DOI: 10.1111/cgf.14345
Eurographics Symposium on Rendering 2021
A. Bousseau and M. McGuire
(Guest Editors)

Volume 40 (2021), Number 4

Rendering Point Clouds with Compute Shaders and Vertex Order Optimization

Markus Schütz, Bernhard Kerbl, Michael Wimmer

TU Wien

Figure 1: *Performance of `GL_POINTS` compared to a compute shader that performs local reduction and and early-z testing, and a high-quality compute shader that blends overlapping points. Retz point cloud (145 million points) courtesy of Riegl.*

Abstract

In this paper, we present several compute-based point cloud rendering approaches that outperform the hardware pipeline by up to an order of magnitude and achieve significantly better frame times than previous compute-based methods. Beyond basic closest-point rendering, we also introduce a fast, high-quality variant to reduce aliasing. We present and evaluate several variants of our proposed methods with different flavors of optimization, in order to ensure their applicability and achieve optimal performance on a range of platforms and architectures with varying support for novel GPU hardware features. During our experiments, the observed peak performance was reached rendering 796 million points (12.7GB) at rates of 62 to 64 frames per second (50 billion points per second, 802GB/s) on an RTX 3090 without the use of level-of-detail structures.

We further introduce an optimized vertex order for point clouds to boost the efficiency of GL_POINTS by a factor of 5× in cases where hardware rendering is compulsory. We compare different orderings and show that Morton sorted buffers are faster for some viewpoints, while shuffled vertex buffers are faster in others. In contrast, combining both approaches by first sorting according to Morton-code and shuffling the resulting sequence in batches of 128 points leads to a vertex buffer layout with high rendering performance and low sensitivity to viewpoint changes.

CCS Concepts

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

1. Introduction

Point clouds are three-dimensional models that consist of individual points with no connectivity. They are typically obtained by reconstructing the real world, for example with laser scanners or photogrammetry. Laser scanners obtain point-sampled surface models by measuring the distance from the scanner to surrounding surfaces, and then transforming the distance values and the known orientation of the scanner into 3D coordinates. Photogrammetry uses multiple photos to create a 3D model that best fits these images. The amount of points produced by these methods ranges from a few million up to trillions of points, depending on the scan resolution and the extent of the scanned area. The *Actueel Hoogtebestand Nederland* (AHN2) [AHN2] data set, for example, is an aerial laser

scan of the entire Netherlands, comprising 640 billion points, for a total of 1.6 terabytes in compressed form [MVvM*15]. Terrestrial laser scans (e.g., of buildings, monuments, caves, smaller regions) usually yield from tens of millions up to a few billion points.

The rendering of large point clouds is an active field of research that includes topics such as high-quality rendering, as well as the generation and effective use of level-of-detail (LOD) structures. Arguably, the most widespread solution for rendering a given set of points is by using the native point primitive that modern graphics APIs provide in addition to lines and triangles, such as `GL_POINTS` for OpenGL and WebGL, `D3D11_PRIMITIVE_TOPOLOGY_POINTLIST` for DirectX, `VK_PRIMITIVE_TOPOLOGY_POINT_LIST` for Vulkan and "point-list" for WebGPU. A key property of point primitives is that they require only one single coordinate vector as input, thereby keeping vertex buffer usage and bandwidth requirements to a minimum. OpenGL and Vulkan enable developers to specify the pixel size of points inside the vertex shader (rasterizing them as rectangles), whereas DirectX only supports a size of 1 pixel per point. Rendering larger points in DirectX and DirectX-based solutions (e.g., WebGL, WebGPU) requires developers to emulate sized points. In case of WebGL, this is done by instancing a quad model at each point location, as seen in the backend ANGLE [ANGLE]. `GL_POINTS` is the standard high-performance point rendering primitive in research (e.g., [RDD15; Ric18; Mor17; GM04; NMN20; SP04; DDG*04; MB20]) and software (e.g., CloudCompare [CC], Potree [POT], QGIS [QGIS], Cesium [Cesium]). It is also used in high-quality techniques that rely on quads with specific pixel sizes [GP03; SP11], as well as impostors [MB20] and particle systems [Lea14]. However, in this paper, we will only consider cases where each point is drawn to at most one pixel of the framebuffer.

Modern graphics processing units (GPUs) are capable of rendering several million points (smartphones, integrated graphics) up to around 100 million points (high-end GPUs) in real-time (60 fps) using `GL_POINTS` or its counterparts. Rendering arbitrarily large point clouds, however, requires hierarchical level-of-detail (LOD) structures that only load and render a small subset of the full model at any given time. Unfortunately, generating these structures requires time-consuming preprocessing steps that impede quick inspection of larger scenes. In this paper, we focus on raising the raw rendering performance of given point cloud data sets by exploiting the GPU compute pipeline, as well as low-overhead vertex reordering schemes. Based on our compute-based variants, we also propose an easy-to-implement, high-quality point cloud rendering method that can reduce aliasing while achieving higher frame rates than the hardware pipeline. Since different hardware and graphics APIs impose diverse restrictions on the features that are exposed by compute shaders, we explore several variations of our compute-based solutions that exploit different feature levels to heed these limitations. For instance, developers that target the soon-to-be-released WebGPU standard may adopt our basic variants, while users of lower-level graphics APIs can exploit hardware-accelerated primitives for peak performance.

Hence, our contributions to the state of the art are:

- Suitable alternatives to *glDrawArrays(*`GL_POINTS`*, ...)*, based on compute shaders that use 64bit atomic operations to draw into an interleaved depth and color buffer.
- Performance improvements and extensions to the basic compute shader with early-z (as suggested in [GKLR13]) and group-wide reduction of points with warp-level primitives to reduce contention in global GPU memory.
- A high-quality shader that provides anti-aliasing within pixels by blending overlapping points together (similar to mipmapping), while still being faster than the aliased results of `GL_POINTS`. The basic version, *HQS*, is a simplified and compute-based implementation of the blending algorithm in [BHZK05], while the *HQS1R* version reduces memory accesses by updating the sum of colors and fragment counts with a single atomic instruction per point.
- An easy-to-implement policy for rearranging points in a shuffled Morton order that significantly improves stability and rendering performance with `GL_POINTS`.
- A thorough evaluation to quantify the impact of using different compute shader techniques, point orderings and GPUs for rendering a range of point-cloud data sets, both unstructured and embedded in LOD data structures.

2. Related Work

2.1. Compute-based Triangle Rasterization

Although the GPU's rendering pipeline is becoming increasingly programmable, its implementation in hardware necessitates numerous restrictions. The general-purpose capabilities of modern GPUs have led developers to pursue parallel software rasterizers as an alternative. Freepipe [LHLW10] describes one of the first practical software implementations running on a GPU, using a single thread per triangle. The authors report that Freepipe surpasses the performance of OpenGL-based solutions in carefully selected applications. Later approaches managed to achieve a higher level of parallelism and significantly improved performance via tile-based rendering and coverage masks [PTSO15; LK11]. Complete sort-middle streaming pipelines have been realized in both CUDA and OpenCL [KKSS18; KB21]. Recently, Unreal Engine presented a hybrid approach that renders small, about pixel-sized, triangles with a compute-based software rasterizer. They report that software rasterization is, on average, three times faster for small triangles, while large triangles are still passed on to the standard hardware rasterizer [Nanite].

