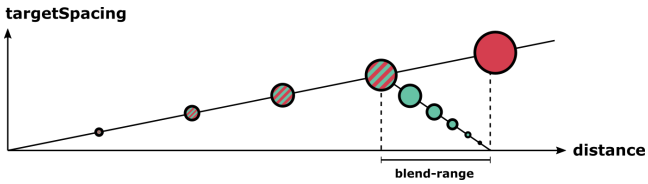
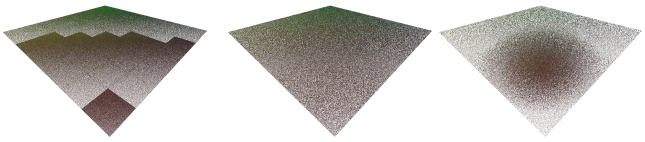


Figure 6: Point sizes are set to targetSpacing in order to fill holes from filtering by targetSpacing. To avoid points popping in at full size the moment they become visible (red), we let them grow to full size within a certain blend-range (green).

fading, we propose an additional blend range within which points grow to their full size, as shown in Fig. 6. At the moment when a point first becomes visible, that is as soon as $targetSpacing = pointSpacing$, its size is set to 0. Once we get closer, the point grows until it reaches its full size of $targetSpacing$ when $targetSpacing = blendRangeFactor \cdot pointSpacing$. Within the blend range, point sizes are linearly interpolated between $[0, targetSpacing]$. We suggest a value of 0.8 for the $blendRangeFactor$ so that users have to move another 20% closer to the point until it reaches its full size. Note that this point-wise fading method depends on $targetSpacing$ but not on time. As a result, points fade in or out depending on the distance to the user as well as distance to the center of the screen. Note also that the blend range is defined as a fraction of the $pointSpacing$, and is therefore larger at lower levels of detail. It may take 10 meters until points of lower LODs grow to their full size and only 1 meter for points at higher LODs, but it is always the same fraction of distance from the appearance of a point to full growth.

4 Results

Fig. 7b shows that our CLOD approach achieves a more uniform distribution of points in screen-space, as opposed to DLOD methods like in Fig. 7a. The desired density at any location can be specified in the reduce shader by any mapping of a 3D coordinate to a target spacing. Equation 2 and Equation 3 are the two we propose for desktop and VR rendering, respectively, but they can be replaced or combined with other mathematical equations.



(a) DLOD (b) CLOD (c) CLOD focused

Figure 7: Distribution of 150k points with DLOD and CLOD.

4.1 Implementation

Our CLOD renderer is implemented in JavaScript based on Google's V8 engine [3], with custom bindings to OpenGL 4.5 and OpenVR [2]. While fast for a scripting engine, tree-traversal of hundreds to thousands of octree nodes, as done by Potree [25], is an expensive operation in JavaScript that, combined with also relatively expensive OpenGL draw calls, can cost a couple of milliseconds of CPU time per frame. A considerable advantage of the proposed CLOD implementation in scripting engines is that it eliminates tree-traversal of the point cloud and reduces draw calls to a single call to *glDispatchCompute* and *glDrawArrayIndirect*, at the cost of reserving 1 millisecond of GPU time per frame for the reduce shader.

Source code samples for this work are available at https://github.com/m-schuetz/ieeevr_2019_clod.

4.2 Performance

OpenGL execution times were measured with glQueryCounter, which returns the timestamp after all previously scheduled OpenGL commands have finished. The difference in timestamps gives us the duration of the OpenGL commands that were executed in-between. The validity of the results of glQueryCounter were cross-checked with NVIDIA's performance profiler Nsight.

Compared with the octree-based DLOD method of Potree, our CLOD method requires an additional millisecond per frame to create the reduced vertex buffer, but frees the CPU from tree-traversal and reduces the number of draw-calls down to a single call to *glDispatchCompute* and *glDrawArrays*. Draw performance is similar for both approaches with a similar amount of detail. Assuming the same number of rendered points, the CLOD approach just redistributes points more uniformly.

