Web-Based Streaming and Rendering Performance Comparison: CityJSONSeq, FlatCityBuf, and OGC 3D Tiles 1.1

Ziya Usta[1], Alper Tunga Akın[2]

[1]Department of Geomatic Engineering, Engineering Faculty, Artvin Çoruh University, Artvin, Turkey - ziyausta@artvin.edu.tr
[2]Department of Geomatic Engineering, Engineering Faculty, Karadeniz Technical University, Trabzon, Turkey - alpertunga@ktu.edu.tr

Keywords: 3D City Models, CityJSONSeq, FlatCityBuf, OGC 3D Tiles, WebGPU, Web-based Visualization.

Abstract

While CityGML was traditionally the dominant standard and format for 3D city models, the ecosystem now includes CityJSON and its streaming derivatives, CityJSONSeq and FlatCityBuf as other formats. Consequently, modern web-based distribution relies on three competing frameworks, namely CityJSONSeq utilizing sequential JSON Lines, FlatCityBuf providing zero-copy binary encoding, and OGC 3D Tiles 1.1 delivering pre-triangulated glTF. Despite their concurrent evolution, empirical comparisons of their data-loading and rendering efficiency are lacking. This study bridges this gap by evaluating these formats across five real-world CityJSON datasets, ranging from 853 to 145,865 building geometries. The evaluation utilizes a custom, bandwidth-controlled loading harness and an independent WebGL2 and WebGPU rendering pipeline, developed without third-party scene-graph libraries to strictly isolate the graphics API. Results demonstrate that 3D Tiles/glTF consistently yields the smallest file sizes, reducing payloads by 29 to 73 percent compared to CityJSONSeq. It is also the fastest to parse and, by eliminating client-side triangulation, reaches a fully render-ready state significantly quicker, performing up to 47.1 times faster than CityJSONSeq and 36.9 times faster than FlatCityBuf on the largest dataset. Among CityJSON-derived formats, FlatCityBuf is particularly effective for large-scale data parsing due to its zero-copy deserialization capability, substantially outperforming CityJSONSeq. When paired with an optimized decoder, it also achieves greater overall rendering readiness at scale. Nevertheless, because both CityJSON formats require client-side triangulation, 3D Tiles/glTF maintains an order-of-magnitude performance advantage. Ultimately, these empirical findings provide web-based Geographic Information System architects with definitive, scale-dependent guidance for selecting the optimal streaming format.

1. Introduction

The proliferation of Urban Digital Twins (UDTs) and smart city applications has catalyzed a paradigm shift from desktop-bound Geographic Information Systems (GIS) to interactive, web-based 3D visualization platforms (Batty, 2018; Usta et al., 2026). Modern 3D city models (3DCMs) are no longer merely visual representations; they are highly detailed, semantically rich datasets encompassing complex geometries, textures, and deep attribute hierarchies compliant with data models such as CityGML (Biljecki et al., 2015; Kolbe, 2009). As urban datasets grow exponentially in size and spatial complexity, disseminating and visualizing this data over the web has become a critical technical bottleneck (Chaturvedi et al., 2015). Traditional file-based delivery mechanisms over HTTP suffer from prolonged loading times, excessive client-side memory consumption, and rigid data monoliths that inhibit dynamic interaction (Mao & Ban, 2011; Vitalis et al., 2020). Consequently, the geospatial community has aggressively pursued novel web-streaming formats and spatial indexing strategies to enable scalable, real-time exploration of massive urban environments directly in the browser.

Historically, the industry has addressed massive-scale 3D visualization through the OGC 3D Tiles specification, currently in its advanced 1.1 iteration (OGC, 2022). Adopted widely as the de facto standard for massive heterogeneous 3D geospatial datasets, 3D Tiles leverages Hierarchical Level of Detail (HLOD) and spatial bounding volumes to selectively stream only the data visible within a user's viewport. By utilizing batched draw calls, implicit tiling, and modern graphics integrations like glTF, 3D Tiles 1.1 delivers unparalleled client-side rendering performance, particularly when paired with next-generation APIs like WebGPU. However, the creation of a 3D Tileset mandates a complex, one-way preprocessing pipeline that often abstracts or fragments the native semantic structures inherent to the source data. While 3D Tiles excels at rapid visual rendering, its rigid hierarchical nature sometimes lacks the flexibility required for on-the-fly, granular semantic querying and seamless data interoperability.