2.2. Compute-based Point Cloud Rendering

Günther et al. [GKLR13] were the first to suggest compute-based point rendering as an alternative to the standard point primitives of OpenGL. Their OpenCL implementation uses a busy-loop to wait until a pixel is free to write to, locks the pixel with an atomic compare-and-swap, updates the depth and color buffers, and afterward unlocks the pixel again. The authors observed that adding a custom early-z test enables the compute-based approach to scale significantly better with the number of fragments per pixel than

GL_POINTS. Lukac et al. [Lpa14] use a very similar *atomicMin*-based algorithm in their hybrid point and volume renderer. However, their method is not thread-safe and leads to artifacts with dense point clouds. Instead of a busy-loop, our method updates an interleaved depth and color buffer with a single atomic instruction. Marrs et al. [MWH18] use compute-based point-cloud rendering as a means to reproject a depth map to several different views. Since no colors are required, they simply use *InterlockedMax* (DirectX's counterpart to *atomicMax*) to write the largest depth into the target buffer. In contrast, our approach writes depth and color information and exploits local reduction to maximize performance.

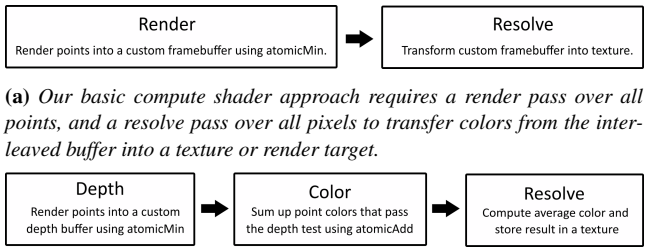
2.3. Hole Filling

The approaches presented in this paper only address rendering at most one pixel per point, which may lead to holes if the point density is not high enough from a given view of the scene. However, several hole-filling algorithms have been proposed to eliminate the gaps between points of a point cloud—many of them via fast screen-space methods that operate by analyzing the neighborhood of a pixel.

Grossman and Dally [GD98], Pintus et al. [PGA11] and Marroquim et al. [MKC07] first draw points and then build an image pyramid (e.g., [SKE06]) that is subsequently used to fill in gaps. This involves determining hidden surfaces that were rendered through gaps between points of the front-most visible surface. Afterward, background pixels and points that are determined to be hidden are replaced with colors from the image pyramid. Instead of an image pyramid, Rosenthal and Linsen [RL08] use an iterative approach that covers wider areas with higher iteration counts, but often already succeeds with fewer iterations. In contrast to methods based on computing color values between the gaps of rendered pixels, Auto Splats [PJW12] takes an additional step to compute oriented and sized splats, based on the initial rendering, and draws these splats in a separate rendering pass. The resulting splats contain normals that can be used for shading, and their size is adapted to fill in the gaps.

2.4. High-Quality Rendering

In addition to visible holes, rendering point clouds is prone to aliasing artifacts due to the lack of mipmapping. Arikan et al. [APW16] create multiple textured depth maps out of point clouds, and then ray-trace the depth maps in order to obtain a high-quality reconstruction of the underlying point cloud data. Surface splatting [ZPVG01] is an extension of point clouds that uses oriented disks as a better representation of the underlying surface. The splat sizes are adjusted to cover holes, and overlapping splats are blended together, which results in anti-aliasing similar to mipmapping. Botsch et al. [BHZK05] implement an efficient surface-splatting method that uses a depth pass to generate a shifted depth buffer, an attribute pass to sum up weighted fragments that pass the depth test, and a normalization pass to divide the weighted sum by the sum of weights. Our work also presents a high-quality shader based on blending overlapping points, but without considering oriented disks with variable radii. The already discussed Auto Splats method [PJW12] uses the same high-quality rendering method, but computes the required attributes (normals and radii) on the fly.



(a) *Our basic compute shader approach requires a render pass over all points, and a resolve pass over all pixels to transfer colors from the interleaved buffer into a texture or render target.*

(b) *The high-quality approaches require two geometry passes (depth and color) over all points, and a resolve pass over all pixels that writes blended color values into a displayable texture.*

Figure 2: *Proposed compute shader pipelines.*

2.5. Level-of-Detail Rendering

While our compute-based rendering method attempts to speed up the rendering of a given set of points, LOD methods aim to improve the performance by reducing the number of points being rendered. These goals are complementary and can be combined to further improve the overall point-cloud rendering performance. An LOD approach that is built on GL_POINTS may benefit from faster compute-based alternatives. LODs are addressed via sequential point trees [DVS03], layered point clouds [GM04] and their variations [WBB*08; GZPG10; SW11; EBN13; MVvM*15; KJWX19; BK20]. We assess layered point cloud rendering in Section 5.4.

3. Compute-Shader Rendering

In this chapter, we first describe a basic compute shader-based method for point cloud rendering, followed by various optimizations to improve performance and quality. A detailed evaluation of each variation is provided in Section 5.

Figure 2 shows the individual compute-shader passes for the basic and high-quality approaches. All proposed variations require at least one render pass over all points, and a resolve pass over all pixels. The render pass transforms points to screen space and writes them into a storage buffer that acts as a framebuffer. The resolve pass transfers the results from our custom framebuffer into an OpenGL texture. The high-quality variations further require an additional pass over all points to compute a depth map, for a total of two geometry passes and one screen pass.

3.1. Basic Approach

Our compute-based approach resolves visibility and color simultaneously by encoding the depth and color in a single 64bit integer, and uses 64bit *atomicMin* to pick points with the lowest depth, as shown in the following GLSL sample:

```
1 vec4 pos = worldViewProj * position;
2
3 int pixelID = toPixelID(pos);
4 int64_t depth = floatBitsToInt(pos.w);
5 int64_t point = (depth << 24) | rgb;
6
7 atomicMin(framebuffer[pixelID], point);
```

Listing 1: *Render pass draws closest points via atomicMin.*

The compute shader replaces the role of both vertex and fragment shader in the standard rendering pipeline. In Listing 1, we only write the vertex color, stored in variable *rgb*, to the framebuffer, as is common when rendering point clouds. Alternatively, we may also compute illumination if normals are present, or any other mapping from attributes to RGB color values. The target of the render pass is an OpenGL storage buffer containing one 64bit integer per pixel. For each point, the color value is stored in the 24 least significant bits of the integer (8 bits per color channel), and the 32bit depth value is stored in the 32 next-higher bits. Using *atomicMin*, we can then make sure that the point with the smallest depth value is chosen for each pixel. After the render pass, we transfer the results from this storage buffer into an OpenGL texture with a resolve pass over all pixels:

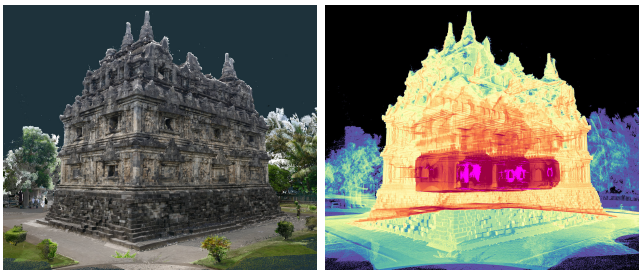
```
uint pixelID = x + y * imageSize.x;
// Read RGB component from framebuffer
uvec4 icolor = colorAt(pixelID);
// Write result into an OpenGL texture
imageStore(texture, ivec2(x, y), icolor);
```

Listing 2: *Resolving custom framebuffer to OpenGL texture.*

One beneficial side effect of this approach is the ability to directly use the linear depth value (*pos.w*) for depth-buffering. In contrast, depth values in the conventional hardware rendering pipeline are subject to further transformations and associated loss of precision. In order to allow perspectively correct triangle rasterization, the typically used transformation matrices result in hyperbolic depth buffer values which, when combined with floating-point data types, causes a disproportionately larger accumulation of precision towards the near plane. We note that methods such as reverse-z [DPV18; LJ99] can help achieve a near-optimal precision of the hardware pipeline by mapping the far plane to 0 and the near plane to 1. Nevertheless, since we don't require interpolation across pixels, we can directly use the unmodified linear depth value to obtain the maximum depth-buffer precision that a typical floating point transformation matrix may achieve. If traditional depth-buffer values are required (e.g., for hybrid rendering with triangles), developers may integrate associated conversions in the resolve pass.

