A Data-Oriented Point-Based Rendering Engine on WebGPU

Alen Kurtagić*
Supervised by: Ciril Bohak,† and Žiga Lesar‡

Faculty of Computer and Information Science
University of Ljubljana
Ljubljana / Slovenia

Figure 1: Real-time visualization of a 3D Tiles point cloud of Ljubljana city center using WebGPU.

Abstract

Geospatial point-based datasets commonly exceed billions of points. Real-time rendering therefore requires out-of-core streaming, level of detail techniques, and numerically stable handling of Earth-scale coordinates. This paper presents a cross-platform C++ point-based rendering engine built on WebGPU. The engine follows a data-oriented design and supports streaming of 3D Tiles datasets. To maintain numerical stability at geospatial scales, the system uses camera-relative rendering using double-precision on CPU. Point cloud rasterization follows a multi-pass rendering pipeline coordinated by a render graph. To reduce the overhead of frequent GPU memory allocation during streaming, the engine includes a buffer suballocator that reduces allocation overhead compared to direct WebGPU buffer creation. The resulting system enables real-time rendering of massive point clouds on consumer hardware.

Keywords: Geospatial, Point Clouds, Real-Time Rendering, WebGPU

1 Introduction

Point-based representations are commonly used to model geospatial environments. Most notably in the form of point clouds, where each point stores a position and optional attributes such as color or intensity, without explicit connectivity or surface structure. These are typically acquired using LiDAR or photogrammetry technologies. Advances in these technologies have led to rapid growth in the size of point cloud datasets, which can span entire countries and contain billions of points. This scale introduces significant challenges for real-time visualization: datasets exceed GPU memory capacity, while coordinate magnitudes lead to numerical instability in single-precision arithmetic.

This paper addresses these challenges through a rendering engine. The system is built on Dawn[1] WebGPU implementation, enabling cross-platform deployment. It supports out-of-core streaming of massive datasets through 3D Tiles[2], while minimizing the overhead of frequent buffer creation during streaming through efficient buffer management. To maintain numerical stability at geospatial scales, the system employs camera-relative rendering, where positions are translated relative to the camera using double-precision arithmetic on the CPU. Rasterization is implemented as a multi-pass pipeline and is thus coordinated through a render graph.

The remainder of the paper is structured as follows. Section 2 reviews related work. Section 3 presents the system architecture. Section 4 describes 3D Tiles streaming, followed by camera-relative rendering in Section 5 and the point cloud rendering pipeline in Section 6. Section 7 discusses future work, and Section 8 concludes the paper.

*ak84795@student.uni-lj.si
†Faculty of Computer and Information Science, University of Ljubljana, Slovenia, ciril.bohak@fri.uni-lj.si
‡School of Engineering and Management, University of Nova Gorica, Slovenia, ziga.lesar@ung.si

[1] Dawn, a WebGPU C++ implementation
[2] OGC 3D Tiles Specification

2 Related Work

Potree [7] is a widely used WebGL-based system for interactive visualization of massive point clouds. It organizes data in an octree hierarchy and streams nodes out-of-core. Its rendering pipeline builds on multi-pass splatting techniques by Botsch *et al.* [1] and post-processing techniques for enhancing depth perception by Boucheny [2].

CesiumJS[3] is another widely used WebGL-based geospatial visualization library. Unlike Potree's focus on point clouds, CesiumJS targets heterogeneous geospatial content, including terrain, imagery meshes and point clouds.

Wegen *et al.* [12] observe that recent challenges such as massive point cloud sizes and web-based rendering are still rarely addressed in prior work on point cloud visualization.

These systems commonly rely on precomputed hierarchical Level of Detail (LOD) structures for out-of-core rendering. SimLOD [8] removes the need for precomputed hierarchies by constructing the LOD structure incrementally during rendering, allowing immediate visualization without an offline preprocessing step. Erler *et al.* [3] go further, combining sparse subsampling with neural heightmap refinement to visualize terabyte-scale datasets without precomputed hierarchies.

Schütz *et al.* [10] proposed Continuous Level of Detail (CLOD) to eliminate popping artifacts at density boundaries between adjacent octree nodes. Their method operates during rendering, where points are already loaded in memory and are stochastically filtered to achieve smooth density transitions.