4.2.1 Performance of Reduce Pass

In this section, we would like to justify our decision not to use hierarchical culling algorithms, and show that repeatedly iterating over all points of the full point cloud is feasible for our test data of up to 104 million points.

We benchmarked the reduce shader on three systems: a GTX 1080 desktop system with an Intel i7-3770K processor, a GTX 1060 desktop system with an AMD Ryzen 5 1600X, and a 940 MX notebook system with an Intel i7-7500U.

Table 1 contains profiling data of the reduce step for the point cloud of the Endeavor building site and the Matterhorn on all three test systems. On a GTX 1080, point clouds can be reduced at a rate of 16 to 17 million points per millisecond. The reduction of 86 million points down to 5 million points takes about 5.45 milliseconds out of 16.6ms for 60Hz desktop renderings or 11.1ms for 2x90Hz VR rendering. In practice, the available time to complete a frame in VR is closer to 9ms, and the reduced data set has to be rendered twice, once for each eye. The reduce step is therefore distributed over multiple frames. The profiling information in Table 1 helps to estimate how many points it should process in each frame. Allocating around one millisecond to the reduce step means we can process around 16 million points per frame on a GTX 1080, and obtain the final reduced result after 5 to 7 frames for point clouds of 86M points and 104M points. Due to its lower memory capacity,

the 940MX was profiled only for the first 50 million points of the data set, and a maximum target buffer size of 5 million points.

Table 1: Performance of the reduce step.

model	GPU	reduced to	duration	points/ms
Endeavor (86M points)	GTX 1080	1.0M	5.22ms	16.58M
		5.0M	5.45ms	15.90M
		10.0M	6.04ms	14.33M
	GTX 1060	1.0M	8.41ms	10.30M
		5.0M	9.20ms	9.41M
		10.0M	10.25ms	8.45M
Matterhorn (104M points)	GTX 1080	1.0M	6.25ms	16.76M
		5.0M	6.60ms	15.87M
		10.0M	6.95ms	15.07M
Endeavor (50M points)	940MX	1.0M	60.11ms	0.83M
		5.0M	65.00ms	0.77M

Table 2: Theoretical memory bandwidth according to NVIDIA Control Panel and the utilization by the reduce shader.

GPU	bandwidth	utilized	rate
GTX 1080	320 GB/s	267.52 GB/s	84%
GTX 1060	192 GB/s	165.12 GB/s	86%
940MX	14.4 GB/s	13.28 GB/s	92%

Table 2 shows how much of the theoretically available memory bandwidth is utilized by the reduce shader. The values are computed by taking the numbers for *points / ms* in Table 1 for a reduction to 1 million points, and multiplying them by 16 to convert from points per millisecond to megabytes per millisecond. We believe the numbers to be reasonable because the reduce shader does little more than copying a tenth or less of the input data to the target buffer.

The reduction step has an overall relatively stable and predictable performance that is mostly dependent on the number of input points, and to a lesser extent on the number of output points, mostly because the number of output points is at least an order of magnitude lower than the number of input points.

4.2.2 Performance of Draw Pass

Draw performance depends on a variety of factors and greatly differs between different viewpoints, even with the same number of points and MSAA settings, as shown in Table 3. Factors that affect performance include drawing order, point size and coverage. Drawing points from front to back is preferable to exploit early fragment testing. A certain geometric locality of points that are stored consecutively in the vertex buffer also benefits draw performance compared to a shuffled vertex buffer, which is why we keep points in the same order as they were stored inside the octree nodes during build-up. Drawing the same amount of points with the same point sizes to a smaller part of the screen can also significantly improve performance by a factor of two.

The numbers in Table 3 are meant to illustrate the range of draw performances in regular use cases and are by no means an exhaustive benchmark. We did not exploit early fragment testing but we believe there is opportunity to do so, for example by writing the results of the reduce step into different target buffers, depending on view-distance.