3.2. Early Depth Test

Rendering hundreds of millions of points to framebuffers with approximately 2 million target pixels (1920x1080) can lead to the projection of several thousand points to a single pixel, depending on viewpoint and scan density distribution, as shown in Figure 3. Such a large number of atomic operations on the same memory location causes contention with severe negative impacts on the performance. In practice, this manifests as lower framerates after zooming out, even if the number of points within the view frustum stays the same. In the traditional rendering pipeline, early-z (also known as early depth test or early fragment test) is done before the fragment shader to avoid calling the shader program for fragments that are occluded by previously processed fragments. In our compute shader-based approach, we can achieve early depth testing by comparing against the current value at the storage buffer location we are trying to write to. *atomicMin* only needs to be called if the currently processed point has a smaller depth than a previously written point. A GLSL snippet for early depth testing is given in Listing 3:



(a) *RGB* **(b)** *Points per pixel*

Figure 3: *(a) Candi Sari point cloud. (b) Heatmap of the number of points in each pixel. 9,427 pixels contain over 10k points (in pink). Points: 725M. Resolution: 1920x1080. Candi Sari point cloud courtesy of TU Wien – Baugeschichte.*

```
// Fetch 64bit value from interleaved buffer
uint64_t oldPoint = framebuffer[pixelID];

// Higher bits encode depth, can compare directly
if(point < oldPoint)
{
    atomicMin(framebuffer[pixelID], point);
}
```

Listing 3: *Avoiding atomicMin calls with an early depth test.*

In contrast to [GKLR13], our simplified solution exploits the fact that previously written depth values may already be available for reading in fast L1 caches. Although loading and evaluating the current depth is not synchronized, this does not affect the rendered image because the real depth buffer value can only become smaller in the meantime. In the worst case, we merely invoke superfluous *atomicMin* calls.

3.3. Local Reduction with Warp-Level Primitives

Individual GPU threads are grouped in concurrently scheduled warps (NVIDIA), wavefronts (AMD) or subgroups (OpenGL). On NVIDIA microarchitectures, each warp consists of 32 threads that can operate based on the SIMT (single instruction, multiple threads) principle. Threads within each warp can communicate efficiently, opening up the possibility to combine intermediate results. In our case, we can combine up to 32 points into a single framebuffer update if they fall inside the same pixel, thereby reducing the amount of expensive *atomicMin* calls from 32 to just one.

In Listing 4, all threads in the subgroup first check if they have the same target pixel ID. If this is the case, the threads compute the minimum depth value of the whole subgroup and only the thread with the smallest depth proceeds to write a point to the framebuffer. The final `if` clause applies to both the slow and the fast, reduction-based path, causing 32 atomic updates in the former and a single update in the latter.

```
int minDepth = depth;

if(subgroupAllEqual(pixelID))
  minDepth = subgroupMin(depth);

// Different pixels or thread has lowest depth
if(minDepth == depth)
  atomicMin(framebuffer[pixelID], point);
```

Listing 4: *If all points in a warp fall into the same pixel, atomicMin is only called in the thread with the lowest depth. Otherwise, atomicMin must be called by all threads.*

The warp-wide reduction mostly affects very dense point clouds and zoomed-out views. In order to accelerate other cases as well, we can perform a quick, fine-granular local reduction where pairs of threads check whether their updates can be merged, as outlined by Listing 5. Doing so can help to reduce the number of atomic operations, even for close-up viewpoints, depending on the order of vertices in memory.

```
uint idXOr = subgroupClusteredXor(pixelID, 2);
uint minDepth = subgroupClusteredMin(depth, 2);

// Update if different pixels or lowest depth
if(idXOr != 0 || minDepth == depth)
  atomicMin(framebuffer[pixelID], point);
```

Listing 5: *If two neighboring threads target the same pixel, only the one with the lower depth value will write its result.*

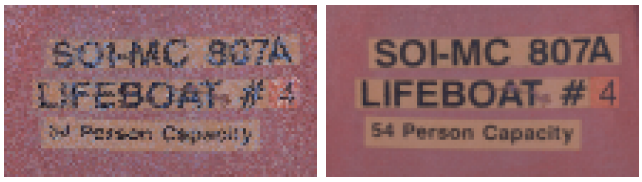
3.4. Full Warp-Wide Deduplication

Taking advantage of recent GLSL language extensions, we can fully deduplicate the pixel updates within a subgroup, i.e., we can further reduce framebuffer updates to a single *atomicMin* per accessed pixel, as shown in Listing 6.

```
// Find subgroup (sg) indices with same pixelID
uvec4 sg = subgroupPartitionNV(pixelID);

// Lowest depth among threads with same pixelID
int minDepth;
minDepth = subgroupPartitionedMinNV(depth, sg);
// Write if thread owns point with lowest depth
if(depth == minDepth)
  atomicMin(framebuffer[pixelID], point);
```

Listing 6: *The deduplication shader can reduce pixel updates to only one per accessed pixel in each SIMT subgroup.*

Each thread first requests a mask of all threads that intend to write to the same pixel. This mask is then used to compute the minimum depth of the respective threads, and only the threads with the lowest depth will update the pixel. Note that there may still be superfluous *atomicMin* calls if two or more points in a warp have the same pixel ID and depth value. While it is possible to remove these duplicates with additional warp-wide logic operations, but in practice, we found no discernible benefits from doing so.



(a) *Basic Shaders* **(b)** *HQS Shaders*

Figure 4: *(a) `GL_POINTS` and our basic atomicMin-based approaches are prone to aliasing artifacts. (b) High-quality shading blends overlapping points together, improving quality and legibility of high-frequency features, such as text. Lifeboat point cloud courtesy of Weiss AG.*

3.5. High-Quality Rendering

While the presented approaches are fast and require only one pass over the entire geometry, they, like all single-pixel update methods, exhibit noticeable aliasing. Point-cloud renderings are colored per-vertex, which causes artifacts similar to rendering textured models without mipmapping. Mipmapping addresses the cases where the bounds of a single pixel map to a large area of a texture. Instead of picking a single texture element (texel), all texels contained in a pixel should contribute to that pixel. Mipmaps store precomputed averages of potentially contributing texels to avoid recomputing averages of potentially thousands of texels in each frame. Unfortunately, mipmapping is not feasible in our case: two-dimensional textures are not applicable to regular point-cloud data sets due to the lack of 2D neighborhood information. Instead, we propose a compute shader-based high-quality approach to calculate the average of overlapping points inside a pixel directly at runtime.

One concept for high-quality shading is that all overlapping points of the front-most visible surface should contribute to the pixel color, rather than just the point that is closest to the camera. Botsch et al. [BHZK05] achieve this through a multi-pass approach that first computes a depth map, which is then shifted slightly towards the far plane. They then sum up the colors of fragments that pass the depth test, and finally obtain the average by dividing the sum of all colors by the number of fragments that contributed to the sum. Figures 1 and 4 illustrate the difference between basic approaches, including `GL_POINTS`, and such a high-quality shading (HQS) approach. HQS reduces image noise by computing a blend of all relevant samples, and increases the fidelity of high-frequency features that might otherwise be lost. In order to apply this method on the GPU, we employ two geometry render passes—a depth and a color pass—over all points. The depth pass is implemented similar to the previously introduced *atomicMin* approaches, but without the need to update colors. We therefore implement it with simple 32bit atomic operations instead of 64bit.