Van Oosterom *et al.* [11] introduce a CLOD approach in which each point is assigned a CLOD value as part of the data representation itself. This enables smooth transitions during streaming, reducing the need to load unnecessary points in the first place.

Compute rasterization was introduced by Schütz *et al.* [9], who replaced the graphics pipeline entirely with compute shaders. Each point is projected and its depth and color are packed into a single 64-bit integer. A single `atomicMin` on a storage buffer resolves visibility and writes color simultaneously, rendering over two billion points in real time. While 64-bit `atomicMin` operation has been accepted as a WebGPU proposal[4], it is not yet available in Dawn's implementation. Furthermore, compute rasterization scales poorly with point sizes larger than one pixel.

3D Gaussian splatting [5] represents scenes using anisotropic 3D Gaussians instead of points and has emerged as a prominent recent approach, demonstrating that modern point-based rendering is increasingly moving beyond traditional point cloud representations.

3D Tiles is an Open Geospatial Consortium (OGC) community standard for streaming hierarchical geospatial datasets, including point clouds and Gaussian Splats. Previously mentioned CesiumJS viewer is built around this format. Cesium also provides Cesium Native[5], an open-source C++ library for 3D Tiles tile selection and loading.

data-oriented design (DOD) favors contiguous memory layout and composition over class hierarchies, primarily to improve cache utilization [4]. The entity component system (ECS) pattern is its most common realization: entities are plain identifiers, components are data stored in typed arrays, and systems are free functions that iterate over components.

Bevy[6] is an open-source Rust engine built on DOD principles. It splits state into a main world for simulation and a render world for GPU resources, with an extraction step that copies relevant data between them each frame. Bevy's plugin-based composition, two-world model and pipelined rendering inspired the architecture described in Section 3.

EnTT[7] is a modern C++ library for implementing the ECS pattern. It provides a lightweight, unopinionated set of ECS building blocks, including registries, views, and reactive utilities. This makes it well suited for flexible data-oriented architectures.

3 System Architecture

The system is implemented in C++20 with support for cross-platform deployment, including web builds via Emscripten[8]. It is organized into loosely coupled modules, as depicted in Figure 2.

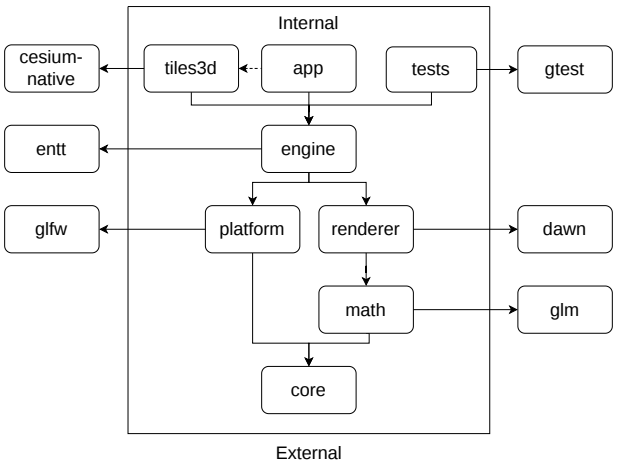

Figure 2: Module dependency. Solid arrows denote build-time dependencies; dashed arrows denote optional runtime dependencies.

The `core` module provides shared infrastructure, including a thread pool and timing utilities. It also defines system-level macros for logging and assertions. The `math` module

[3] CesiumJS, a web-based visualization library
[4] WebGPU proposal for 64-bit atomics
[5] Cesium Native, a 3D Tiles streaming library
[6] Bevy, a data-oriented game engine written in Rust
[7] EnTT, a modern ECS library
[8] Emscripten, a compiler toolchain to WebAssembly

extends GLM[9] with bounding volume, frustum, color and other types. The `platform` module wraps GLFW[10] to provide windowing and input handling. The `renderer` module provides an abstraction layer over Dawn's WebGPU implementation. It also includes a render graph, a buffer suballocator and a small shader preprocessor for resolving include directives.