5 User Study

The improved acceptance of our method was evaluated in a user study with 23 participants. We chose to evaluate our results by a user study instead of comparing error metrics to a ground truth because

Table 3: Draw performances of the Endeavor point cloud from different views with 1x to 8x MSAA into a framebuffer with 1920 x 1080 pixels on a GTX 1080, with our CLOD method compared to the octree method of Potree.

Method	View	#points	duration (ms)		
			1xAA	4xAA	8xAA
CLOD	1	2M	0.73	2.04	3.92
		4M	1.10	2.46	4.83
	2	2M	0.78	3.06	5.43
		4M	1.07	3.08	6.04
Octree	1	2M	0.74	2.15	3.67
		4M	1.19	2.78	4.82
	2	2M	0.86	2.63	5.15
		4M	1.73	3.16	5.08

popping artifacts, perceived quality, and potential issues with density reductions in the periphery are largely a perception issue.

We invited staff of a visual computing research center, staff of an archaeology research center, and students and staff of the visual computing group of a university to participate in our user study. All three institutions work with point clouds to a certain extent, so many of our participants have already captured, processed, or visualized point clouds before. Our participants were between 25 to 59 years old, 78% were male and 22% female. All participants are regular users of computers. Two participants have never experienced VR, 5 and 7 rarely and sometimes experience VR, and 9 participants regularly or often experience VR. 7 participants work sometimes with point clouds, and another 7 regularly or often. 6 users have never experienced point clouds in VR before, 12 users rarely, 2 sometimes and 3 regularly work with point clouds in VR.

Users were confronted with two point clouds, Matterhorn and Endeavor, in four scenarios inside the VR environment. They could freely switch between three LOD methods, labeled A, B, and C, with the touch pad on the controller. Method A is an octree-based level-of-detail system, implemented after Potree [25], which adjusts point sizes to the level of detail. Method B is based on the same octree approach, but points have a fixed pixel size. Method C is our approach that renders with continuous level of detail.

The four scenarios are:

1. Matterhorn: 104M points; high level of detail.
2. Endeavor Small-Points: 86M points; low level of detail; small point sizes, insufficient to fill holes
3. Endeavor Large-Points: 86M points; medium level of detail; point sizes for method A and B adapted to cover holes
4. Endeavor Illuminated: 86M points; medium level of detail; no colors, each point is white, but illuminated by Eye-Dome-Lighting [5]. We decided to evaluate this case because the original colored data exhibits strong noise and flickering artifacts since it is a combination of individual scans with different light exposures.

Tests were executed on a VIVE (6 participants) and a VIVE Pro (17 participants), depending on availability. Render-target resolution was set to about 1512 times 1680 pixels for both in the SteamVR settings menu. Level of detail was set conservatively to maintain 90 fps in the worst case on all of our test systems, which were equipped with either a GTX 1070 or GTX 1080. The point budget for the octree methods, and the CLOD factor for our method was set to render no more than 3 million points at a time, so that participants

were presented with the same images (apart from HMD resolution) regardless of the used system.

The workflow for each participant was as follows:

1. Hand participant an informed user consent form to sign.
2. Ask questions about the demographics. We asked the user's age, gender, their occupation, and how often they work with computers, virtual reality, point clouds, and point clouds in virtual reality.
3. Show users the four scenarios, one after another. During each scenario, users were asked to rate their overall impression, how noticeable changes in the level of detail are, and how irritating these changes are on a scale of 1 to 10 for each method. We filled out their answers in Google forms so that they could remain inside the VR environment and take another look before giving their answers.
4. After the four scenarios, users were asked to choose their favourite method, and the second best.