In the color pass, we replace *atomicMin* by *atomicAdd* in order to sum up the values of all contributing points and their count. Contributing points should be those that belong to the front-most visible surface. However, since there is no notion of a surface in a point cloud, we instead propose considering all points within an ε range, proportional to the distance of the closest point. In practice, we obtained good results with an ε of 1%. To provide sufficient

capacity for computing the sum of a large amount of points, we recommend a buffer with 32 bits per color channel, and a fourth channel for the fragment count. Summing up all channels requires four 32bit *atomicAdd* calls, but we suggest packing the channels into two 64bit integers to reduce the number of *atomicAdd* updates to two per point, as shown in Listing 7. The resolve pass then divides the color sums by the total number of points that contributed to them, as outlined by Listing 8.

```
uint bufferVal = ssDepthbuffer[pixelID];
float bufferDepth = uintBitsToFloat(bufferVal);
float depth = pos.w;

if(depth <= bufferDepth * 1.01){
  int64_t rg = (r << 32) | g;
  int64_t ba = (b << 32) | 1;

  atomicAdd(ssRG[pixelID], rg);
  atomicAdd(ssBA[pixelID], ba);
}
```

Listing 7: *Accumulate colors and point count within a range* ε *proportional to the smallest depth (1% farther away, in this case).*

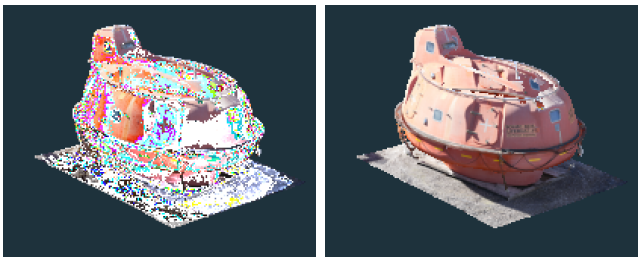
```
uint64_t rg = ssRG[pixelID];
uint64_t ba = ssBA[pixelID];

uint a = uint(ba & 0xFFFFFFFFUL);
uint r = uint((rg >> 32) / a);
uint g = uint((rg & 0xFFFFFFFFUL) / a);
uint b = uint((ba >> 32) / a);
```

Listing 8: *Normalize to the average color in the resolve pass.*

3.5.1. Single-Atomic High-Quality Variants

The number of atomic operations in the high-quality shader can be further reduced by encoding the RGB channels and counter into a 64bit integer. A distribution of 18 bits for the RGB channels and 10 bits for the counter guarantees that we can correctly compute the sum of up to 1023 points per pixel with a single 64bit atomic operation per point. Thus, we refer to this HQS variant as HQS1×.

While this naïve approach may work well for small point clouds and close-up views, the counter and color values overflow whenever more than 1023 points contribute to a pixel's average color value. For larger scenes, this leads to strongly noticeable rendering artifacts, as shown in Figure 5. Hence, we propose a robust alternative: instead of a 18-18-18-10 fragmentation, we use a 16-16-16-16 pattern. This ensures that up to 255 8bit color values can be stored in RGB, and the counter can register $2^{16} = 65\,536$ increments. We further modify the shader to check the return value after performing the 64bit atomic addition. If the count exceeds 255, the update is written a second time into separate fallback buffers with two atomic operations, as described in Section 3.5. The thread that detected an exact count of 255 is further tasked with transferring the pixel data to the fallback buffers and atomically setting the full 64bit value to zero. The unused $2^{16} - 256$ counts merely serve as a safety margin for—potentially several thousand—simultaneously executing threads to detect the overflow. If more or significantly fewer than 65k simultaneous accesses to a pixel are expected, the bit pattern



(a) *HQS (1×atomicAdd)* **(b)** *With overflow protection*

Figure 5: *(a) The 1×atomicAdd shader is prone to artifacts when a pixel receives over 1023 points. (b) The robust version adds overflow protection to eliminate these artifacts.*

may be changed accordingly. In practice, however, we found that this robust variant, *HQS1R*, with a 4×16 pattern shows negligible overhead compared to the unstable HQS1× version. Listing 9 shows the method's implementation. The associated resolve pass must then combine the results from all updated buffers.

```
uint64_t rgba = (r<<48)|(g<<32)|(b<<16)|1;
// Try rendering with just one atomicAdd
uint64_t old = atomicAdd(ssRGBA[pixelID],rgba);
uint count = uint(old & 0xFFFF);

if (count >= 255){
  // Overflow detected
  int a = 1;
  if (count == 255){
    // The first thread that overflows resets
    // the overflown buffer value to zero, and
    // transfers the r, g, b, a values before
    // the overflow to the fallback buffer.
    atomicExchange(ssRGBA[pixelID], 0);
    r += (old>>48) & 0xFFFF;
    g += (old>>32) & 0xFFFF;
    b += (old>>16) & 0xFFFF;
    a += 255;
  }
  int64_t rg = (r << 32) | g;
  int64_t ba = (b << 32) | a;
  // All overflown threads write the value
  // to the fallback buffer using 2 atomicAdd
  atomicAdd(fallback[2 * pixelID + 0], rg);
  atomicAdd(fallback[2 * pixelID + 1], ba);
}
```

Listing 9: *Summing up colors with a 16-16-16-16 bit pattern per point for the r-g-b-a channels and fallback on overflow.*

4. Vertex Order Optimization

Beyond various compute-based solutions for point rendering, we also investigate the impact of point order in memory. Reordering vertices and indices is a common task for mesh optimization prior to rendering and can significantly improve data access patterns. Similarly, point clouds can be stored in various different vertex orders. Since we use non-indexed draws, clearly the fetching of point-cloud data is already optimally coalesced. However, we

can also attempt to achieve a point ordering that positively impacts the updates of the framebuffer itself. One potential point ordering method is to shuffle them randomly. Shuffled or partially shuffled data sets can be used for progressive rendering [SMOW20] or as a simple level-of-detail structure, where rendering increasingly larger random subsets is akin to rendering increasingly higher levels of detail. Unfortunately, shuffling and the consequent absence of all data locality can be detrimental to rendering performance: Distributed writes to random framebuffer locations quickly lead to cache thrashing and fail to exploit the GPU's available memory bandwidth. A common method for increasing locality without spatial data structures is to sort by Morton code. In the resulting Z-curve, points that are stored side-by-side in memory are also likely to be close in 3D and in 2D after projection, thus favoring aggregate updates of close-by framebuffer locations.

While ordering points by Morton code may in fact be adequate for compute variants that write to buffers backed by linear memory, it is suboptimal for the hardware rendering pipeline: for parallel rasterization, all output geometry is sorted into 2D viewport tiles. On NVIDIA architectures, each tile is statically assigned to a graphics processing cluster (GPC) with exclusive access to the underlying memory. GPC tiles are spread over the viewport in repeating 2D patterns to facilitate uniform load balance. However, it has been shown that these static patterns cannot compensate for extreme spatial clustering of in-flight scene geometry [KKSS17]. To simultaneously address the hardware pipeline's demands for locality and sufficient utilization of all available GPCs, we thus propose an alternative layout that requires minimal overhead and is easy to produce: shuffled Morton order. First, all points are sorted according to Morton code. The resulting sequence is divided into batches of spatially sorted points. Batches are then moved to random locations without changing their internal order. The final sequence yields uniformly distributed groups of spatially close points. This new ordering accurately recreates the typical workload of the rasterization engine, i.e., geometry primitives that provoke an update for a localized set of fragments. In practice, we found that shuffled Morton order drastically improves performance when rendering with `GL_POINTS`. As an unexpected side effect, it also tends to reduce fluctuations and performance gaps across different compute-based methods (see following Section 5 for details).