The `engine` module builds on EnTT to provide an ECS with plugin-based composition, scheduling, and pipelined rendering. It integrates the lower modules into a cohesive runtime through built-in plugins. The `tiles3d` module optionally extends the engine with cesium-native integration for streaming 3D Tiles.

Finally, the `app` module composes the desired plugins into a running engine instance and configures the scene.

3.1 Plugin Composition

The engine is configured through plugins. A plugin is any type that implements a `build(Engine&)` method, enforced by a C++20 concept. Plugins may optionally declare a `require(Engine&)` method, which can assert conditions such as the presence of other plugins. A `finish(Engine&)` method, if provided, runs after all plugins have been built, allowing deferred plugin initialization that depends on resources registered by other plugins.

The engine ships with 17 built-in plugins, though any can be omitted or replaced. Figure 3 depicts these plugins and dependencies between them.

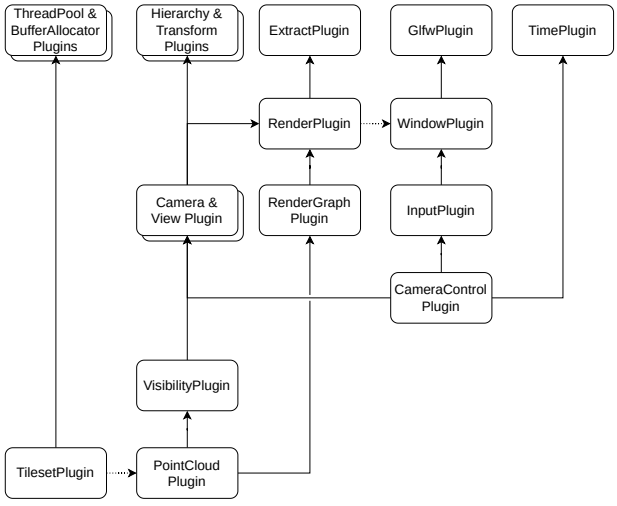

Figure 3: Plugin dependencies. Solid arrows denote required dependencies; dashed arrows indicate optional integration where the plugin adapts its behavior if the dependency is present.

[9] GLM, a maths library for graphics

[10] GLFW, a multi-platform windowing library

3.2 Data Model

The `engine` module builds on EnTT for its ECS implementation. The EnTT registry stores entities, component pools and shared resources. The engine maintains two separate worlds, each backed by an independent registry. The main world serves as the source of truth. The render world holds a rendering-oriented representation extracted from the main world each frame.

A key design challenge arises from the need to express relationships between entities. While some ECS frameworks provide built-in abstractions for this purpose, EnTT does not provide a canonical mechanism for this. As noted by the library author, various designs for relationships have been explored, but none have been considered suitable for general adoption [11]. To address this, a lightweight relation system is introduced.

A relation is a directed link between two entities within the same registry. A `Relation<Tag>` component on the source entity stores the target entity. On construction, a hook automatically attaches an `InverseRelation<Tag>` component on the target entity, maintaining a back-reference to the source entity. On destruction of either component, hooks automatically remove their counterpart. The tag type parameter ensures that each relation kind (e.g., ChildOf, RendersTo) produces distinct component types, allowing an entity to participate in multiple relation kinds.

Entities can also be linked across worlds. Tagging a main world entity with a `SyncTo<Render>` component creates a corresponding render world entity, with both sides pointing to their counterpart.

3.3 Scheduling Systems

Systems are free functions operating on the registry. A stage is an ordered group of these systems. Plugins register systems into these stages during initialization. Table 1 lists the stages for each world.

Table 1: World stages.

World	Stage	Purpose
Main	Input	Process input
	PreUpdate	React to input
	Update	Run main logic
	PostUpdate	React to changes
	Extract	Transfer data
Render	Prepare	Upload GPU resources
	Queue	Build draw lists
	PreDraw	Record draw commands
	Draw	Submit draw commands
	PostDraw	Present texture

[11] EnTT's author on relations

The two-world data model exists to enable pipelined rendering. By separating state into independent registries, the only data dependency between worlds is the `Extract` stage – a synchronization point where relevant state is copied between registries. Outside this stage, the two worlds are independent and can execute concurrently. This allows the render world to run on a dedicated thread, so that the main thread can advance to frame $N+1$ while the render thread renders frame N.