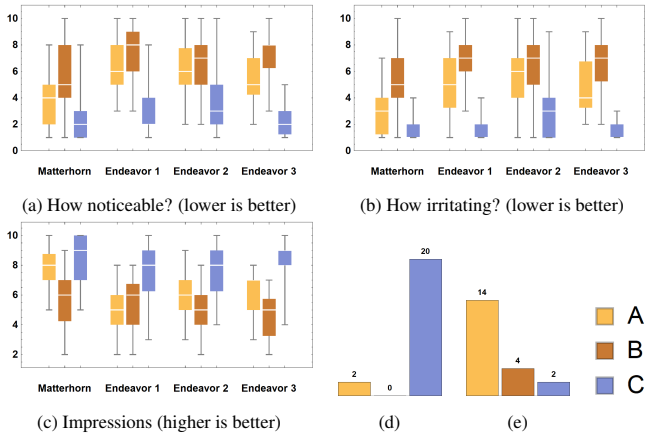
Our hypotheses were:

1. $H1_A$ and $H1_B$: Method C gives a better impression of the point cloud than methods A and B.
2. $H2_A$ and $H2_B$: Method C has less noticeable changes in the level of detail than methods A and B.
3. $H3_A$ and $H3_B$: Method C has less irritating changes in the level of detail than methods A and B.

We tested our hypotheses using the Mann-Whitney U test as it is non-parametric and does not assume a normal distribution, and compared the p-values with a significance level of $\alpha = 0.005$. 22 out of 24 combinations of scenarios and hypotheses have shown a significant improvement of our method ($p < 0.0037$). The non-significant cases are $H1_A$ in the Matterhorn scenario (p = 0.152; Fig. 8c), and $H2_A$ in the Matterhorn scenario (p = 0.012; Fig. 8a), i.e. method C does not leave a statistically significantly better impression and LOD is not significantly less noticeable compared to method A in the Matterhorn scenario. This is not unexpected since the Matterhorn data set has a low geometric complexity and mostly low-frequency color variations that do not pose a challenge for any of the evaluated methods. However, our method C shows significant improvements over A and B in all scenarios of the Endeavor data set, because it has a high geometric complexity that state-of-the art DLOD methods do not handle well.

Apart from that, 20 out of 23 participants rated our method the preferred one. One user did not name a preferred method, and three users did not name a runner-up.

We would also like to discuss one notable result. Fig. 8a and Fig. 8b show that our method was more noticeable and irritating in scenario 3, compared to its results in the other three scenarios. We believe that this is largely caused by the inhomogeneous and noisy colors of the points in this data set, combined with a higher level of detail and point sizes as opposed to scenario 2. The point cloud is a mix of laser scans from many different positions, each colored by photographs at the respective location. Because of the varying light exposure at different positions, some scans ended up darker than others, even in overlapping regions. As a consequence, overlapping regions are represented by bright to white points from one scan, and dark to black points from another scan. When a user navigates towards such a region, black points would suddenly appear in a previously mostly grey region. This is less of an issue at lower levels of detail where colors are averaged over the area they represent.



(a) How noticeable? (lower is better) (b) How irritating? (lower is better)

(c) Impressions (higher is better) (d) (e)

Figure 8: Box plots of the users' ratings for methods A, B, and our method, C. (d) Overall preferred method. (e) Runner-up.

6 Conclusions and Future Work

We presented a method to create and render point clouds with a continuous level of detail that is fast enough for VR applications. The continuous LOD results in subtle and less irritating changes of detail as users move through the scene, and therefore improves the VR experience over discrete LOD methods. This is achieved through three main contributions: first, a compute shader that iterates over the point cloud and selects points according to a target spacing; second, a randomization of point LODs to avoid harsh changes between levels of detail, and third, a transition period where the size of new points is increased smoothly. A user study has confirmed that users prefer our method to state-of-the-art point-rendering methods.

We were able to implement our CLOD method without hierarchical traversal while matching the detail and performance of an octree-based DLOD method. The downside is that this currently limits our method to point clouds that fit in GPU memory, and also increases the number of frames it takes to compute a new down-sampled version of the point cloud with the number of input points. However, we believe that CLOD can be implemented in a similar hierarchical and out-of-core fashion as state-of-the-art DLOD methods by applying the reduction step to the nodes of the DLOD structure and discard points the same way we do now. In the future, we would like to evaluate optimal ways to realize this in order to achieve continuous level of detail for arbitrarily large point clouds.