5. Evaluation

In this section, we evaluate our proposed compute shader-based point cloud rendering approaches and compare them against each other, as well as adequate baselines under relevant aspects that can affect the performance of rendering large point clouds. The evaluated methods are:

- **GL_POINTS**: Hardware pipeline using OpenGL `GL_POINTS` primitive. Points are rendered with the default size of one pixel. The *gl_PointSize* variable is unmodified.
- **just-set**: A reference method with no program control overhead or performance penalties for memory synchronization. A compute shader draws points by setting the framebuffer's pixel colors in a non-atomic fashion. Not viable in practice, due to the lack of depth buffering and resulting flickering due to the unpredictable order in which points are written by multiple compute threads.
- **busy-loop**: Implementation of the busy-loop approach by Günther et al. [GKLR13], including their early-z test.
- **atomicMin**: Our basic compute-shader approach, which uses *atomicMin* to store the point with the smallest depth inside the framebuffer, as described in Section 3.1.
- **early-z**: The basic **atomicMin** method, but with an additional early-z test, as described in Section 3.2.
- **reduce**: The basic **atomicMin** method, but with an additional warp-wide reduction to decrease the number *atomicMin* calls from 32 to 1 if all points in the warp are inside the same pixel, or by up to 50% if pairs of adjacent threads write to the same pixel (outlined in Section 3.3).
- **reduce&early-z**: Applies both, early-z and reduce.
- **dedup**: A full deduplication shader, that reduces the amount of *atomicMin* calls in a subgroup to one per pixel (except if depth values are identical). See Section 3.4 for details. This method also includes an early-z test.
- **HQS**: *High-Quality Shading*. Computes average color of overlapping points in a certain depth range (Section 3.5).
- **HQS1R**: Robust high-quality shading with overflow protection using one *atomicAdd* in its fast path, and two atomic updates as fallback, as described in Section 3.5.1.

Furthermore, the order in which points are stored in the vertex buffer has a significant impact on rendering performance—up to an order of magnitude between differently sorted point clouds. The evaluated orderings are:

- **Original**: Points are kept in the order in which we originally received them. Typical orderings vary from sorted by Morton code, by scan position and timestamp, or even partially shuffled to simulate coarse LODs.
- **Morton**: The point cloud is sorted by a 21bit Morton code [MOT]—equivalent to sorting points using depth-first traversal into an octree with a depth of 21 levels.
- **Shuffled**: Points are shuffled by assigning a random value to each point, and then sorting according to that value.
- **Shuffled Morton**: Points are first sorted by Morton code, then grouped into batches of 128 points, and finally the batches are shuffled, with points inside each batch remaining in order. Doing so preserves basic locality between points within each batch, but avoids excessive locality that might lead to contention or imbalanced workload.

Performance for each method and ordering is measured using OpenGL timestamp queries at the start and end of each frame (right before *glfwSwapBuffers*). Reported performance numbers are computed as the average of all frames over one second. In the following, we compare our methods and relevant baselines with respect to the most significant aspects, based on our observations on four test systems:

- NVIDIA RTX 3090 24GB, AMD Ryzen 7 2700X (8 cores), 32GB RAM, running Microsoft Windows 10.
- NVIDIA RTX 2070 Super 8GB, Intel i7-4771 (8 cores), 16GB RAM, running Microsoft Windows 10.
- AMD Radeon RX Vega 64 8GB, Intel i7-4771 (8 cores), 16GB RAM, running Microsoft Windows 10.
- NVIDIA GTX 1060 3GB, AMD Ryzen 5 1600X (6 cores), 32GB RAM, running Microsoft Windows 10.

For the sake of brevity, we will focus on the illustration of results from the RTX 3090 and highlight differences where appropriate. Detailed benchmarks for the GTX 1060 and RTX 2070 are found in the paper's supplemental material. We would like to note that all compute-based benchmarks were evaluated on NVIDIA GPUs, since we only had access to an older AMD Radeon RX Vega 64 that does not expose 64bit atomic integer operations to GLSL. The results of our compute-based rendering approaches might therefore only be valid for NVIDIA GPUs.

5.1. Hardware vs. Compute Pipelines

Table 1 shows benchmark results comparing all evaluated methods for different data sets ranging from 4 million to 796 million points. The results clearly show that compute-based methods (including high-quality variants with additional anti-aliasing) are significantly faster than **GL_POINTS**, the predominantly used method in state-of-the-art solutions, by a factor of 10× or more. The improvements are least pronounced for the lion model due to the implied overhead caused by the additional resolve pass, which takes about 0.06ms on an RTX 3090, and compute pipeline launches in general. After investigating with Nsight Systems, we found that the *glDispatchCompute* call itself appears to have a larger overhead than *glDrawArrays*, which is significant for small data sets, but negligible when rendering larger scenes.

Among the individual compute-based methods, the **busy-loop** approach is often already outperformed by our most basic **atomicMin** and **reduce** methods, with the exception of particularly large or shuffled data sets. **early-z** can yield significant performance improvements (34% on average), but has adverse effects on rendering the smaller lion model. This is expected, because early-z testing only pays off once the depth buffer is partially filled by previously processed points and thus mostly affects larger scenes. Synergizing the properties of **reduce** and **early-z** in **reduce&early-z** yields a particularly strong and stable contender for all scenarios (1% faster than fastest of either **reduce** and **early-z**, 33% faster than slowest on average). It is outperformed only on the RTX 3090 by the **dedup** method, whose frame time is 1–2% faster on average and less affected by different vertex orderings. In contrast to the **busy-loop** approach, which thrives on shuffled vertex order, the performance of our compute-based methods is best when using sorted or partially sorted data sets. Specifically, the highest performance for every single scene was recorded on the RTX 3090, using **dedup** with Morton sorted scenes (at least 3% faster than the next better technique). **dedup** is most effective on the RTX 3090 and 2070, but clearly trails behind **reduce&early-z** on the GTX 1060. This comes as no surprise, since the *subgroupPartitionNV* and *subgroupPartitionedMinNV* primitives in GLSL are only accelerated by GPU hardware since the Volta and Ampere architectures, respectively. In contrast, **reduce&early-z** only requires warp shuffle instructions, which have been accelerated in hardware since the Kepler generation. For our high-quality variants, we found that **HQS1R** always runs faster than **HQS** (up to 37%, 18% on average). Both methods are up to 4× faster than the aliased **GL_POINTS** (see supplemental for visual comparison).

5.2. The Impact of the Viewpoint on Performance

A key issue when rendering large point clouds is the spatial clustering of fragments, which can lead to contention and poor occupancy. The viewpoint plays an important role in that aspect. Close-up viewpoints of a model are typically advantageous, because many points fall outside the view frustum, but also because the points themselves are distributed more evenly, so the chance of individual pixels containing a large number of points at once is low. However, when zooming out, points start to cluster inside an exceedingly smaller range of pixels, some of which may now receive tens of thousands of points, as shown in Figure 3. We observe that the performance of rendering vertices in their original or Morton order with **GL_POINTS** is strongly affected by the viewpoint. Rendering the *Retz* scene with a zoomed-in and zoomed-out view yields 31.95ms vs. 153.22ms (×4.79), respectively. The simple **busy-loop** and **atomicMin** approaches show similar trends (7.76ms vs. 16.66ms → ×2.15 and 5.07ms vs. 21.45ms → ×4.23), while sophisticated compute shader variants like **reduce&early-z** and **HQS1R** are mostly unaffected (3.37ms vs. 3.33ms and 6.91ms vs. 7.90ms, respectively). However, these figures are reversed when using shuffling. Fully shuffled data sets are less sensitive to viewpoint changes with **GL_POINTS** (30.71ms vs. 39.52ms → ×1.29), but increase view dependence and incur overall negative impact on the performance of our more elaborate compute-based methods, including high-quality shading variants. In opposition to these varied results, shuffled Morton order yields peak or close-to-peak performance over all rendering methods, regardless of viewpoint. However, the best results for a particular combination of rendering method and point ordering were observed with the **dedup** approach (3.06ms vs. 3.04ms) on Morton sorted data sets, showing a margin of just 3% between the close-up and overview scenario. A detailed illustration of exemplary viewpoint benchmarks with different levels of zoom on an RTX 3090 is provided as supplemental material.

