Eurographics Symposium on Parallel Graphics and Visualization (2021)
M. Hadwiger, M. Larsen, F. Sadlo (Editors)

UnityPIC: Unity Point-Cloud Interactive Core

Y. Wu[1], H. Vo[1,3,4], J. Gong[2], and Z. Zhu[1,3]

[1]Department of Computer Science, The CUNY Graduate Center
[2]Civil and Environmental Engineering, Rutgers University
[3]Department of Computer Science, The City College of New York
[4]Center for Urban Science and Progress, NYU

Abstract

In this work, we present Unity Point-Cloud Interactive Core, a novel interactive point cloud rendering pipeline for the Unity Development Platform. The goal of the proposed pipeline is to expedite the development process for point cloud applications by encapsulating the rendering process as a standalone component, while maintaining flexibility through an implementable interface. The proposed pipeline allows for rendering arbitrarily large point clouds with improved performance and visual quality. First, a novel dynamic batching scheme is proposed to address the adaptive point sizing problem for level-of-detail (LOD) point cloud structures. Then, an approximate rendering algorithm is proposed to reduce overdraw by minimizing the overall number of fragment operations through an intermediate occlusion culling pass. For the purpose of analysis, the visual quality of renderings is quantified and measured by comparing against a high-quality baseline. In the experiments, the proposed pipeline maintains above 90 FPS for a 20 million point budget while achieving greater than 90% visual quality during interaction when rendering a point-cloud with more than 20 billion points.

CCS Concepts

• ***Large data visualization*** → *point cloud visualization;* • ***Spatial data structure*** → *hierarchical data structure;*

1 Introduction

Point clouds surged in popularity as a way to capture the real-world environment. Accurate collection methods can produce point clouds with millimeter precision, and mobile scanning systems are available to perform city-scale bulk collection. Depending on the size of the target and the resolution of the scan, the number of points in a point cloud can exceed the billions. Although ineffective in its raw format, point cloud data contain high and low level semantic information that can be extracted via post-processing techniques. Such information is useful across a wide array of applications such as urban planning [UDGGR20, ZWW*20], disaster evaluation [FGKG15], and virtual site visits [MTRGGA*11, RCB19]. Recent hot-topics such as autonomous vehicles heavily leverage point clouds for localization and object-detection [CLF*20].

This work is focused on a subclass of point clouds that exceed the size of the available main memory, which will be referred to as large point clouds. The visualization of large point clouds requires a pre-processing step to generate an efficient out-of-core hierarchical structure that supports LOD renderings. In the context of user-driven interactive point cloud applications, a scalable rendering pipeline is a shared commonality that forms the basis for interactions. Often times, the support for large point clouds is a requirement to ensure accessibility across machines with ranging performance specifications. However, the development of a scalable rendering pipeline for point clouds is non-trivial and consumes a significant portion of the application's development timeline. There exists a motivation to generalize the rendering pipeline as an implementable interface, which will greatly expedite the development process. As an engineering decision, the Unity Development Platform is chosen as the platform for the interface due to its large existing framework and diverse support for 3-D applications. Furthermore, Unity projects are portable across popular operating systems and target platforms.

Many scalable point cloud visualization applications exist in the current literature. However, their performances vary with respect to their out-of-core data structure and the rendering technique used. The data structure is often a tree structure where successive levels represent higher resolutions of the point cloud, and the type of structure used directly determines the cost of frustum culling, which is equivalent to the number of nodes traversed. Deeper trees allow for more precise LOD refinements, but are more costly to traverse in comparison. On the other hand, the visual quality of the scene is determined by the rendering technique and the available point budget, which is the maximum number of points any scene is allowed to have. Although interactive FPS can always be maintained by lowering the point budget, it is necessary to emphasize that greater performance is essential in order to maximize visual quality. Improving the GPU performance will raise the upper-bound on the point budget and consequently raise the potential for higher

visual quality, given that the rendering technique has the capacity for attaining higher visual quality.

The main contribution for this work is a Unity component which encapsulates the proposed point cloud rendering pipeline. The proposed pipeline allows for rendering large point clouds with improved performance. To improve the visual quality and rendering performance of the pipeline, two algorithmic improvements were proposed: dynamic batching and delayed expansion. The proposed dynamic batching scheme addresses the adaptive point sizing problem for LOD point cloud structures. Delayed expansion is an approximate rendering algorithm that significantly reduces the number of fragment operations. As a point of reference, we've achieved a 75% FPS increase at 10 million points and a 130% increase at 20 million points. The overall pipeline is then evaluated quantitatively in the context of interactive visual quality. According to the experiments, the pipeline maintained above 90 FPS for a 20 million point budget while achieving greater than 90% visual quality during interaction when rendering a point-cloud with more than 20 billion points.

2 Related Works

Unstructured point cloud formats are converted to out-of-core multi-resolution hierarchical data structures to enable LOD interactive rendering in real-time. In order to produce consistent LODs, an effective sampling method is required. Sainz et al. [SPL04] provides an overview of the basic hierarchical point cloud data structures, and in the work of Potree, Schütz et al. [Sch16] included a survey of various sampling techniques.

QSplat [RL00] was the earliest work to develop a multi-resolution rendering system that can scale to large out-of-core point clouds. Each node in the hierarchy is represented by a spherical bounding volume and contains exactly one point. Sequential Point Tree (SPT) [DVS03] is an ordered array structure optimized for GPU rendering that allows LOD representations. XSplat [PSL05] is an SPT based rendering system with an out-of-core extension. The Layered point cloud (LPC) [GM04] was the first to introduce object-space point clouds for each node in its hierarchy. Object-space point clouds are subsamples of the original point cloud confined to specific bounds. Nested octree [WS06] is similar to LPC, but uses an octree instead of a binary tree. Internal octrees are memory optimized SPTs that allow quick LOD selection by rendering up to a particular index. Instant Points by Wimmer et al. [WS06] achieved high GPU throughput at an interactive FPS by leveraging block-based culling and memory optimized SPTs. Modifiable nested octrees [SW11] are an extension of nested octrees. The inner octree is replaced by a grid, which allows fast insertions and deletions. The selection octree is introduced to accommodate point selection with a volumetric brush. Scheiblauer et al. [Sch14, SW11] proposed a large-scale point cloud rendering system that supports editing of point clouds with more than a billion points as part of the SCANOPY project. Potree [Sch16] is a web-based point cloud renderer that supports large point clouds. Its hierarchy is stored in chunks that can be loaded on demand during run-time, granting even greater scalability over the initial load time. The Potree file format chose a poisson-disk sampling method after surveying various sampling methods, which produced the best results visually. Other Unity-based large point clouds have been proposed [Fra17, SNTO19]. However, they do not focus on making novel improvements to the pipeline performance.

Schültz et al. [SMOW20] proposed a progressive rendering pipeline without hierarchical acceleration structures. Although it can only render point clouds that fit in GPU memory, it achieves impressive throughput and pleasant visual convergence by using randomization. The idea behind the reprojection technique introduced in their work is similar to our proposed delayed expansion rendering algorithm. It leverages the 2-D view-port space as the basis for determining visibility (occlusion culling).