To address the distinct need for streaming native, semantically intact city models, CityJSON Text Sequences (CityJSONSeq) was formally introduced as part of the CityJSON 2.0 specification (Ledoux et al., 2024). Standard CityJSON, a lightweight, developer-friendly JSON encoding of the CityGML data model (Ledoux et al., 2019), suffers from a structural limitation: a client must download and parse the entire file into memory before a single building can be rendered. CityJSONSeq circumvents this bottleneck by decomposing the dataset into independent JSON Lines (JSONL). This streaming-friendly architecture allows web viewers to parse and render individual 3D features sequentially as they arrive over the network, drastically reducing initial latency, time-to-first-pixel, and capping memory overhead. Nevertheless, because CityJSONSeq relies on text-based encoding, deserializing massive JSON strings over HTTP still incurs considerable CPU overhead, posing a performance ceiling for truly massive or country-wide datasets.

Recognizing the computational limitations of text-based streaming, recent geospatial research has introduced FlatCityBuf, a cloud-optimized binary format designed to revolutionize 3D city model I/O (Baba et al., 2025). Built upon the FlatBuffers

serialization library, FlatCityBuf provides zero-copy binary encoding combined with advanced spatial indexing, adopting a philosophy similar to the highly successful 2D FlatGeobuf format which is an OGC Community Standard. This architecture allows clients to execute HTTP Range Requests to fetch only specific portions of a dataset based on spatial bounding boxes or feature IDs, completely bypassing the need to parse the entire file. Recent benchmarks demonstrate that FlatCityBuf can yield parsing speeds up to 30 times faster than standard CityJSON while consuming roughly a quarter of the memory footprint (Baba et al., 2025). By marrying the semantic richness of CityJSON with the raw I/O performance of a cloud-native binary structure, FlatCityBuf presents a formidable, high-performance alternative to both traditional text streaming and purely visual tiling solutions.

Despite the rapid concurrent development of these distinct approaches, HLOD visualization (OGC 3D Tiles 1.1), sequential text streaming (CityJSONSeq), and cloud-optimized binary retrieval (FlatCityBuf), there remains a conspicuous absence of comprehensive, comparative performance evaluations in the current literature. Web GIS architects and researchers currently lack empirical guidelines on how these formats stack up regarding network latency, client-side memory footprint, parsing overhead, and frames-per-second (FPS) during continuous client-side rendering. This paper bridges that critical research gap by conducting a rigorous web-based benchmarking of CityJSONSeq, FlatCityBuf, and OGC 3D Tiles 1.1. By systematically evaluating their streaming and rendering performances across varying network conditions and varying levels of 3D dataset complexity, this study provides a definitive framework for selecting the optimal 3D geospatial format for next-generation Urban Digital Twins.

2. Methodology

The experimental methodology involves a systematic evaluation of the three formats using five diverse, real-world 3D city models, assessing initial load times, data transfer rates over a controlled network, and sustained frame rates during simulated interactive navigation. Two experiments were designed around the two performance dimensions identified in the literature: (1) data loading and deserialization time, and (2) rendering and interactivity performance. Both experiments were implemented as browser-executed, in-house benchmarking harnesses rather than relying on third-party viewers, so that every measured stage of the pipeline: network transfer, parsing, CPU-side mesh construction, GPU upload, and frame presentation, and they could be timed independently and attributed to a specific processing stage rather than reported as an opaque aggregate