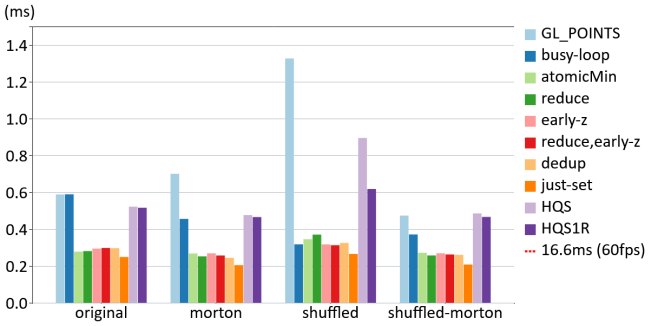
5.3. The Impact of Vertex Order on Performance

Table 1 also illustrates the impact of the four assessed vertex orders across different models and viewpoints. Overall, we find that elaborate compute-based methods perform better with Morton layouts, **GL_POINTS** and **busy-loop** perform better with shuffled points. Morton order predictably works well for methods that exploit locality, like **reduce** and **dedup** as well as close-up views with **GL_POINTS** and **atomicMin**. Shuffled points are favorable for zoomed-out views with **GL_POINTS** (3.9× over original) and **atomicMin** (2.1× over original). In contrast, shuffled Morton order is mostly view-independent and yields the highest performance for our compute-based methods in 65% of all scenarios, outperformed only occasionally by Morton order with **dedup** or **reduce&early-z** methods. It also raises the performance of high-quality shading by 7% on average.

The most drastic effects of choosing shuffled Morton, however, are observed for **GL_POINTS**: rearranging input vertices with the new layout accelerates rendering performance by up to 4× (2.25× on average) on an RTX 3090 and up to 2.5× (1.6× on average) on an RTX 2070, compared to the original ordering. Creating custom vertex orders with the described policies only requires a one-time preprocessing step, which may take from less than a milliseconds

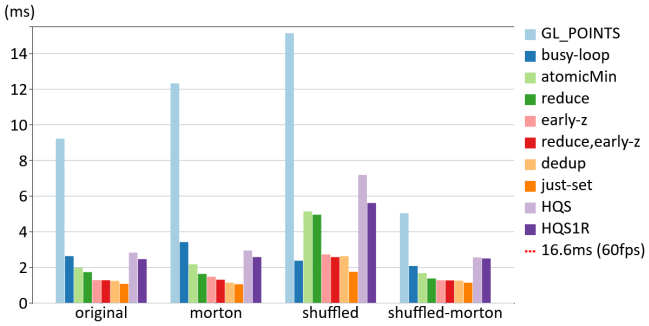
Model: Lion

Points: 4 million (64MB)



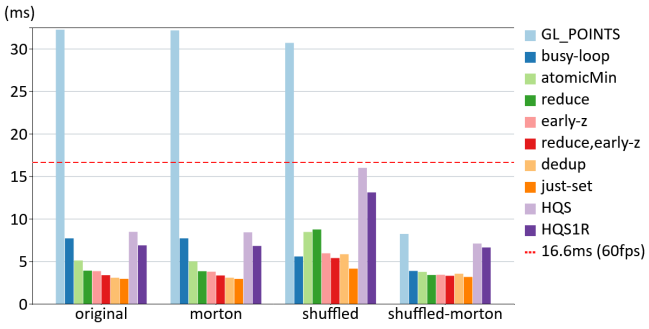
Model: Lifeboat

Points: 47 million (752MB)



Model: Retz

Points: 145 million (2.3GB)



Model: Endeavor

Points: 796 million (12.7GB)

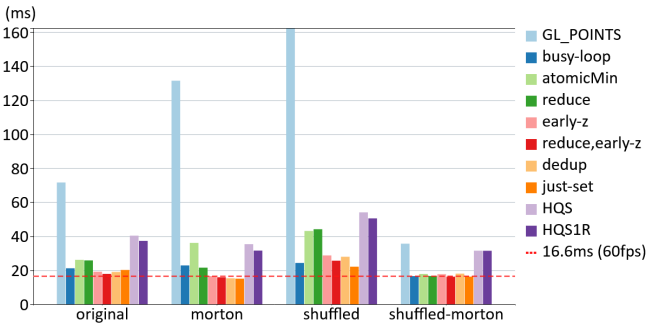


Table 1: *Frame times (ms) for a given viewpoint and vertex order on an RTX 3090 (lower is better).*

#nodes	points/node	GL_POINTS	compute
1624	2.4k	718 FPS	500 FPS
80	49.4k	1615 FPS	2096 FPS

Table 2: *Performance of rendering 4 million points in LOD structures with different granularity settings. Compute-based methods benefit from fewer nodes with a more points.*

to a few seconds, depending on the implementation and scene size. Hence, we claim that shuffled Morton is well-suited for the hardware pipeline and can be considered as a drop-in solution to accelerate applications based on `GL_POINTS`.

The results discussed above are specific to NVIDIA GPUs. The benchmark results on our AMD Radeon RX Vega are limited to rendering with `GL_POINTS`. We observed that the shuffled or shuffled Morton vertex orders are always the fastest vertex orders. Shuffled data sets rendered 1.4–5× faster than Morton ordered models, and shuffled Morton ordered data sets rendered 1.7–4.5× faster than Morton ordered models. For details, please see the supplemental material. Shuffled and shuffled Morton show very similar performance, with no clear winner on that GPU.

5.4. Level-of-Detail Performance

The *layered point-cloud* (LPC) [GM04] structure and its variations are a widely used LOD structure for point clouds (e.g., [QGIS; POT; ENT; A4D]) that stores subsamples of the full point cloud in the nodes of a spatial acceleration structure (e.g., binary tree, octree, kd-tree). Each node is essentially a small point cloud containing 100 to 20 000 points, and the union of all smaller point clouds yields the original. This way, the number of points that are rendered in each frame can be limited to approximately 1 to 10 million points, which corresponds to about 100 to 4 000 octree nodes. The points in each node are stored in separate vertex buffers, and in each frame the point cloud is rendered by invoking as many *glDrawArrays* calls as there are visible nodes.

In order to evaluate the suitability of compute-based rendering for layered point clouds, we use PotreeConverter 2.1 [PConv] to build an octree from the lion data set. With default granularity settings, PotreeConverter produces 1624 octree nodes with an average of 2.4k points per node. We also evaluate a modified case with lower granularity settings, which leads to an octree with 80 nodes and an average of 49.4k points in each node. Table 2 shows the performances obtained by rendering both octree data sets with **GL_POINTS** and **reduce&early-z** as a representative for compute-based methods. Clearly, **GL_POINTS** performs better when rendering a large number of nodes with a small amount of points in each, while compute performs better for fewer nodes with a higher number of points. According to these results, we claim that compute-based rendering is useful to LOD rendering approaches that ensure individual LOD chunks with at least several ten thousand points. Methods that pack all points into a single buffer [DVS03; SKW19] are very likely to benefit since compute approaches outperform **GL_POINTS** in all evaluated orderings.

5.5. Recommendations

Interpreting the results of our experiments (illustrated in Table 1, full data in supplemental material), our recommendations for vertex order and rendering method are as follows:

Recommended Vertex Order We recommend using the **shuffled Morton** order for the brute-force rendering of large point clouds, since it is typically the fastest or within small margins of the fastest ordering according to our benchmarks, regardless of the technique being used. Regular Morton order can achieve peak performance with advanced compute shader approaches on modern GPU hardware, but is vastly outperformed by shuffled Morton order when rendering point clouds using the `GL_POINTS` primitive.

Recommended Rendering Method The choice of the optimal rendering method is governed by the capabilities of the available hardware and graphics API. In our experiments, we observed peak performance results with the **dedup** approach in combination with Morton ordered points. However, these results strongly depend on recent GLSL extensions that may be unavailable for less specialized standards (e.g., WebGPU) and hardware-accelerated features of the NVIDIA Volta architecture or later. In contrast, the **reduce&early-z** approach may be implemented with vendor-agnostic extensions, performs well on the older Pascal architecture, and is consistently within slim performance margins of the **dedup** approach on newer models. We would therefore recommend using **reduce&early-z** for wider support, and consider **dedup** as an optional improvement on Volta, Turing or Ampere architectures, since **dedup** shows consistently high performance with the smallest fluctuations between different viewpoints.