Recent advances in augmented reality and virtual reality technology gave rise to immersive interactions in 3-D environments. Unity has been central to augmented reality (AR) and virtual reality (VR) applications, with the Mixed-Reality Toolkit (MRTK) [Mic16] at the core of it. AR and VR also imposed stricter interactive requirements for 3-D applications. Applications must maintain 90fps on a set of stereo displays, otherwise the interactive experience can be nauseating. Recent work by Schültz et al. [SKW19] focuses on maintaining continuous LOD and reducing clutters in high density rendered regions. They suggest that these qualities can improve the interactive experience. Stets et al. [SSCG17] proposed a VR application for point cloud labeling.

Point splatting techniques determine the quality of the rendering. Screen-facing circles or squares are the simplest. However, they lead to unstable visuals due to frequent spatial-aliasing and z-fighting. To achieve higher visual quality, Botsch et al. [BHZK05] proposed High-Quality Blending, a multi-pass rendering algorithm that includes three passes: a visibility pass, an attribute pass, and a normalization pass. The original work uses normal-oriented splats, however, normals aren't always available so a simplified version with screen-facing splats is used instead in Potree [Sch16]. Schütz et al. [Sch16] later proposed an interpolation method using paraboloids to resolve overlapping points. The resulting one-pass algorithm improved high resolution details such as texts in close-viewing, but it does not address aliasing. Jump flooding [Far14, RT06] is an approximation approach to rendering point clouds. However, the algorithm requires many iterations and isn't suitable for interactive rendering.

Per-point adaptive point sizing is difficult to achieve for Potree's data structure as a node may contain points belonging to multiple LODs. Schütz et al. [Sch16] leveraged the GPU's parallel processing capabilities to determine the point sizes on a per-point basis. Metadata from the rendered part of the hierarchy is packed into an array and sent to the GPU. Per-point adaptive point sizing is achieved by traversing that hierarchy in the vertex shader.

3 Preliminaries

3.1 Standard Large Point Cloud Rendering Pipeline

The standard large point cloud rendering pipeline can be summarized in three parts:

- **Traversal:** The traversal process determines the set of visible points for a given camera position and orientation. For every new camera position and orientation, the hierarchy of the internal data structure is traversed until either all visible nodes are visited, or the point budget is satisfied. The traversal order is typically priority driven and the priority metric is implementation dependent.

- **Loading:** Points are loaded from the out-of-core structure for the visible nodes determined by the traversal process. Loaded points are queued for renderings.
- **Rendering:** The rendering step will upload the newly loaded points to the GPU. The new scene is rendered with all the uploaded points.

Note that pipeline parallelism is easily achieved with the above three processes. UnityPIC's specialization of the standard rendering pipeline will be discussed in Section 4.

3.2 Point Cloud Rendering in Unity

Unity is popularly known as a game engine in its infancy. However, its platform has since been extended to support applications in various fields such as automotive, film, architecture, and construction. Furthermore, recent contribution from Microsoft, the MRTK, has defaulted Unity as the main platform for AR and VR development. The proposed rendering pipeline, UnityPIC, is implemented as a Unity component in C#.

As per Unity's default paradigm, the standard approach to rendering point clouds would be to create a *GameObject* for each visible node. The node's points would be stored in the *MeshFilter* assigned to that *GameObject*. However, a major limitation of *MeshFilters* prevents this from being a practical solution: there is an implementation defined limit of 65,536 vertices per *MeshFilter*. It isn't uncommon for the point count in a single node to surpass that limit, and each *GameObject* can only be assigned one *MeshFilter*. New *GameObjects* would need to be created for the additional points, adding more overhead to the rendering pipeline. It is common for the number of *GameObjects* to be in the hundreds or even the thousands for a reasonable point budget (5 to 10 million). Each *GameObject* participates in the internal frustum culling and comes with an associated overhead as there is a set of life-cycle functions that are executed for each frame. Given the number of potential *GameObjects*, this overhead can become significant. Another problem with *MeshRenderers* and *MeshFilters* is that they are incompatible with multi-pass rendering algorithms such as High-Quality Blending, where an intermediate pass renders to a back-buffer.

The above overheads and limitations compel the use of *CommandBuffers* and *ComputeBuffers*, which are extensions to Unity's graphics API that allow for more explicit control of the rendering pipeline. *CommandBuffers* are queues for graphics operations that can be executed by either: 1) an explicit call to the graphics API, or 2) attaching it to a particular stage in Unity's rendering pipeline. *ComputeBuffers* are GPU storage units, which are buffers with a specified size and stride. Contrary to *MeshFilters*, there is no size limit to *ComputeBuffers*, and custom packing is allowed, although the corresponding shader code would be required to process it. Data stored in *ComputeBuffers* can be visualized through *CommandBuffer.DrawProcedural(...)* or *CommandBuffer.DrawProceduralIndirect(...)*, which maps to the corresponding functions in the supported graphics API. These graphics commands are specific and their executions are controlled by the developer, allowing more complex rendering algorithms. With *CommandBuffers* and *ComputeBuffers*, the only overhead is the draw-call allocations for each frame. Even so, the resulting overhead is significantly reduced in comparison to *MeshRenderers* and *MeshFilters*.

3.3 Internal Data Structure

UnityPIC conforms to the widely used Potree file format [Sch16]. PotreeConverter [Sch15] is an open-sourced tool that can convert raw point cloud file formats, such as LAS, LAZ, and PTX, to the format used by Potree. The Potree file format is an octree with cubic bounding boxes. A point spacing value is set for the root, and it is halved for nodes on each subsequent level. The point spacing is the minimum spacing that must be maintained between any two points, thus it essentially defines the resolution of the node. Internal nodes of the octree maintain the point spacing constraint, while leaf nodes act as buckets, ignoring the constraint. The points for internal nodes are sampled without replacement using a poisson-disk sampling method. Therefore, no copies of the points are stored, and the original point cloud can be retrieved through the union of all the nodes. The points for each node are stored in their own separate file, and the meta-data for the hierarchy is stored separately from the point data file. Rather than storing the meta-data in a single file, it is partitioned into multiple smaller files that can be loaded on demand.

Potree's file format addresses the issue of unbounded initial load time for the hierarchy meta-data. The size of the hierarchy is negligible compared to the size of the point data, but if the entire hierarchy is demanded at the start, it can cause significant delay. This format is currently the most suitable for large point clouds in terms of accessibility and scalability. The main drawback of on-demand hierarchy loading is the halting of the traversal process due to the loading itself. IO operations can take longer than one frame (assuming 60 FPS), and visually, it can be noticeable during erratic and rapid camera movements. In that case, the inconsistency in the scene convergence speed can be disruptive to the interactive experience. A potential fix for this is to load the demanded hierarchy chunk asynchronously as done in Potree's work [Sch16]. During traversal, nodes whose hierarchy requires loading will be skipped. However, doing so can cause some nodes to be queued for rendering but later can be cancelled. As the skipped nodes become available for traversal, they may replace some already loaded but less significant nodes in order to maximize the utility of the available point budget.