Five open, real-world LOD2 CityJSON 2.0 datasets were selected to span nearly three orders of magnitude in scale: Rotterdam, Den Haag, and Montreal (small to medium municipal building stocks), and New York and Zurich (large, city-wide building stocks), sourced in their original coordinate reference systems (Rotterdam: EPSG:28992; Den Haag: EPSG:7415; Montreal, New York, and Zurich did not declare a reference system in their CityJSON metadata, so EPSG:32618, EPSG:2263, and EPSG:2056 respectively were inferred from their coordinate value ranges, used only for the 3D Tiles ECEF transformation and not for any distance-dependent metric in this study). Table 1 summarizes their scale using two counts that are stable across all three formats and are not affected by the differing per-format feature-bundling conventions described below: the number of unique, welded vertices, and the number of triangulated surfaces in the final, welded mesh produced by the pipeline described in Section 2.2, confirmed (Section 3.2) to match to within rounding across all three formats for a given dataset.

Dataset	Vertex count	Triangle Count
Rotterdam	26,679	37,478
Den Haag	24,835	41,183
Montreal	32,242	62,993
New York	1,044,145	1,930,066
Zurich	3,565,748	4,928,827

Table 1. Test datasets: unique (welded) vertex count and triangulated surface count in the final mesh

Nominal "feature" or "object" counts, by contrast, are not directly comparable across formats and are not used as a complexity metric in this paper: CityJSONSeq and FlatCityBuf bundle a parent Building together with all of its child BuildingPart geometries into a single feature (one JSON Line, or one FlatBuffers CityFeature, per top-level building), whereas the 3D Tiles/glTF export used in this study (Section 2.2) enumerates every geometry-bearing CityObject, including each individual BuildingPart, as a separate glTF mesh feature. For Rotterdam (853), Montreal (294), and New York (23,777) the two counts coincide because every building in those datasets is a single, undivided Building object; for Den Haag (845 CityJSONSeq/FlatCityBuf features versus 1,991 glTF mesh features) and Zurich (52,834 versus 145,865), which contain multi-part buildings, the glTF feature count is substantially higher for identical underlying geometry. This is why Table 1 reports vertex and surface counts instead.

All CityJSONSeq (.jsonl) files were generated from the source CityJSON files using cjio's CityJSONSeq export. FlatCityBuf (.fcb) files were generated with the official fcb command-line tool (fcb_core/fcb_cli, v0.7.6), which converts CityJSON directly to the FlatBuffers-based binary encoding, including its packed R-tree spatial index. Since no equivalent open-source CityJSON-to-3D-Tiles/glTF converter with semantic metadata support was available, a dedicated conversion pipeline was implemented for this study: each building's LOD2 geometry (MultiSurface, CompositeSurface, or Solid boundaries) is triangulated using ear-clipping with a Newell-plane projection (Foley et al., 1990) step to correctly handle non-planar 3D polygons such as vertical wall surfaces, and re-projected to Earth-Centered, Earth-Fixed (ECEF) coordinates.

2.1 Data Loading and Deserialization Time

The process of transferring 3D geospatial data from a remote server to a client's graphics pipeline intrinsically involves two critical phases: network data loading (I/O) and client-side deserialization (parsing). In web-based Urban Digital Twins (UDTs), these phases often constitute the primary performance bottlenecks, dictating the time-to-first-pixel and overall application responsiveness. Deserialization overhead is particularly problematic for semantically rich city models, as the client application must reconstruct complex topological relationships, decode deep attribute tables, and map coordinate reference systems before executing any rendering commands Consequently, evaluating how modern web formats architect their payloads to mitigate these computational overheads is paramount to optimizing client-side performance.

This experiment measures the time to transmit a dataset from a server to a browser client and deserialize it into an in-memory representation, isolating network transfer time from parsing time.

A custom HTTP server was implemented to serve all files under a simulated, fixed bandwidth of 20 Mbps, a representative broadband condition, enforced via a single rate-limiting token bucket shared across all concurrent connections on the server—rather than one limiter per connection, so that formats issuing multiple parallel HTTP requests, notably FlatCityBuf's range-request architecture are not given an unrealistic aggregate-bandwidth advantage over formats that use a single connection. For each dataset and format, the browser client measured: (i) transfer time, from request start to full response body received, and (ii) parse time, the subsequent deserialization into an in-memory structure. All the results reported in Section 3.