6. Limitations

We would like to highlight two specific limitations of our approach and evaluation: First, the proposed compute-based rendering method requires at least 64bit atomic integer operations, which may not be available on older or mobile GPUs, and which became available in DirectX only recently. Second, the evaluation is limited to NVIDIA GPUs since the only AMD GPU at our disposal is relatively old and does not expose aforementioned 64bit integer atomics to GLSL shaders. The results might therefore not be representative for GPUs of all vendors.

However, at least the high-quality shading approach (Section 3.5) could also be implemented with only 32bit atomic operations. The depth-pass is already implemented using only 32bit *atomicMin*, and the color pass only uses 64bit *atomicAdd* as an optimization. Instead of two 64bit *atomicAdd* operations, colors and counters can also be summed up with four 32bit *atomicAdd* operations.

7. Conclusions

We have explored various methods for rendering point clouds with compute shaders, and shown that combining an *atomicMin*-based approach with early-z and warp-wide reduction or deduplication gives us the fastest results—up to an order of magnitude faster than using `GL_POINTS`. **dedup**, in particular, achieves consistently high performances in combination with Morton order, and

a throughput of up to 50 billion points per second (800GB/s) on an RTX 3090, which amounts to 85% of its specified memory bandwidth of 936GB/s. Additionally, a high-quality compute shader that computes the average of overlapping points was also shown to be faster than `GL_POINTS`, and results in a quality that is comparable to mipmapping or anisotropic filtering for textured meshes. `GL_POINTS` still outperforms compute when rendering a large number of small vertex buffers (up to a few thousands of points per buffer), but compute takes over when the buffers are larger (tens of thousands of points per buffer). From these results, we conclude that compute-based rendering with early-z and reduction enabled is a good choice for rendering unstructured point clouds with millions to hundreds of millions of points, and potentially useful for LOD rendering methods that employ larger LOD chunks.

We also investigated the impact of vertex order on rendering performance and found that Morton order and shuffled vertex buffers outperform each other in different scenarios. Combining both into a shuffled Morton vertex order preserves the advantages of both, and yields consistently high rendering performance over various different viewpoints.

In the future, we would like to further explore the application of compute shaders to the LOD rendering of point clouds. We believe that with sophisticated work aggregation schemes, compute shaders could prove advantageous for the fast and fine-grained selection and rendering of the most suitable LOD chunks, compared to the current layered point cloud applications that select a set of coarse chunks on the CPU side and then invoke one draw call for each.

8. Acknowledgements

The authors wish to thank *Riegl Laser Measurement Systems* for providing the data set of the town of Retz, *NVIDIA* for the Endeavor building site data set, *Weiss AG* for the lifeboat data set, and *TU Wien, Institute of History of Art, Building Archaeology and Restoration* for the Candi Sari data set.

This research has been funded by the FFG project *LargeClouds2BIM* and FWF project no. P32418, as well as the Research Cluster "Smart Communities and Technologies (Smart CT)" at TU Wien.

References

[A4D] *Arena4D*. https://veesus.com/veesus-arena4d-data-studio/, Accessed 2021.04.13 10.

[AHN2] *AHN2*. https://www.pdok.nl/introductie/-/article/actueel-hoogtebestand-nederland-ahn2-, Accessed 2021.03.27 1.

[ANGLE] *ANGLE: Emulating OpenGL Point Sprites in DirectX*. https://github.com/google/angle/blob/9b1c569b14e90765cdd7c07e449400e88f1d6c45/src/libANGLE/renderer/d3d/d3d11/StateManager11.cpp#L3904-L3913, Accessed 2021.03.19 2.

[APW16] ARIKAN, MURAT, PREINER, REINHOLD, and WIMMER, MICHAEL. "Multi-Depth-Map Raytracing for Efficient Large-Scene Reconstruction". *IEEE Transactions on Visualization & Computer Graphics* 22.2 (Feb. 2016), 1127–1137 3.

[BHZK05] BOTSCH, M., HORNUNG, A., ZWICKER, M., and KOBBELT, L. "High-quality surface splatting on today's GPUs". *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics, 2005.* 2005, 17–141 2, 3, 5.

[BK20] BORMANN, PASCAL and KRÄMER, MICHEL. "A System for Fast and Scalable Point Cloud Indexing Using Task Parallelism". *Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference*. Ed. by BIASOTTI, SILVIA, PINTUS, RUGGERO, and BERRETTI, STEFANO. The Eurographics Association, 2020 3.

[CC] *CloudCompare - 3D point cloud and mesh processing software*. https://www.danielgm.net/cc, Accessed 2019.07.01 2.

[Cesium] *Cesium*. https://github.com/CesiumGS/cesium/, Accessed 2021.03.19 2.

[DDG*04] DUGUET, FLORENT, DRETTAKIS, GEORGE, GIRARDEAU-MONTAUT, DANIEL, et al. "A Point-Based Approach for Capture, Display and Illustration of Very Complex Archeological Artefacts". *VAST 2004: The 5th International Symposium on Virtual Reality, Archaeology and Cultural Heritage*. Ed. by CHRYSANTHOU, Y., CAIN, K., SILBERMAN, N., and NICCOLUCCI, F. The Eurographics Association, 2004 2.

[DPV18] REED, NATHAN. *Depth Precision Visualized*. July 2015. URL: https://developer.nvidia.com/content/depth-precision-visualized 4.

[DVS03] DACHSBACHER, CARSTEN, VOGELGSANG, CHRISTIAN, and STAMMINGER, MARC. "Sequential Point Trees". *ACM Trans. Graph.* 22.3 (July 2003), 657–662 3, 10.

[EBN13] ELSEBERG, JAN, BORRMANN, DORIT, and NÜCHTER, ANDREAS. "One billion points in the cloud – an octree for efficient processing of 3D laser scans". *ISPRS Journal of Photogrammetry and Remote Sensing* 76 (2013). Terrestrial 3D modelling, 76–88 3.

[ENT] *Entwine*. https://entwine.io/, Accessed 2021.04.13 10.

[GD98] GROSSMAN, JEFFREY P and DALLY, WILLIAM J. "Point sample rendering". *Eurographics Workshop on Rendering Techniques*. Springer. 1998, 181–192 3.

[GKLR13] GÜNTHER, CHRISTIAN, KANZOK, THOMAS, LINSEN, LARS, and ROSENTHAL, PAUL. "A GPGPU-based Pipeline for Accelerated Rendering of Point Clouds". *J. WSCG* 21 (2013), 153–161 2, 4, 7.

[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 2, 3, 10.

[GP03] GUENNEBAUD, GAEL and PAULIN, MATHIAS. "Efficient screen space approach for Hardware Accelerated Surfel Rendering". *Vision, Modeling and Visualization*. Munich, Germany: IEEE Computer Society, Nov. 2003, 485–495 2.

[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*. 2010, 93–100 3.

[KB21] KIM, MINGYU and BAEK, NAKHOON. "A 3D graphics rendering pipeline implementation based on the openCL massively parallel processing". *The Journal of Supercomputing* (Jan. 2021) 2.

[KJWX19] KANG, L., JIANG, J., WEI, Y., and XIE, Y. "Efficient Randomized Hierarchy Construction for Interactive Visualization of Large Scale Point Clouds". *2019 IEEE Fourth International Conference on Data Science in Cyberspace (DSC)*. 2019, 593–597 3.