4 UnityPIC: Rendering Pipeline

4.1 Hierarchy Traversal

The hierarchy traversal iteratively adjusts a cut on the octree structure, where the cut divides the visible and non-visible nodes. The process starts from the root of the tree and proceeds following a priority metric. The metric used is the projected area of a node. The metric only matters in a relative sense so using $\frac{d_{ps}}{z}$ is sufficient for perspective projection, where d_{ps} is the point spacing defined in Potree for each hierarchical level and z is the distance from the node to the camera along the z-axis. Frustum culling is performed by projecting its vertices to the viewport and then finding overlap between the convex hull of the vertices and the viewport. However, this will only produce accurate results if the vertices don't fall behind the near clipping plane. In that case, the bounding-box is cut using the near clipping plane. The vertices behind the near clipping plane are discarded, and new vertices are generated along the edges at the point of intersection with the near clipping plane.

An attention mechanism is simulated by introducing two multi-

(a) *min-weight*$_{xy} = 1.0$ *(uniform weight)*

(b) *min-weight*$_{xy} = 0.5$

(c) *min-weight*$_{xy} = 0.1$

Figure 1. *Renderings of a top-down view over city blocks in Manhattan with varying values of min-weight*$_{xy}$*, a user defined parameter that specifies the weight applied to a node's traversal priority based on its 2-D projected position.*

plicative weights to the node priorities. In immersive 3-D interactions, we assume that the user focuses on the center of the rendering. Hence, the nodes closer to the center of the rendering are more heavily weighted. The first weight is computed as:

$$w_{xy} = 1 - (\sigma\sqrt{x^2+y^2})^2 \tag{1}$$

where x, y are screen coordinates ranging from -1 to 1, which are acquired by projecting the centroid of the node's bounding-box. σ is computed based on *min-weight*$_{xy}$, a user specified parameter that specifies the value of the weight when the distance is 1. Thus, the appropriate σ can be computed by substituting w_{xy} and $\sqrt{x^2+y^2}$ with the values *min-weight*$_{xy}$ and 1, respectively. Figure 1 demonstrates the effects of different *min-weight*$_{xy}$ values. It presents a trade-off between details around the center versus near the edges. It should be noted that if *min-weight*$_{xy}$ is too small, the transition between LODs can become noticeable. In Figure 1c, the transition between LODs near the boundaries of the image becomes noticeable. The top-left corner of the orange zoom-box shows a higher resolution, and a clear diagonal line can be drawn to separate the 2 different LODs. Setting both parameters to 0.5 works well in practice. Figure 1b demonstrates *min-weight*$_{xy} = 0.5$, where the center of the rendering is visibly sharper while the transition around the edges remain smooth.

The second weight is based on the depth of the node so that closer

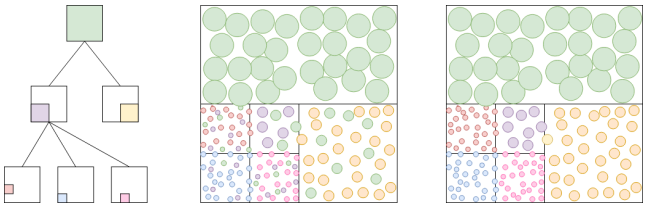


Figure 2. *An example hierarchy is given on the left image. The middle image shows the rendering results if done without dynamic batching. The color of the points corresponds to the color of the node they belong to. The right image shows the rendering results after dynamic batching is applied.*

points are prioritized. It is computed as:

$$w_z = 1 - (\sigma z)^2 \tag{2}$$

where z linearly increases from 0 at the near clipping plane to 1 at the far clipping plane. Another user specified parameter, *min-weight*$_z$, specifies the value of the weight at the far clipping plane. The weight functions are chosen such that the decay of the weight increases as the distance increases.

4.2 Point Loading: Integration with Dynamic Batching

Point loading involves IO and parsing. The proposed dynamic batching algorithm is integrated with point loading in order to maximize cache coherency.

The goal of the dynamic batching scheme is to address the adaptive point sizing problem, and the main obstacle to achieving adaptive point sizing is that the points within a single node may need to be rendered with different point sizes. Figure 2 demonstrates this problem in the 2-D space. The left image presents a hierarchy with 6 nodes. Rendering all the points from the nodes using adaptive point sizing yields the middle image. Notice that this rendering is difficult to achieve. For instance, the node that contains the green points spans 3 different resolutions, and thus the points are required to be rendered with 3 different point sizes in order to achieve adaptive point sizing.

The idea behind the dynamic batching scheme is to allow a node to only render points that belong to its LOD, and delegate the rendering of other points to its descendants. The proposed dynamic batching algorithm generates, for each node, an augmented point list which includes the original points that belong to the node, and additional points inherited from its ancestors. Moving forward, the following terms are defined for the sake of clarity:

- **Original points** - The points that belong to a node as defined in Potree's file format.
- **Inherited points** - The points that a node inherits from its parent. It's simply all the points from its parent that falls within the node's bounds. Note that this includes the points that the parent inherited from the parent's parent as well, and so on. This definition is recursive, and the node essentially inherits points from all of its ancestors.
- **Augmented point list** - The list of points including original

points and inherited points. The list is partitioned such that all the points that belong to the same octant are contiguous.

To see how adaptive point sizing can be achieved with the augmented point list, consider the green and yellow nodes from Figure 2. Notice how the augmented points of the yellow node already includes the green points that fall within its bounds. For the yellow node, all of its augmented points are rendered. For the green node, only the points in the top two quadrants are rendered. The points in the bottom right quadrant are rendered by the yellow node, and the points in the bottom left quadrant are rendered by the other descendants. The right image of Figure 2 shows the rendering results after dynamic batching. Notice that all the nodes only render with one point size. The rendering of augmented point lists will be further discussed in Section 4.3.1. To summarize, the work of Schütz et al. [Sch16] computes point sizes on the fly, while the dynamic batching algorithm acts as a pre-processing step to simplify point sizing to a per-node basis.

Algorithm 1: Dynamic batching algorithm

```
1  Function get_octant(p)
2      return the octant that p belongs to;
3  Procedure dynamic_batching(A_p, n_p, A_o, n_o)
4      A_aug[0...n_p + n_o − 1]; // Augmented point list.
5      C[0...7] = 0; // Point counts of each partition.
6      O[0...7] = 0; // Offset of each partition.
7      Compute offsets for each octant and store them in O;
8      for i ← 0 to n_p − 1 do
9          q ← get_octant(A_p[i]);
10         A_aug[C[q] + O[q]] = A_p[i];
11         C[q] = C[q] + 1;
12     end
13     for i ← 0 to n_o − 1 do
14         q ← get_octant(A_o[i]);
15         A_aug[C[q] + O[q]] = A_o[i];
16         C[q] = C[q] + 1;
17     end
18     return A_aug;
```

Algorithm 1 presents the pseudo-code for the dynamic batching algorithm. A_p and A_o are the inherited and original points, respectively, and n_p, n_o are their respective sizes. Due to the way that the points are stored, retrieving A_p from the node's parent doesn't require any computations. Lines 8 to 17 copies the points from A_p and A_o into their respective octant partitions.