2.2 Rendering and Interactivity Performance

This experiment measures the time to reach a rendered first frame and the sustained frame rate during continuous rendering, for each dataset, format, and graphics API. To isolate the graphics API itself as the only variable, rather than the relative optimization maturity of a third-party scene-graph library's WebGL versus WebGPU backends, two minimal renderers were implemented directly in raw WebGL2/GLSL and raw WebGPU/WGSL, without any scene-graph or rendering library. Both renderers share identical camera and projection matrices, an identical single-directional-light Lambertian shading model, and identical CPU-side mesh-construction code, so that any measured difference is attributable only to the two browser graphics APIs.

Because CityJSONSeq and FlatCityBuf transmit un-triangulated polygonal boundaries, an in-browser triangulator was implemented using ear-clipping with the same Newell-plane projection used in Section 2.2. This implementation includes the same per-object vertex-welding step and the same normal-free, shader-computed-shading approach as the glTF export pipeline, so that all three formats are rendered from meshes of matching vertex count, index count, and shading model; 3D Tiles/glTF, in contrast, arrives already triangulated and welded, and requires no equivalent client-side step. This means the welding cost itself is part of what is measured for CityJSONSeq and FlatCityBuf, paid by every client on every page load, whereas for 3D Tiles/glTF it was paid once, offline, during the Section 2.2 export and is not repeated in the browser. For each dataset, format, and graphics-API combination, four stages were timed independently. These stages are data loading, which involves fetch and format-specific deserialization; triangulation, encompassing CPU-side mesh construction and welding where applicable, which is always 0 ms for glTF; GPU upload, comprising vertex, index, and uniform buffer alongside pipeline creation; and time-to-first-rendered-frame. This final metric consists of GPU upload plus frame presentation only, deliberately excluding data loading and triangulation so that it reflects only the render backend. Following the first frame, the camera was orbited around the dataset's bounding-box centre at a fixed angular velocity, driven by wall-clock time rather than frame count, for five seconds per configuration, recording every frame interval to compute average and minimum FPS. All the results reported in Section 3.

3. Results

3.1 Data Loading and Deserialization

Table 2 reports file size and Table 3 reports transfer, parse, and total time at the simulated 20 Mbps bandwidth, for all five datasets and three formats.

Dataset	CityJSONSeq (MB)	FlatCityBuf (MB)	3D Tiles/glTF (MB)
Rotterdam	1.24	1.17	0.88
Den Haag	2.90	2.81	0.97
Montreal	4.60	4.82	1.23
New York	95.45	76.23	39.36
Zurich	247.12	189.65	119.16

Table 2. On-disk file size per format.

Dataset	Format	Transfer (ms)	Parse (ms)	Total (ms)
Rotterdam	CityJSONSeq	505.3	12.0	517.3
	FlatCityBuf	515.8	9.3	525.1
	3D Tiles/glTF	378.1	0.4	378.5
Den Haag	CityJSONSeq	1220.3	16.0	1236.3
	FlatCityBuf	1191.6	13.4	1205.0
	3D Tiles/glTF	402.1	0.4	402.5
Montreal	CityJSONSeq	1931.0	33.6	1964.6
	FlatCityBuf	2056.5	14.8	2071.3
	3D Tiles/glTF	510.2	0.6	510.8
New York	CityJSONSeq	40060.0	204.7	40264.7
	FlatCityBuf	33102.5	8.6	33111.1
	3D Tiles/glTF	16501.5	7.8	16509.3
Zurich	CityJSONSeq	103685.7	616.1	104301.8
	FlatCityBuf	81513.3	22.8	81536.1
	3D Tiles/glTF	49985.6	15.8	50001.4

Table 3. Experiment 1: transfer, parse, and total data-loading time at a simulated 20 Mbps shared bandwidth