Z-fighting issues are currently augmented by repeatedly recreating a new vertex buffer. Normally, repeatedly rendering two overlapping triangles at the same depth would return the same result as long as they are rendered in the same order and if they are projected with the same matrices, i.e., if there is no motion. Since the order of points inside the down-sampled model is unpredictable and changes with each iteration, so does the order in which they are drawn by the GPU, which results in constant z-fighting, even if there is no motion. We currently don't address this issue other than by carefully selecting the near clip plane.

We currently reduce the point density in the periphery mainly to account for the distortion that is applied to the rendered image before it is displayed in an HMD. The same principle could likely be used for foveated rendering in order to adjust the point density to the viewing direction of the user's eyes. A potential issue could be that our method requires five to six frames to compute a new reduced vertex buffer, which may or may not be too slow for quick eye movements.

Rendering performance of our method is currently harder to predict in advance compared to DLOD methods, because it is fidelity-based rather than budget-based. The same quality settings may result

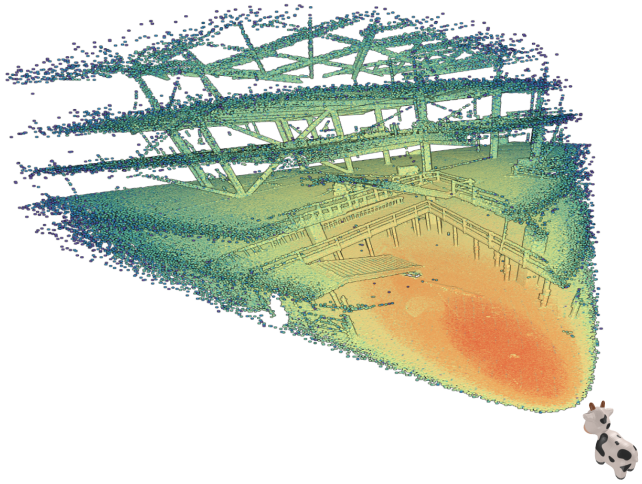


Figure 9: Third-person view of a CLOD subsample selected for the user's current viewpoint.

in 1 million points in one viewpoint, and 3 million points in another. However, adapting the level of detail to the performance of previous frames may be less noticeable than with DLOD methods.

Acknowledgments

The authors wish to thank NVIDIA for providing the Endeavor data set, consisting of laser scans of the building site of their new headquarter, and Pix4D for providing a photogrammetry scan of the Matterhorn. This research was enabled by the Doctoral College Computational Design (DCCD) of the Center for Geometry and Computational Design (GCD) at TU Wien.

References

[1] Entwine. `https://entwine.io/`. Accessed 2018.11.24.

[2] Openvr. `https://github.com/ValveSoftware/openvr`. Accessed 2018.11.28.

[3] V8 javascript engine. `https://v8.dev/`. Accessed 2018.11.28.

[4] Endeavor - scan of the building site of nvidia's new headquarter. `https://www.nvidia.com`, April 2016. Accessed 2018.11.23.

[5] C. Boucheny. Visualisation scientifique de grands volumes de donnes : Pour une approche perceptive. 2009.

[6] R. L. Cook. Stochastic sampling in computer graphics. *ACM Trans. Graph.*, 5(1):51–72, Jan. 1986. doi: 10.1145/7529.8927

[7] C. Dachsbacher, C. Vogelgsang, and M. Stamminger. Sequential point trees. *ACM Transactions on Graphics*, 22:657, 07 2003. doi: 10.1145/1201775.882321

[8] S. Discher, L. Masopust, S. Schulz, R. Richter, and J. Döllner. A point-based and image-based multi-pass rendering technique for visualizing massive 3d point clouds in vr environments. In *2018 Journal of WSCG*, vol. 26, 06 2018. doi: 10.24132/JWSCG.2018.26.2.2

[9] J. Futterlieb, C. Teutsch, and D. Berndt. Smooth visualization of large point clouds. *IADIS International Journal on Computer Science and Information*, 2:146158, 2016.