A pair of binary semaphores coordinates the two threads. The first thread to reach the synchronization point blocks until the other catches up. Once both are idle, the main thread runs extraction – temporarily injecting the render registry into the main registry as a shared resource, so that extraction systems can access both. To minimize this sequential overhead, extraction should avoid any heavy computation. The main thread then signals the render thread to begin and immediately advances to the next frame.

In the case of web builds, both worlds execute sequentially on a single thread, preserving the same stage structure, as multithreaded execution is not reliably available.

3.4 Render Graph

The `renderer` module provides a render graph for organizing multi-pass rendering. Passes declare which resources they read, write, or create, along with an execute callback that records GPU commands. Resources are either transient or imported from outside the graph, such as the swapchain texture.

Transient resources exist only within a single frame – a pass declares a resource description (e.g., a depth texture of a given format and size) and the render graph allocates it before the first pass that uses it and releases it once no subsequent pass reads it.

Before execution, a compilation step culls passes whose outputs are never read, propagating backward through the dependency graph. Surviving passes execute in the order they were added.

3.5 GPU Buffer Suballocator

The WebGPU API provides a blocking `createBuffer` method for creating buffers. Each buffer must declare its usage type (e.g., storage, vertex) at creation time. This usage determines which GPU operations the buffer can participate in. Although direct buffer creation is simple, each call carries backend overhead. Streaming point clouds requires frequent creation and destruction of buffers to hold incoming geometry, making this overhead add up.

To address this, the `renderer` module provides a buffer suballocator. It maintains a separate pool for each usage type, where each pool holds preallocated buffers, called pages.

When an allocation is requested, the suballocator searches the corresponding pool for a free range. Each page tracks its largest free range, allowing the search to skip full pages. Freed ranges are inserted back into a sorted list and joined with adjacent ranges to reduce fragmentation. If no page has sufficient space, a new one is created. Instead of returning a new buffer each time, suballocator returns an offset into an existing one. This reduces what would be many backend API calls to purely CPU-side logic.

Suballocator measurements are performed on an Intel Core i9-14900K with an NVIDIA GeForce RTX 4080 (16 GB VRAM) on DirectX12 WebGPU backend. The suballocator is benchmarked against direct buffer allocation, where each buffer is created with a separate `createBuffer` call. The benchmark performs 10,000 allocations of vertex buffers with sizes uniformly distributed between 128 KB and 1 MB, representative of point cloud tile data. Direct allocation cost is shown in Table 2.

Table 2: Average per-operation cost of direct buffer allocation.

Alloc (µs)	Free (ns)
57.9	174.7

Table 3 shows suballocator performance across page sizes, where larger pages favor allocation speed at the cost of slower frees.

Table 3: Average per-operation cost of the suballocator across page sizes.

Page (MB)	Pages	Alloc (µs)	Free (ns)
16	707	18.8	18.9
32	352	16.6	19.6
64	176	15.5	24.2
128	88	14.6	36.2

The suballocator achieves 3.1–4.0× speedup in allocation over direct buffer creation across all page sizes, and a 4.8–9.2× improvement in free time. A page size of 32–64 MB offers a practical balance between the two.

4 3D Tiles Streaming

A 3D Tiles tileset organizes data into a bounding volume hierarchy expressed in the Earth-centered Earth-fixed coordinate system (ECEF) coordinate system, which uses a Z-up convention. Each tile contains a reference to renderable content along with metadata used to determine whether it should be loaded. The hierarchy can take the form of an octree, quadtree, k-d tree or grid, with octrees being the most common choice for point clouds. Each tile declares a `geometricError` – the error, in meters, introduced if this tile is rendered and its children are not. This value decreases at each successive level of the tree, with leaf tiles approaching zero. The `tiles3d` module converts from

the Z-up ECEF convention to the system's Y-up convention when loading tiles.