With the welded glTF export, 3D Tiles/glTF now leads on every measure in Table 3. Its transfer time is the fastest of the three formats in every dataset, a direct consequence of Table 2's smaller payload; New York is the clearest case, where 3D Tiles/glTF's 39.36 MB transfers in 16,501.5 ms against CityJSONSeq's 95.45 MB at 40,060.0 ms and FlatCityBuf's 76.23 MB at 33,102.5 ms. Its parse time is also the fastest at every scale (0.4–15.8 ms), 26.2 to 56.0 times faster than CityJSONSeq's JSON parsing. This is independent of file size, and thanks to the format's own architecture. Even before the welding glTF, glTF had the biggest payload but had the fastest parse time as well in our initial tests. Because, our parseGLB() function only reads the binary container and binds the accessors directly to TypedArray views, meaning it doesn't even copy the data, it merely opens a "window" over the ArrayBuffer. There is no JSON tokenization, no per-object JavaScript object creation. Binding typed-array views to a pre-welded binary buffer requires no per-object JSON tokenization. jsonl, on the other hand, runs text.split('\n') + JSON.parse() for each line — thousands/millions of numbers need to be parsed as text and converted into JS objects. Similarly, fcb decodes FlatBuffers line by line into feature objects. Result, example Zurich: glb parse = 15.8ms, jsonl parse = 616.1ms (39x difference), fcb parse = 22.8ms. Consequently 3D Tiles/glTF's total loading time is the lowest of the three formats in every dataset, 1.4 to 3.9 times faster than CityJSONSeq and faster than or comparable to FlatCityBuf.

FlatCityBuf's own parse-time trend from before welding is unchanged, since welding only affects the glTF pipeline: its parse time is comparable to, or slower than, CityJSONSeq's on the three smallest datasets (e.g. Montreal: 14.8 ms vs. 33.6 ms is actually faster here, but Den Haag: 13.4 ms vs. 16.0 ms is only marginally so, and on Rotterdam FlatCityBuf's 9.3 ms is not

decisively faster than CityJSONSeq's 12.0 ms once measurement noise at this scale is considered), but drops sharply on the two largest datasets: 8.6 ms and 22.8 ms on New York and Zurich respectively, 23.8 and 27.0 times faster than CityJSONSeq at the same scale. FlatCityBuf's total time remains competitive with, and on New York and Zurich faster than, CityJSONSeq (1.22 and 1.28 times faster), by combining a smaller file size with its large-scale parsing advantage, but no longer wins outright now that 3D Tiles/glTF is both smaller and faster to parse than FlatCityBuf itself.

3.2 Rendering and Interactivity Performance

Table 4 (see the appendix) reports the four Experiment 2 timing stages and sustained frame rate for every dataset, format, and graphics API. Triangle counts (final column) confirm that all three formats produced near-identical meshes for a given dataset, validating the cross-format comparison.

Four findings stand out. First, triangulation cost is zero for 3D Tiles/glTF in every single trial, while CityJSONSeq and FlatCityBuf require a client-side triangulation-and-welding step that grows with dataset size, reaching 3.2–3.7 seconds on Zurich (over four million triangles); this cost is inherent to deduplicating vertices against a per-object lookup while also ear-clipping every polygon, work that CityJSONSeq and FlatCityBuf must repeat in every browser session, while 3D Tiles/glTF pays it once, offline, during export. Second, this makes 3D Tiles/glTF reach a fully render-ready state dramatically faster at scale: on Zurich, its total pipeline time (99.7–183.0 ms) is 25.0–47.1 times faster than CityJSONSeq (4570.6–4692.6 ms) and 20.4–36.9 times faster than FlatCityBuf (3673.8–3738.4 ms); on New York (total 42.1–55.6 ms) the gap is 29.4–37.8 times over CityJSONSeq and 21.6–30.1 times over FlatCityBuf. Third, with the corrected decoder (Section 2.4), FlatCityBuf is now faster than CityJSONSeq at every single stage, not only parsing: its data-loading time is 9–42% higher than CityJSONSeq's on the three smallest datasets but drops to 0.47 times CityJSONSeq's on New York and Zurich (e.g. Zurich: 436.3–468.1 ms versus CityJSONSeq's 874.3–933.9 ms, roughly 1.9–2.1 times faster), the same small-scale-overhead/large-scale-advantage pattern already seen in Experiment 1; its triangulation time is also modestly lower than CityJSONSeq's at every scale (e.g. Zurich: 3219.7–3237.7 ms versus 3613.1–3720.0 ms), plausibly because earcut and the welding lookup index directly into FlatBuffers' typed-array-backed boundary data rather than into plain JavaScript arrays produced by JSON.parse. Fourth, once geometry reaches the GPU, average sustained frame rate remains essentially identical across all formats and both graphics APIs, pegged near 120 (the test environment's refresh-rate ceiling) in all 30 configurations regardless of dataset size or triangle count, indicating GPU throughput was never the bottleneck for scene even at 4.9 million triangles; minimum FPS shows occasional single-frame stalls (27 of 30 configurations at or above 95.2 FPS, three dipping lower: Rotterdam/3D Tiles-glTF/WebGPU at 58.5, New York/FlatCityBuf/WebGL2 at 95.2, Zurich/3D Tiles-glTF/WebGPU at 97.1), but these did not repeat the same configurations that dipped in an earlier run of this experiment, indicating environmental noise rather than a reproducible, format- or API-specific weakness (Section 5).