[KKSS17] KERBL, BERNHARD, KENZEL, MICHAEL, SCHMALSTIEG, DIETER, and STEINBERGER, MARKUS. "Effective Static Bin Patterns for Sort-Middle Rendering". *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*. Ed. by HAVRAN, VLASTIMIL and VAIYANATHAN, KARTHIK. ACM, 2017 7.

[KKSS18] KENZEL, MICHAEL, KERBL, BERNHARD, SCHMALSTIEG, DIETER, and STEINBERGER, MARKUS. "A High-performance Software Graphics Pipeline Architecture for the GPU". *ACM Trans. Graph.* 37.4 (July 2018), 140:1–140:15 2.

[Lea14] LEACH, CRAIG. "A GPU-Based Level of Detail System for the Real-Time Simulation and Rendering of Large-Scale Granular Terrain". MA thesis. University of Cape Town, 2014, 133 2.

[LHLW10] LIU, FANG, HUANG, MENG-CHENG, LIU, XUE-HUI, and WU, EN-HUA. "FreePipe: A Programmable Parallel Rendering Architecture for Efficient Multi-Fragment Effects". I3D '10. Washington, D.C.: Association for Computing Machinery, 2010, 75–82 2.

[LJ99] LAPIDOUS, EUGENE and JIAO, GUOFANG. "Optimal Depth Buffer for Low-Cost Graphics Hardware". *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*. HWWS '99. Los Angeles, California, USA: Association for Computing Machinery, 1999, 67–73 4.

[LK11] LAINE, SAMULI and KARRAS, TERO. "High-Performance Software Rasterization on GPUs". HPG '11. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2011, 79–88 2.

[Lpa14] LUKAC, N., PELIC, D., and ALIK, B. "Hybrid Visualization of Sparse Point-Based Data Using GPGPU". *2014 Second International Symposium on Computing and Networking*. 2014, 178–184 3.

[MB20] MICHEL, É. and BOUBEKEUR, T. "Real Time Multiscale Rendering of Dense Dynamic Stackings". *Computer Graphics Forum* 39.7 (2020), 169–179 2.

[MKC07] MARROQUIM, RICARDO, KRAUS, MARTIN, and CAVALCANTI, PAULO ROMA. "Efficient Point-Based Rendering Using Image Reconstruction". *Proceedings Symposium on Point-Based Graphics*. 2007, 101–108 3.

[Mor17] MOREL, JULES. "AN ANDROID APPLICATION TO VISUALIZE POINT CLOUDS AND MESHES IN VR". *CGVCVIP 2017*. 2017 2.

[MOT] *Morton*. https://www.forceflow.be/2013/10/07/morton-encodingdecoding-through-bit-interleaving-implementations/, Accessed 2021.03.19 7.

[MVvM*15] MARTINEZ-RUBI, OSCAR, VERHOEVEN, STEFAN, van MEERSBERGEN, M., et al. "Taming the beast: Free and open-source massive point cloud web visualization". Capturing Reality Forum 2015, Salzburg, Austria. Nov. 2015 2, 3.

[MWH18] MARRS, ADAM, WATSON, BENJAMIN, and HEALEY, CHRISTOPHER. "View-warped Multi-view Soft Shadows for Local Area Lights". *Journal of Computer Graphics Techniques (JCGT)* 7.3 (July 2018), 1–28 3.

[Nanite] *Nanite | Inside Unreal*. https://youtu.be/TMorJX3Nj6U?t=4063, Accessed 2021.06.16 2.

[NMN20] NYSJÖ, F., MALMBERG, F., and NYSTRÖM, I. "RayCaching: Amortized Isosurface Rendering for Virtual Reality". *Computer Graphics Forum* 39.1 (2020), 220–230 2.

[PConv] *Potree Converter*. https://github.com/potree/PotreeConverter/, Accessed 2021.03.19 10.

[PGA11] PINTUS, RUGGERO, GOBBETTI, ENRICO, and AGUS, MARCO. "Real-Time Rendering of Massive Unstructured Raw Point Clouds Using Screen-Space Operators". *Proceedings of the 12th International Conference on Virtual Reality, Archaeology and Cultural Heritage*. VAST'11. Prato, Italy: Eurographics Association, 2011, 105–112 3.

[PJW12] PREINER, REINHOLD, JESCHKE, STEFAN, and WIMMER, MICHAEL. "Auto Splats: Dynamic Point Cloud Visualization on the GPU". *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*. Ed. by CHILDS, H. and KUHLEN, T. Eurographics Association 2012. Cagliari, May 2012, 139–148 3.

[POT] *Potree*. http://potree.org, Accessed 2021.03.19 2, 10.

[PTSO15] PATNEY, ANJUL, TZENG, STANLEY, SEITZ, KERRY A., and OWENS, JOHN D. "Piko: A Framework for Authoring Programmable Graphics Pipelines". *ACM Trans. Graph.* 34.4 (July 2015) 2.

[QGIS] *QGIS: A Free and Open Source Geographic Information System*. https://qgis.org/, Accessed 2021.03.19 2, 10.

[RDD15] RICHTER, RICO, DISCHER, SÖREN, and DÖLLNER, JÜRGEN. "Out-of-Core Visualization of Classified 3D Point Clouds". *3D Geoinformation Science: The Selected Papers of the 3D GeoInfo 2014*. Ed. by BREUNIG, MARTIN, AL-DOORI, MULHIM, BUTWILOWSKI, EDGAR, et al. Cham: Springer International Publishing, 2015, 227–242 2.

[Ric18] RICHTER, RICO. "Concepts and techniques for processing and rendering of massive 3D point clouds". PhD thesis. Jan. 2018 2.

[RL08] ROSENTHAL, PAUL and LINSEN, LARS. "Image-space point cloud rendering". *Proceedings of Computer Graphics International*. 2008, 136–143 3.

[SKE06] STRENGERT, MAGNUS, KRAUS, MARTIN, and ERTL, THOMAS. "Pyramid methods in GPU-based image processing". *Proceedings vision, modeling, and visualization*. Vol. 2006. Citeseer. 2006, 169–176 3.

[SKW19] SCHÜTZ, MARKUS, KRÖSL, KATHARINA, and WIMMER, MICHAEL. "Real-Time Continuous Level of Detail Rendering of Point Clouds". *2019 IEEE Conference on Virtual Reality and 3D User Interfaces*. Osaka, Japan: IEEE, Mar. 2019, 103–110 10.

[SMOW20] SCHÜTZ, MARKUS, MANDLBURGER, GOTTFRIED, OTEPKA, JOHANNES, and WIMMER, MICHAEL. "Progressive Real-Time Rendering of One Billion Points Without Hierarchical Acceleration Structures". *Computer Graphics Forum* 39.2 (May 2020), 51–64 7.

[SP04] SAINZ, MIGUEL and PAJAROLA, RENATO. "Point-based rendering techniques". *Computers & Graphics* 28.6 (2004), 869–879 2.

[SP11] SCHEIBLAUER, CLAUS and PREGESBAUER, MICHAEL. "Consolidated Visualization of Enormous 3D Scan Point Clouds with Scanopy". *Proceedings of the 16th International Conference on Cultural Heritage and New Technologies*. Vienna, Austria, Nov. 2011, 242–247 2.

[SW11] SCHEIBLAUER, CLAUS and WIMMER, MICHAEL. "Out-of-Core Selection and Editing of Huge Point Clouds". *Computers & Graphics* 35.2 (Apr. 2011), 342–351 3.

[WBB*08] WAND, MICHAEL, BERNER, ALEXANDER, BOKELOH, MARTIN, et al. "Processing and interactive editing of huge point clouds from 3D scanners". *Computers & Graphics* 32.2 (2008), 204–220. DOI: https://doi.org/10.1016/j.cag.2008.01.010 3.

[ZPVG01] ZWICKER, MATTHIAS, PFISTER, HANSPETER, VAN BAAR, JEROEN, and GROSS, MARKUS. "Surface splatting". *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. 2001, 371–378 3.