At runtime, the geometric error is projected into screen space to decide which tiles to render. For a tile with geometric error ε at distance d from the camera, the screen-space error (SSE) in pixels is

$$\mathrm{SSE} = \frac{\varepsilon \cdot f}{d}, \quad f = \frac{h}{2\tan(\theta/2)}, \tag{1}$$

where h is the viewport height in pixels and θ the vertical field of view. When a tile's SSE exceeds a configurable threshold, it is refined. The refinement strategy is specified per tile – `replace` refinement substitutes the parent with its children, while `additive` refinement renders children alongside the parent. Point cloud tilesets typically use additive refinement.

5 Camera-Relative Rendering

ECEF coordinates place the origin at the Earth's center. Thus, points near the Earth's surface typically have coordinate magnitudes on the order of 10^6 to 10^7 meters.

Normal binary floating-point numbers are represented in the form

$$(-1)^s \times 1.m \times 2^e, \tag{2}$$

where s is the sign bit, m is the mantissa, and e is the exponent. For a binary floating-point format with p bits of mantissa precision, the spacing between representable values for a given exponent e is

$$\Delta = 2^{e-(p-1)}. \tag{3}$$

The absolute spacing thus grows exponentially with the exponent.

Table 4 compares the spacing for single-precision ($p = 24$) and double-precision ($p = 53$) at magnitudes typical of ECEF coordinates. At these scales, single-precision spacing is too coarse for stable rendering, whereas double-precision remains sufficient by a wide margin.

Table 4: Approximate spacing between consecutive representable values at different coordinate magnitudes.

Magnitude	e	float32 Δ	float64 Δ
10^3 m	9	2^{-14} m	2^{-43} m
10^6 m	19	2^{-4} m	2^{-33} m
10^7 m	23	2^{0} m	2^{-29} m

To render a point, its world-space position must first be transformed into camera space. For a camera with position c and orientation R, a world-space point p is transformed in homogeneous coordinates as

$$\begin{bmatrix} p_{\text{view}} \\ 1 \end{bmatrix} = \begin{bmatrix} R^\top & -R^\top c \\ 0 & 1 \end{bmatrix} \begin{bmatrix} p \\ 1 \end{bmatrix}, \tag{4}$$

or equivalently,

$$p_{\text{view}} = R^\top (p - c). \tag{5}$$

The transformation contains a subtraction of the camera position from the point. At geospatial scales, both p and c have magnitudes on the order of 10^6 m, while their difference may only be meters. Subtracting two nearly equal large values causes the leading digits to cancel, making the result highly sensitive to already present rounding errors – a phenomenon known as catastrophic cancellation. Because single-precision provides only 24 bits of mantissa, this subtraction becomes highly sensitive at geospatial scales.

GPUs are primarily optimized for single-precision floating-point arithmetic. Although some hardware supports double-precision, double-precision throughput on consumer GPUs is often only a small fraction of single-precision throughput. Furthermore, WebGPU Shading Language (WGSL) does not yet expose double-precision types.

The solution is to perform the $p - c$ subtraction on the CPU-side in double-precision and upload the result to the GPU in single-precision.

To achieve this the `engine` module stores all CPU transformations in double-precision. In the render world, the camera position vector and rotation matrix are stored separately. Because the camera translation is applied during the CPU-side subtraction, the view matrix uploaded to the GPU contains only the inverse camera rotation, so that the camera is implicitly always at the origin.

During the `Prepare` stage, the camera position is subtracted from the translation column of each visible object's model matrix. The resulting camera-relative matrix is then cast to single-precision and uploaded to the GPU.

6 Point Cloud Rendering

The point cloud rendering pipeline consists of multiple passes organized through the render graph described in Section 3.4.

6.1 Continuous Level of Detail Filtering

As described in Section 4, the 3D Tiles standard organizes tiles into a hierarchy of discrete detail levels. When tiles at different depth levels are rendered together, differences in point density at their boundaries become apparent, producing an inconsistent appearance. As the camera moves, transitions between levels can also cause popping artifacts.

Schütz *et al.* address this with Continuous Level of Detail (CLOD), which replaces discrete tile-level visibility with gradual per-point visibility. In their data structure, each point is assigned an integer level corresponding to its depth in an octree hierarchy. By adding a pseudo-random value in $[0, 1)$ to each point's depth integer, the discrete levels are spread into a continuous range, allowing filtering on a continuous scale rather than at integer boundaries.