Memory Overhead: Memory usage is estimated to increase by roughly 33%. Consider the fact that point clouds are typically scans of surfaces, we hypothesize that the number of original points for a node is roughly 4 times larger than the number of points inherited from the parent's original points. Extrapolating this relationship, the number of original points of a node is roughly 16 times larger than the number of points inherited from its parent's parent, and so on. The approximate size increase of the augmented point list is then:

$$\sum_{i=1}^{d} \frac{1}{4^i} \leq \frac{1}{3} \tag{3}$$

Id	*occupancy*	*visibility*	*inheritance*	*render*
1	11101111	01110111	10000001	00000001
2	00110111	11110000	00000111	00000000
3	10101001	11111111	10100001	10100001

Table 1. *Additional meta data stored for each node in order to determine the visible points for rendering. The render mask is derived from the other three.*

where d is the hierarchical depth of the node. To assess the significance of this increase, consider the typical memory usage during run-time. For instance, 20 million points at 20 bytes per point (12 for position, 4 for color, and 4 for other attributes) takes about 400 MB, and that increases to about 533 MB when using the dynamic batching scheme. We argue that the increase is acceptable given that a 20 million point budget is already an overestimate of typical point budgets used for interactive point cloud rendering.

CPU Overhead: The increased CPU overhead sums up to 2 additional iterations over the original and inherited points. Therefore, given that the number of points is increased by a constant factor, the running time of point loading remains the same asymptotically, which is $O(n)$. In practice, this overhead is trivial when compared to the IO time. Also, the procedure is run in parallel with the application's main thread, thus the application's rendering performance is not affected.

4.3 Point Rendering

In Potree's file format, the original point cloud can be retrieved from the union of all the nodes. Rendering a scene is simply rendering the most prioritized nodes that fit within the point budget.

4.3.1 Rendering Augmented Point Lists

With the proposed dynamic batching algorithm, a node can be excluded if all of its visible points are inherited and rendered. In order to perform these checks, each node maintains an additional 3 bytes of meta-data, with each byte containing information that pertains to the eight octants:

- The *occupancy* mask specifies occupancy of the octants within each node. If at least one point fall within the octant, the bit is set to 1, otherwise it is 0.
- The *visibility* mask maintains the visibility of its octants. Visible octants have their bits set to 1.
- The *inheritance* mask specifies the points that have been inherited. Points within an octant are inherited when that child node is loaded for rendering. If the points within the octant are inherited, the bit is set to 0. The *inheritance* mask is initialized with the *occupancy* mask and updated when the children nodes are rendered or removed.

Table 1 showcases an example of the meta-data. Looking at the *render* mask, node 1 has one octant rendered, node 2 is excluded from rendering, and node 3 has three octants rendered. Each octant is mapped to a specific bit, consistent with Potree's standard. The *render* mask encapsulates the rendering information and is computed

(a) *Paraboloids with 300 million points*

(b) *Paraboloids, increasing LOD*

(c) *High-Quality Blending with 300 million points*

(d) *High-Quality Blending, increasing LOD*

Figure 3. *A rendered scene from the Dublin city data-set. Figure 3a and Figure 3c show the difference in quality between paraboloids and high-quality blending. Figure 3b shows that the rendering quality may stop improving despite higher LOD when using paraboloids. Figure 3d shows that high-quality blending consistently improves upon higher LOD.*

as:

$$render = visibility \wedge inheritance \tag{4}$$

A node is rendered if $render \neq 0$, but only the points from octants with a 1-bit are rendered. The 1-bits indicate visible but un-inherited octants. Note that only one draw-call is allocated for each node, not one for every visible octant. Doing so causes the vertex shader to process every point, but the points that belong to non-visible octants are dropped by outputting no geometry from the geometry shader. Rendering augmented points causes increased vertex shader and geometry shader calls. However, the performance impact is small given that fragment shaders occupy significantly more GPU bandwidth, and the number of fragment operations remain the same since redundant points are dropped in the geometry shader.

4.3.2 Point Splatting Technique

The interpolation method from Schütz et al. [Sch16] and high-quality blending from Botsch et al. [BHZK05] were implemented and tested with UnityPIC. While interpolation can resolve overlapping points and achieve results that resemble a Voronoi diagram when the rendered surface is perpendicular to the camera, it remains disadvantaged as the method doesn't permit sub-pixel representation. As a result, interpolation is incapable of leveraging larger point budgets to create more detailed renderings.

Figure 3 showcases the difference in quality between interpolation and high-quality blending. As shown in Figure 3a, despite having a point budget of 300 million points, interpolation with paraboloids was unable to obtain good visual quality. Surfaces appear noisy and edges are unrefined. Figure 3b shows the center part of the scene rendered in increasing LODs. Upon reaching sub-pixel point sizes, further increasing the LOD yields no visual improvement. On the other hand, Figure 3c and Figure 3d shows that high-quality blending has a higher capacity for more detailed renderings. The rendering from the 300 million point budget produced noise-less surfaces, and the edges are sharper and smoother in comparison. Similarly, the increasing LODs show continuous refinements. As an important note, the visual differences between the interpolation method and high-quality blending are further amplified during interaction. The aliasing in the interpolation method results in frequent flickering, and far-away objects appear as silhouettes of white noise. In summary, high-quality blending contains a higher capacity for visual quality. Section 5.1 further explores the capacity difference and characteristics of high-quality blending empirically.

Adaptive Blend Depth: High-quality blending is parameterized by the blend depth value. If the blend depth is too small, aliasing will persist. If the blend depth is too large, the occluded parts of a scene may appear on the occluding surface. The base scaling for UnityPIC uses a factor that scales with linear-depth. However, as the blend depth approaches 0 at the near clipping plane, nearby points may not benefit from the blending. To account for nearby points, we leverage the point density information embedded in Potree. In addition to the linear-depth scaling, the radius of the point, which is set to the point spacing, is used as a constant that offsets the blend depth. Conceptually, this can be visualized as the point overlapping with the linear-depth map in 3-D space. The final blend depth is defined as:

$$blend\text{-}depth = radius + factor * linear\text{-}depth \tag{5}$$

4.3.3 Delayed Expansion & Occlusion Culling

The main bottle-neck for GPU performance is the rasterization and fragment shader cost. In the case of point cloud rendering, each vertex is transformed into two triangles using the geometry shader, resulting in a large number of fragment operations. Furthermore, due to the lack of effective occlusion culling methods for point clouds, over-draw is inevitable on say, a consumer-grade display with a full-HD resolution (~2 million pixels) with a 5 million point budget. We present an approximate rendering algorithm which delays any non-trivial rasterization until all occlusions are resolved. The algorithm yields a significant boost in GPU performance while sacrificing minimal rendering accuracy.