[10] E. Gobbetti and F. Marton. Layered point clouds: A simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Comput. Graph.*, 28(6):815–826, Dec. 2004. doi: 10.1016/j.cag.2004.08.010

[11] P. Goswami, Y. Zhang, R. Pajarola, and E. Gobbetti. High quality interactive rendering of massive point models using multi-way kd-trees. In *2010 18th Pacific Conference on Computer Graphics and Applications*, pp. 93–100, Sept 2010. doi: 10.1109/PacificGraphics.2010.20

[12] H. Hoppe. Progressive meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pp. 99–108. ACM, New York, NY, USA, 1996. doi: 10.1145/237170.237216

[13] M. Kraemer. Accelerating your vr games with vrworks. NVIDIAs GPU Technology Conference (GTC), 2018.

[14] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner. Real-time, continuous level of detail rendering of height fields. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pp. 109–118. ACM, New York, NY, USA, 1996. doi: 10.1145/237170.237217

[15] D. Luebke, M. Reddy, J. D. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[16] O. Martinez Rubi, S. Verhoeven, M. van Meersbergen, M. Schütz, P. Oosterom, R. Goncalves, and T. Tijssen. Taming the beast: Free and open-source massive point cloud web visualization. 11 2015. doi: 10.13140/RG.2.1.1731.4326/1

[17] M. L. Muzic, L. Autin, J. Parulek, and I. Viola. cellview: a tool for illustrative and multi-scale rendering of large biomolecular datasets. In K. Bühler, L. Linsen, and N. W. John, eds., *Eurographics Workshop on Visual Computing for Biology and Medicine*, pp. 61–70. EG Digital Library, The Eurographics Association, Sept. 2015. doi: 10.2312/vcbm.20151209

[18] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface elements as rendering primitives. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pp. 335–342. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. doi: 10.1145/344779.344936

[19] S. Rusinkiewicz and M. 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, pp. 343–352. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. doi: 10.1145/344779.344940

[20] P. V. Sander and J. L. Mitchell. Progressive buffers: View-dependent geometry and texture lod rendering. In *ACM SIGGRAPH 2006 Courses*, SIGGRAPH '06, pp. 1–18. ACM, New York, NY, USA, 2006. doi: 10.1145/1185657.1185826

[21] C. Scheiblauer. *Interactions with Gigantic Point Clouds*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, 2014.

[22] D. Scherzer and M. Wimmer. Frame sequential interpolation for discrete level-of-detail rendering. *Computer Graphics Forum (Proceedings EGSR 2008)*, 27(4):1175–1181, June 2008. doi: 10.1111/j.1467-8659.2008.01255.x

[23] J. Schneider and R. Westermann. Gpu-friendly high-quality terrain rendering. *Journal of WSCG*, 14(1-3):49–56, 2006.

[24] M. Schütz. Potree. `https://github.com/potree/potree`. Accessed 2018.11.26.

[25] M. Schütz. Potree: Rendering large point clouds in web browsers. Master's thesis, TU Wien, Austria, 2016.

[26] A. Vlachos. Advanced vr rendering. Game Developers Conference, industry talk, March 2015. Accessed 2018.11.20.

[27] M. Wand, A. Berner, M. Bokeloh, A. Fleck, M. Hoffmann, P. Jenke, B. Maier, D. Staneker, and A. Schilling. Interactive editing of large point clouds. In *SPBG*, 2007.

[28] M. Wimmer and C. Scheiblauer. Instant points: Fast rendering of unprocessed point clouds. In *Proceedings Symposium on Point-Based Graphics 2006*, pp. 129–136. Eurographics, Eurographics Association, July 2006. doi: 10.2312/SPBG/SPBG06/129-136