Figure 4: Final rendering output at 60 FPS on an RTX 4080 using the complete rendering pipeline.

This pseudo-random value is computed deterministically from the point's position

$$r(p) = \text{fract}\big(\cos(p_x + p_y + p_z) \cdot c\big), \tag{6}$$

where c is a large constant that decorrelates nearby points. Since the input depends only on position, each point produces the same value across frames.

Each point's native spacing s_{native} is determined by its depth

$$s_{\text{native}} = \frac{s_{\text{base}}}{2^{\text{depth}}}, \tag{7}$$

where s_{base} is the spacing at the root level.

The target spacing s_{target} defines the desired density at a given distance from the camera

$$s_{\text{target}} = d \cdot k, \tag{8}$$

where d is the distance to the camera and k is a quality factor. Since s_{target} grows linearly with distance, all points at a given distance are equally sized. Figure 5 illustrates this linear relationship.

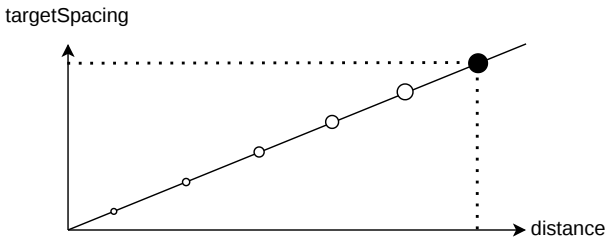

Figure 5: Target spacing s_{target} and point size both scale linearly with distance from the camera.

A point survives when its pseudo-random value falls below the ratio of native to target spacing

$$r(p) < \frac{s_{\text{native}}}{s_{\text{target}}}. \tag{9}$$

As distance increases, s_{target} increases, the ratio shrinks and fewer points survive.

The same filtering principle can be applied to 3D Tiles. The native spacing formula requires a root spacing value that 3D Tiles does not expose. On the other hand, each tile's `geometricError` already encodes analogous information, making it usable as s_{native}. A compute shader evaluates every point, computes the acceptance ratio, and appends surviving points to an output buffer via atomic operations.

6.2 Rasterization

WebGPU does not provide a primitive analogous to OpenGL's `gl_PointSize`, so each point is rasterized as a view-aligned billboard quad composed of two triangles.

Naive rasterization retains only the closest fragment per pixel. Nearby points at similar depths are discarded, even when they likely belong to the same surface.

Botsch *et al.* address this with a three-pass approach. A depth pre-pass writes a depth buffer where each fragment is pushed slightly further from the camera, creating a thin tolerance range. An accumulation pass then renders all points with additive blending, testing against this shifted depth buffer with depth writes disabled. Any fragment that falls within the tolerance range contributes its color, allowing nearby points to blend together. A final normalization pass produces the average color of all contributing fragments, as illustrated in Figure 6.

Figure 6: Naive rasterization (left) and the three-pass blending approach (right), which blends neighboring points at similar depths, creating smoother surfaces.

This approach is adopted with minor modifications. The depth pre-pass offsets fragments by approximately 1% of

their depth to establish the tolerance range. The accumulation pass writes to a transient render graph texture using additive blending. Each billboard quad is sized according to s_{target}, matching the density defined by the CLOD. For performance, the normalization step is merged with the Eye-Dome Lighting (EDL) pass into a single pass.

6.3 Eye-Dome Lighting

Point clouds typically contain no surface normals and therefore cannot be conventionally lit. Boucheny addresses this with EDL, an image-space post-processing technique that derives shading purely from depth discontinuities in the depth buffer.

For each pixel, the depth buffer is sampled at eight neighboring pixels with a configurable radius. For the i-th neighbor, the response is defined as

$$r_i = \max\big(0,\ \log_2 z_c - \log_2 z_i\big) = \log(z_c/z_i) \qquad (10)$$

where z_c and z_i are the linearized depths of the center pixel and the i-th neighbor, respectively. The response measures depth ratios rather than absolute differences, making it scale-invariant. The subtraction form is used because division is computationally more expensive.

The mean response $\bar{r}$ is mapped to a shading factor s as

$$s = \exp(-\bar{r} \cdot k), \qquad (11)$$

where k controls the shading strength. The final pixel color is computed as

$$\mathbf{c}_{\text{out}} = s\,\mathbf{c}_{\text{in}}, \qquad (12)$$

so that pixels near depth discontinuities appear darker, enhancing depth perception.