Algorithm: The approximate rendering algorithm consists of two passes: a *point to-pixel* pass and an *expansion* pass. The *point to-pixel* pass renders the points as single pixels to the a high dynamic range (HDR) texture. For each point, its clip space position, radius, color, and depth are outputted to the target texture at the projected pixel location of the point center. The resulting graphics buffer con-

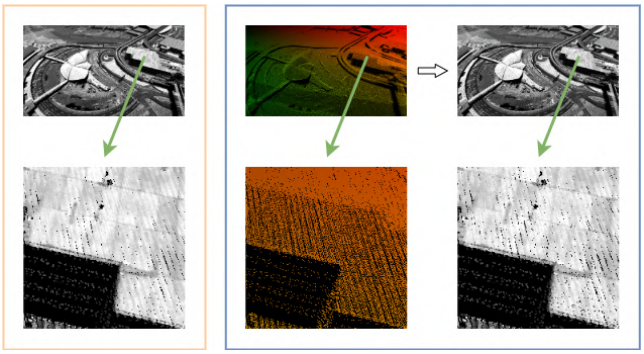


Figure 4. *Tradition point cloud rendering pass (left) versus two pass delayed expansion rendering (right).*

tains the attributes of front-most point for each pixel. Notice that the geometry shader is not necessary for this pass. The *expansion* pass then treats each pixel as a point and performs a normal rendering pass. Figure 4 illustrates the delayed expansion process in comparison to a traditional rendering pass. The clip space positions are required to prevent noticeable snapping during interaction. Without it, the *expansion* pass can only assume the center of the pixel to be the position of the point. The snapping between pixels is noticeable, especially when a point moves diagonally on the 2-D screen, where it is perceived to move in a staircase fashion instead. A collective of points exhibiting this behaviour looks noisy. Also, the intra-pixel position of a point can influence its rasterization results.

Performance: The *point-to-pixel* pass executes less graphics operations than a traditional point cloud rendering pass. First, the geometry shader is omitted since points are rendered as pixels. Second, the number of fragment operations is the same as the number of points. On the contrary, for a traditional point cloud rendering pass, the number of fragment operations is typically several times that. Furthermore, algorithms such as high-quality blending recommends a minimum point size of 2x2 pixels, in which case the number of fragment operations is at least 4 times that of the *point-to-pixel* pass. On the other hand, the *expansion* pass renders up to as many points as the number of pixels on the target resolution. The amount of work done in this pass is independent from the number of points rendered. Although delayed expansion incurs an overhead from the *expansion* pass, the cost saving from the *point-to-pixel* pass is expected to overcome it. Also, notice that the cost of the *expansion* pass becomes less significant as the number of points increases, which means that the percentage of performance gain increases along with it.

Visual Loss: A visual loss is incurred due to the approximation errors. Figure 5 illustrates the 2 types of potential errors: occlusion error (left), and intra-pixel error (right). When a point with a smaller projected size occludes a point with a larger projected size. Any information beyond the projected size of the point in the front is lost. On the right, intra-pixel positions are disregarded, and slight differences in intra-pixel position can affect rasterization. Note that the occlusion error is only relevant when using adaptive point sizing. When the point size is static, more distant points are guaranteed to have a smaller projected size. For the intra-pixel error, when the projected points are the same size, the maximum offset is 1 pixel.

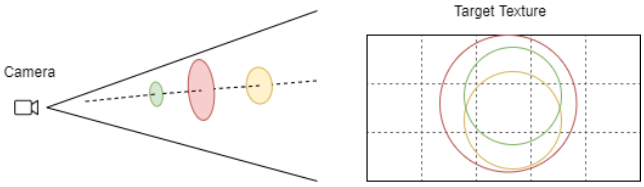


Figure 5. *Occlusion error (left) occurs when a point with a smaller projected radius occludes a point with a larger projected radius, in which case only the point in the front is rendered. Intra-pixel error (right) occurs when rasterization is slightly different between points that render to the same pixel.*

From Figure 4, the generated frame using delayed expansion has more holes due to the intra-pixel error. One potential fix for this is to have slightly higher point sizes.

5 Experiments & Analysis

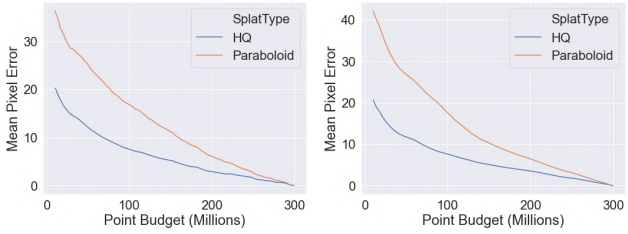
The experiments are performed on Windows (Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz; 64GB RAM; NVIDIA Titan XP). The data is stored on a hard disk drive (HDD) The point clouds used are from 2015 Aerial Laser and Photogrammetry Survey of Dublin City [LAV*17] and 2017 New York City Topobathymetric LiDAR Data [Cor17]. The Dublin City data-set contains 1.4 billion points (18.5 GB), and the NYC data-set contains 22.8 billion points (297 GB). Both data-sets contain only intensity values and no colors. All metrics of rendering quality will be based on intensity values.

Configurations: All renderings are done on FHD resolution (1920x1080). The minimum point size is set to 2 x 2 pixels, as recommended by [BHZK05] to facilitate sufficient blending, and maximum point size is set to 24 x 24 pixels. The blend depth factor is set to 0.01, which scales well with both data-sets. The maximum CPU-to-GPU transfer rate is set to 400,000 points per frame. $\textit{min-weight}_{xy}$ and $\textit{min-weight}_{z}$ are both set to 0.5 for the traversal process.

Intensity Normalization: Unlike color, the range and magnitude of intensity values may vary across data-sets. Normalization is required in order for standardized comparison as well as visualization. To normalize the intensity values, a dynamic range is calculated based on the points in the current scene, and then the values in the range are mapped uniformly to $[0,1]$. Specifically, first a local dynamic range is computed per node, and then the global range is derived as a weighted average of the ranges, where the weight is the number of points in the node. For each node, we assume the distribution of the intensity values to be normal, and the minimum and maximum are taken as two standard deviations to the left and right of the mean, respectively. Based on normal distribution, 4% of the points on the tail ends are expected to saturate.

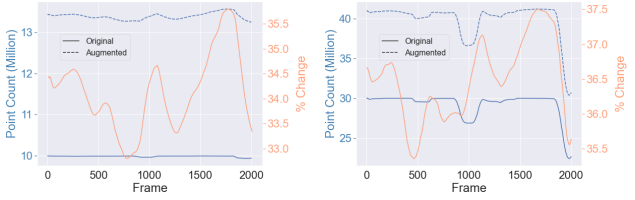
Metric: In order to measure the interactive visual quality, the interactive renderings for each technique are compared against a 300 million point budget baseline rendered with the respective technique. The similarity metric is the mean pixel error, as

$$\frac{1}{hw}\sum_{i=0}^{h}\sum_{j=0}^{w}|\mathrm{B}_{ij}-\mathrm{R}_{ij}| \tag{6}$$



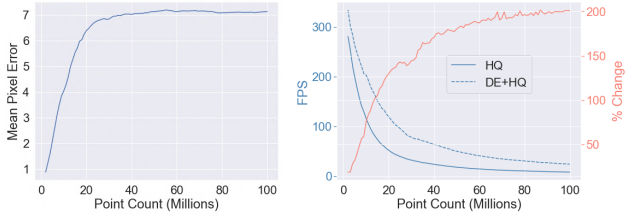
(a) *Dublin City* **(b)** *NYC*

Figure 6. *The convergence of the mean pixel error towards the high resolution baseline.*



(a) *Point Counts, 10 Mil* **(b)** *Point Counts, 30 Mil*

Figure 7. *Plotted results from a camera sequence in the Dublin data-set with point budget set to 10 million and 30 million. A moving average is computed for the sake of visualization.*



(a) *Approximation Error* **(b)** *FPS Difference*

Figure 8. *Figure 8a shows the error for delayed expansion when compared against the original. Figure 8b shows the FPS gain as a result of delayed expansion.*

where h and w are the width and height of the image, and B_{ij} and R_{ij} are the pixel values in the i^{th} row and j^{th} column of the baseline and the interactively rendered image being measured, respectively.

Metric Justification and Significance: The metric relies on an important assumption. It assumes that the baseline, which is rendered with a significantly higher point budget, is indeed of higher resolution and visual quality. To avoid confusions, we emphasize that a separate baseline is generated for each point splatting technique, and that the interactive renderings for each point splatting technique is compared with their own respective baseline. As there is no objective metric that determines the quality difference between different rendering techniques, the quality difference can be observed visually as in Figure 6. However, this does not impact the significance of our experiments as we intend to measure how well the interactive renderings can approach their higher quality baseline. The results are independent across different rendering techniques. Finally, it is important to note that mean pixel error does not measure noisiness of the renderings during interaction, since the mean pixel error does not implicate any relationship between neighboring frames.

5.1 High-Quality Blending: Visual Capacity

To further explore the capacity and efficiency of high-quality blending, the convergence towards the high quality baseline is observed. Figure 6 plots the convergence from a 10 million point budget towards the 300 million point budget baseline. For the sake of comparison, the results using the interpolation method are also included. As expected, the interpolation method starts from a higher mean pixel error and maintains a more linear descent towards its baseline. This implies that 1) the baseline is less sufficiently represented with a low number of points, and 2) the points added later in the convergence are creating more pixel changes. The convergence of high-quality blending exhibits more desirable behaviour. The 300 million point baseline can be represented with less than 10% error using 10 million points. Over the course of the convergence, new points are added on the basis of contribution rather than replacement, thus explaining the slow down in its convergence.

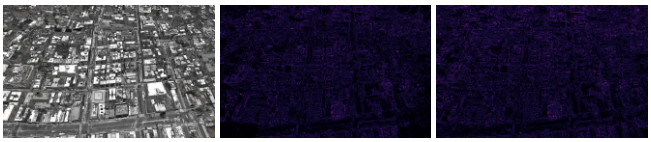
5.2 Dynamic Batching: Memory Usage

The dynamic batching scheme results in increased memory usage due to the redundancy of storing points in multiple LODs. An experiment is performed to empirically confirm the percent increase using the Dublin City data-set. A camera path is used and the original point counts and augmented point counts are captured for each frame. Figure 7 shows the percent of increase for 10 million and 30 million points. The increase remains approximately within the range of 33% to 36% for the 10 million, and increases to approximately 35% to 38% for the 30 million. The results shown here are consistent with the hypothesis in Section 4.2.

5.3 Delayed Expansion: Performance Gain & Visual Loss

The potential sources of error for delayed expansion as mentioned in Section 4.3.3 are studied empirically, with high-quality blending as the point splatting technique. Figure 8a measures the rendering's deviation from the original after integrating delayed expansion. The mean pixel error ranges from about 1.0 (0.4%) at 2 million points to about 7.2 (2.8%) at 100 million points. The rate of increase in the error decreases with respect to the point count. It converges after 40 million points and hovers around 7.2. This is justified when considering that the occlusion error is unlikely to occur due to additional points being far away, and the intra-pixel error is bounded for each pixel and thus constant overall.

The effectiveness of delayed expansion is contingent on a reasonable trade-off between visual quality and GPU performance gain. Figure 8b presents the GPU performance gain after integration. Contrarily to its accumulated error, the performance gain scales positively with the increase in point count, which is consistent with the analysis in Section 4.3.3. Even at 2 million points, there is still a



(a) *Top-down View*

(b) *Grazing View*

Figure 9. *For each view, from left to right: 1) scene visual, 2) difference image using 10 million point budget, and 3) difference image using 20 million point budget. The images being compared are not shown due to them being nearly identical. The scene visual is included as a visual aid for the analysis of the difference images. The difference images are generated using* ꟻLIP *[ANA*20].*

visible performance gain of 19%. The FPS is 75% higher at 10 million points and 130% higher at 20 million points. With a 100 million point budget, the rendering algorithm still maintains an interactive FPS of 23 FPS.

As a global indicator, mean pixel error doesn't give any insight into how the errors manifest in different scenarios. A difference evaluator for alternating images, ꟻLIP [ANA*20], is used to visualize and analyze the local errors. Figure 9 shows the generated difference images. A top-down view is compared in Figure 9a. The difference image shows that the errors are subtle but spreads across the entire view relatively uniformly. The grazing view in Figure 9b shows errors being concentrated at the far-away points. And the nearby points appear less erroneous, as they are sparse in comparison. In general, the difference images with the 20 million point budget exhibits more errors. This concludes that the occurrences of the errors are positively correlated with the amount of occlusion.

5.4 Interactive Performance

For the experiments, a single camera path is used and two camera sequences are captured: one for the baseline and the other for the interaction. The camera path is captured at 90 FPS. The frames for the baseline are generated by rendering until either when: the 300 million point budget is reached, or all the visible points are rendered. The frames of the interaction sequence are captured at the end of the frame regardless of the point count. Note that capturing the frame uses up a portion of the CPU and GPU bandwidth, so the FPS for the interaction sequence may be slightly lower than in practice. As a result, frames are dropped occasionally to synchronize with the captured camera path. The statistics for the skipped frames are derived through linear interpolation. Due to the relatively high point budgets used (10 & 20 million), only the high quality blending technique is used, as we have previously concluded that interpolation with paraboloids is insufficient when rendering a large number of points.

Interactive Visual Quality: Figure 10a and Figure 10d displays the interactive visual quality for a 10 million and 20 million point budget. For the most part, the mean pixel error is maintained below 25, suggesting that above 90% of the visual quality is preserved. The upward spikes correspond to fast rotations and accelerated movements (5x) of the camera. The drop in visual quality could be caused by two main reasons: the delayed adjustment of the point sizes due to the asynchronous traversal thread, and the IO bottleneck. Incorrect point sizes can only occur when the traversal thread is busy performing work from the previous frame. The IO bottleneck leaves a portion of the point budget unused and thus the drop in quality. The valleys correspond to instances where the camera moves into a closer up view of the point cloud. Smaller area of focus suggests that the total number of visible points is smaller, leading to the 300 million point budget to be under-utilized, and thus generating a more attainable baseline. Delayed expansion with high-quality blending is slightly worse in quality than normal high-quality blending, as expected. The mean pixel error worsens when the point budget is increased to 20 million points. However, as shown in Figure 8, the error converges and is relatively small.