With blending alone, the point cloud appears smooth but flat. EDL recovers depth perception by darkening pixels where the depth buffer changes abruptly as shown in Figure 7.

Figure 7: Without EDL (left) and with EDL applied (right), which enhances depth perception by highlighting depth discontinuities.

7 Future Work

Several avenues remain for future exploration. The architecture is designed to accommodate 3D Gaussian splatting, enabling a transition from discrete to high-fidelity rendering. Additional improvements include integrating a Nishita sky model [6] and adopting an infinite z-plane depth buffer to enhance realism and precision in world-scale environments. Finally, the development of an internal out-of-core tiler would remove the current dependency on external preprocessing platforms for generating 3D tiles datasets, resulting in a fully self-contained system.

8 Conclusion

This paper addressed the challenges of real-time rendering of massive geospatial point clouds, including out-of-core data access, numerical instability at Earth-scale coordinates and the overhead of the WebGPU API.

The system follows DOD principles and uses an ECS architecture to structure data. A two-world model separates simulation and rendering state, enabling pipelined rendering, where rendering and simulation of consecutive frames overlap. A render graph is used to organize multi-pass rendering and manage transient GPU resources. Camera-relative rendering is applied to maintain numerical stability at geospatial scales. 3D Tiles streaming enables out-of-core rendering. The rendering pipeline combines CLOD, blending and EDL to further improve visual quality. A buffer suballocator reduces the cost of frequent GPU buffer creation, achieving a 3.1–4.0× speedup.

The resulting system enables efficient real-time visualization of large point clouds on consumer hardware while remaining portable across native and web platforms. These results demonstrate that a data-oriented design combined with WebGPU provides a practical solution for scalable geospatial rendering.

Acknowledgement

The research was conducted within the project Geospatial Information Technologies for Resilient and Sustainable Society (GeoAI) (GC-0006), funded by the Slovenian Research and Innovation Agency (ARIS).

References

[1] Mario Botsch, Alexander Hornung, Matthias Zwicker, and Leif Kobbelt. High-quality surface splatting on today's GPUs. In *Eurographics Symposium on Point-Based Graphics*, 2005.

[2] Christian Boucheny. *Interactive Scientific Visualization of Large Datasets: Towards a Perceptive-Based*

Approach. PhD thesis, Université Joseph Fourier, Grenoble, 2009.

[3] Philipp Erler, Lukas Herzberger, Michael Wimmer, and Markus Schütz. LidarScout: Direct out-of-core rendering of massive point clouds. In *High-Performance Graphics*, 2025.

[4] Richard Fabian. *Data-Oriented Design: Software Engineering for Limited Resources and Short Schedules*. 2018.

[5] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d Gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, 42(4), 2023.

[6] Tomoyuki Nishita, Toshiya Sirai, Katsumi Tadamura, and Eihachiro Nakamae. Display of the earth taking into account atmospheric scattering. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 175–182. ACM, 1993.

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

[8] Markus Schütz, Lukas Herzberger, and Michael Wimmer. SimLOD: Simultaneous LOD generation and rendering for point clouds. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 7(1), 2024.

[9] Markus Schütz, Bernhard Kerbl, and Michael Wimmer. Software rasterization of 2 billion points in real time. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 5(3):1–17, 2022.

[10] Markus Schütz, Katharina Krösl, and Michael Wimmer. Real-time continuous level of detail rendering of point clouds. In *IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, 2019.

[11] Peter van Oosterom, Simon van Oosterom, Haicheng Liu, Rod Thompson, Martijn Meijers, and Edward Verbree. Organizing and visualizing point clouds with continuous levels of detail. *ISPRS Journal of Photogrammetry and Remote Sensing*, 194:119–131, 2022.

[12] Ole Wegen, Willy Scheibel, Matthias Trapp, Rico Richter, and Jürgen Döllner. A survey on non-photorealistic rendering approaches for point cloud visualization. *IEEE Transactions on Visualization and Computer Graphics*, 31(9):4511–4533, 2025.