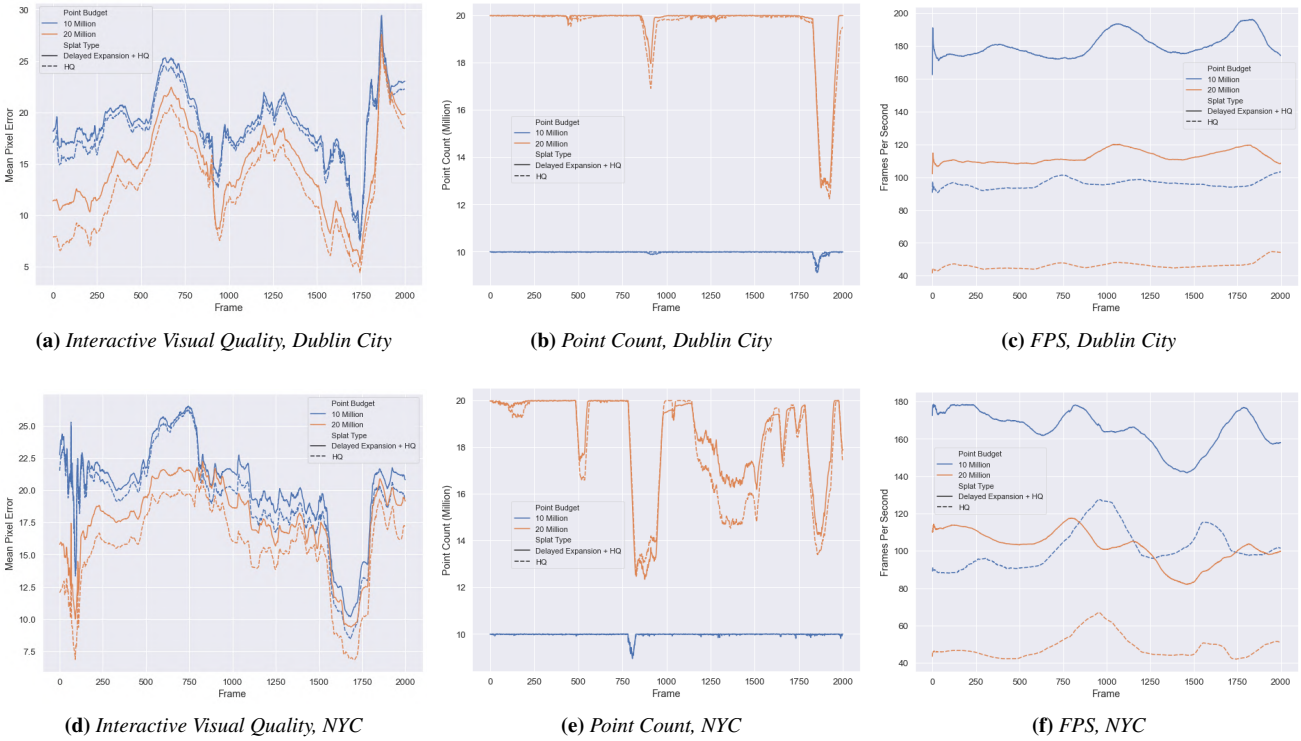
Point Counts: The locations where an IO bottle-neck is encountered can be identified by examining the drops in point counts through out the camera path. Figure 10b and Figure 10e plots the point counts per frame. The drops in point count correspond directly to rapid camera movements, in which the points exit the scene at a rate faster than they're loaded. Interestingly, delayed expansion allows the interaction sequence to maintain a slightly better point count. The differences are more apparent with the 20 million point budget. We suspect that it is due to its better GPU performance. The increased FPS results in less camera positions being dropped and allows the main thread to synchronize more closely with the traversal thread. More camera positions being dropped leads to greater changes during camera updates, to a greater number of points being added or removed. For the 10 million point budget, such differences are less likely as both rendering methods are able to maintain an FPS above 90.

FPS: The FPS statistic is necessary to gain a more in-depth understanding of the interactive performance. It is easy to maintain an interactive FPS by adjusting various aspects of the rendering pipeline, such as the point budget and CPU-to-GPU transfer rate. However, doing so sacrifices interactive visual quality. Figure 10c and Figure 10f plots the FPS for the same camera paths as the previous two. A moving average is taken to smooth the plot. Evidently, the integration with delayed expansion yields a significant performance boost.

In summary, the proposed pipeline maintains above 90 FPS for a 20 million point budget while achieving greater than 90% visual quality during interaction. As a final note, it is necessary to emphasize the significance of delayed expansion. The addition of delayed expansion allowed the pipeline to render 20 million points at a higher FPS than the traditional method at 10 million points.

6 Conclusion

The proposed large point cloud rendering pipeline, UnityPIC, is an standalone module with the goal of accelerating the point cloud application development process. While doing so, it improves performance to maintain scalability in its extension to a wide variety of



(a) *Interactive Visual Quality, Dublin City* **(b)** *Point Count, Dublin City* **(c)** *FPS, Dublin City*

(d) *Interactive Visual Quality, NYC* **(e)** *Point Count, NYC* **(f)** *FPS, NYC*

Figure 10. *Interactive performance for the Dublin City and NYC data-sets.*

point cloud applications with various other processing requirements. To ensure good visual quality, a dynamic batching algorithm is proposed to address the adaptive point size problem. GPU performance is improved significantly through the delayed expansion rendering algorithm. Although its potential is limited in out-of-core point cloud rendering due to the IO bottle-neck, the FPS gain is significant for practical point budgets, as shown empirically. Finally, the performance of the proposed pipeline is justified by a quantitative methodology of assessing interactive performance.

7 Acknowledgements

The research is supported in part by the National Science Foundation Awards #1827505 and #1737533, and Alfred P. Sloan Foundation Award G-2018-11069. Additional support is provided by Air Force Office for Scientific Research (Award #FA9550-21-1-0082), the Intelligence Community Center of Academic Excellence (IC CAE) at Rutgers University (Awards #HHM402-19-1-0003 and #HHM402-18-1-0007), and Vingroup Innovation Award VINIF.2019.20.

References

[ANA*20] ANDERSSON P., NILSSON J., AKENINE-MÖLLER T., OSKARSSON M., ÅSTRÖM K., FAIRCHILD M. D.: ꟻLIP: A Difference Evaluator for Alternating Images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 3*, 2 (2020), 15:1–15:23. 9

[BHZK05] BOTSCH M., HORNUNG A., ZWICKER M., KOBBELT L.: High-quality surface splatting on today's GPUs. In *Proceedings of the Second Eurographics / IEEE VGTC Conference on Point-Based Graphics* (Goslar, DEU, 2005), SPBG'05, Eurographics Association, p. 17–24. 2, 6, 7

[CLF*20] CHEN S., LIU B., FENG C., VALLESPI-GONZALEZ C., WELLINGTON C.: 3D point cloud processing and learning for autonomous driving, 2020. arXiv:2003.00601. 1

[Cor17] CORVALLIS Q.: Nyc topobathymetric data, 2017. URL: https://gis.ny.gov/elevation/NYC-topobathymetric-DEM.htm. 7

[DVS03] DACHSBACHER C., VOGELGSANG C., STAMMINGER M.: Sequential point trees. *ACM Trans. Graph. 22*, 3 (July 2003), 657–662. URL: https://doi.org/10.1145/882262.882321, doi:10.1145/882262.882321. 2

[Far14] FARIAS R.: *POINT CLOUD RENDERING USING JUMP FLOODING*. PhD thesis, 07 2014. 2

[FGKG15] FERNANDEZ-GALARRETA J., KERLE N., GERKE M.: Uav-based urban structural damage assessment using object-based image analysis and semantic reasoning. *Natural Hazards and Earth System Sciences 15* (06 2015), 1087–1101. doi:10.5194/nhess-15-1087-2015. 1

[Fra17] FRAISS S. M.: Rendering large point clouds in unity, Sept. 2017. URL: https://www.cg.tuwien.ac.at/research/publications/2017/FRAISS-2017-PCU/. 2

[GM04] GOBBETTI E., MARTON F.: 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. URL: https://doi.org/10.1016/j.cag.2004.08.010, doi:10.1016/j.cag.2004.08.010. 2

[LAV*17] LAEFER D. F., ABUWARDA S., VO A.-V., TRUONG-HONG L., GHARIBI H.: 2015 aerial laser and photogrammetry datasets for dublin, ireland's city center, 2017. URL: https://geo.nyu.edu/catalog/nyu-2451-38684, doi:10.17609/N8MQ0N. 7

[Mic16] MICROSOFT: Mixed reality toolkit. https://github.com/microsoft/MixedRealityToolkit-Unity, 2016. 2

[MTRGGA*11] MANCERA-TABOADA J., RODRÍGUEZ-GONZÁLVEZ P., GONZÁLEZ-AGUILERA D., FINAT J., ALONSO J. I., FERNANDEZ J., MARTÍNEZ-RUBIO J., MARTÍNEZ R.: From the point cloud to virtual and augmented reality: Digital accessibility for disabled people in san martin's church (segovia) and its surroundings. vol. 6783, pp. 303–317. doi:10.1007/978-3-642-21887-3_24. 1

[PSL05] PAJAROLA R., SAINZ M., LARIO R.: Xsplat: External memory multiresolution point visualization. *Proceedings of the 5th IASTED International Conference on Visualization, Imaging, and Image Processing, VIIP 2005* (09 2005), 628–633. doi:10.5167/uzh-47730. 2

[RCB19] RAHAMAN H., CHAMPION E., BEKELE M.: From photo to 3d to mixed reality: A complete workflow for cultural heritage visualisation and experience. *Digital Applications in Archaeology and Cultural Heritage 13* (05 2019). doi:10.1016/j.daach.2019.e00102. 1

[RL00] RUSINKIEWICZ S., LEVOY M.: Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques* (USA, 2000), SIGGRAPH '00, ACM Press/Addison-Wesley Publishing Co., p. 343–352. URL: https://doi.org/10.1145/344779.344940, doi:10.1145/344779.344940. 2

[RT06] RONG G., TAN T.-S.: Jump flooding in gpu with applications to Voronoi diagram and distance transform. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2006), I3D '06, Association for Computing Machinery, p. 109–116. URL: https://doi.org/10.1145/1111411.1111431, doi:10.1145/1111411.1111431. 2

[Sch14] SCHEIBLAUER C.: *Interactions with Gigantic Point Clouds*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria, 2014. URL: https://www.cg.tuwien.ac.at/research/publications/2014/scheiblauer-thesis/. 2

[Sch15] SCHÜTZ M.: Potreeconverter. https://github.com/potree/PotreeConverter, 2015. 3

[Sch16] SCHÜTZ M.: *Potree: Rendering Large Point Clouds in Web Browsers*. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria, Sept. 2016. URL: https://www.cg.tuwien.ac.at/research/publications/2016/SCHUETZ-2016-POT/. 2, 3, 5, 6

[SKW19] SCHÜTZ M., KRÖSL K., WIMMER M.: Real-time continuous level of detail rendering of point clouds. In *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)* (2019), pp. 103–110. 2

[SMOW20] SCHÜTZ M., MANDLBURGER G., OTEPKA J., WIMMER M.: Progressive real-time rendering of one billion points without hierarchical acceleration structures. *Computer Graphics Forum 39*, 2 (2020), 51–64. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13911, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13911, doi:https://doi.org/10.1111/cgf.13911. 2

[SNTO19] SANTANA NÚÑEZ J. M., TRUJILLO A., ORTEGA S.: Visualization of Large Point Cloud in Unity. In *Eurographics 2019 - Posters* (2019), Fusiello A., Bimber O., (Eds.), The Eurographics Association. doi:10.2312/egp.20191050. 2

[SPL04] SAINZ M., PAJAROLA R., LARIO R.: Points Reloaded: Point-Based Rendering Revisited. In *SPBG'04 Symposium on Point - Based Graphics 2004* (2004), Gross M., Pfister H., Alexa M., Rusinkiewicz S., (Eds.), The Eurographics Association. doi:10.2312/SPBG/SPBG04/121-128. 2

[SSCG17] STETS J. D., SUN Y., CORNING W., GREENWALD S. W.: Visualization and labeling of point clouds in virtual reality. In *SIGGRAPH Asia 2017 Posters* (New York, NY, USA, 2017), SA '17, Association for Computing Machinery. URL: https://doi.org/10.1145/3145690.3145729, doi:10.1145/3145690.3145729. 2

[SW11] SCHEIBLAUER C., WIMMER M.: Out-of-core selection and editing of huge point clouds. *Computers Graphics 35*, 2 (2011), 342 – 351. Virtual Reality in Brazil Visual Computing in Biology and Medicine Semantic 3D media and content Cultural Heritage. URL: http://www.sciencedirect.com/science/article/pii/S0097849311000057, doi:https://doi.org/10.1016/j.cag.2011.01.004. 2

[UDGGR20] URECH P. R., DISSEGNA M. A., GIROT C., GRÊT-REGAMEY A.: Point cloud modeling as a bridge between landscape design and planning. *Landscape and Urban Planning 203* (2020), 103903. URL: http://www.sciencedirect.com/science/article/pii/S0169204619316536, doi:https://doi.org/10.1016/j.landurbplan.2020.103903. 1

[WS06] WIMMER M., SCHEIBLAUER C.: Instant points: Fast rendering of unprocessed point clouds. pp. 129–136. doi:10.2312/SPBG/SPBG06/129-136. 2

[ZWW*20] ZHAO Y., WU B., WU J., SHU S., LIANG H., LIU M., BADENKO V., FEDOTOV A., YAO S., YU B.: Mapping 3D visibility in an urban street environment from mobile lidar point clouds. *GIScience & Remote Sensing 57*, 6 (2020), 797–812. URL: https://doi.org/10.1080/15481603.2020.1804248, arXiv:https://doi.org/10.1080/15481603.2020.1804248, doi:10.1080/15481603.2020.1804248. 1