4. Discussion

An initial, naive implementation that stored a fresh vertex per polygon corner and a persisted per-vertex normal made 3D Tiles/glTF the largest payload of the three formats (1.19–2.22 times larger than CityJSONSeq) and, despite its zero triangulation cost, left it transferring more data than either CityJSON-derived format at every scale. Applying per-object vertex welding and replacing the stored normal with a shader-computed flat normal, bringing the glTF pipeline in line with CityJSON's own convention of storing each vertex once and no normals at all, reduced the exported payload by 65–83% and reversed the size ranking entirely: 3D Tiles/glTF became the smallest format in every dataset (Table 2). This result underscores that the widely repeated claim that pre-triangulated binary formats are "larger but faster" rests on an encoding choice, not a structural property of the format; an unwelded, normal-carrying export was actively working against 3D Tiles/glTF's own architectural advantage in this study's earlier iteration, and the pipeline shows that when a glTF exporter matches CityJSON's storage discipline, there is no size penalty to trade away in the first place.

3D Tiles/glTF is unambiguously the strongest performer across both experiments in this study, winning every measured stage: smallest payload and fastest transfer and parse time in Experiment 1 (Table 3), and by a wide margin the fastest to reach a rendered first frame in Experiment 2 (Table 4), 25.0–47.1 times faster than CityJSONSeq and 20.4–36.9 times faster than FlatCityBuf on New York and Zurich. 3D Tiles/glTF alone requires no client-side triangulation, because that work, now including vertex welding, Section 2.2, is done once, offline, during export, rather than being repeated by every client on every page load. The welding fix in fact widened this gap rather than narrowing it, because deduplicating vertices against a per-object lookup is itself non-trivial CPU work: CityJSONSeq's and FlatCityBuf's client-side triangulation time on Zurich rose from roughly 1.2–1.6 seconds (unwelded) to 3.1–3.3 seconds (welded to match glTF's vertex count) once their in-browser triangulator was extended to perform the same deduplication 3D Tiles/glTF's exporter performs offline. This is the paper's clearest architectural lesson: any CPU-side mesh-preparation cost that a pre-processed, pre-triangulated format avoids must be paid somewhere by the two formats that stream native, un-triangulated CityJSON geometry, and paying it in the browser, once per client, is categorically worse for time-to-first-render than paying it once on the server, however efficiently the in-browser step is implemented.

FlatCityBuf's position changes correspondingly. Its zero-copy architecture still delivers a genuine, literature-consistent deserialization speed-up over CityJSONSeq, 23.8 and 27.0 times faster on New York and Zurich, in the same order of magnitude as the "up to 30 times faster" figure cited in Section 1—but this advantage is scale-dependent rather than universal: on the three smaller datasets (853 to 1,991 features), FlatCityBuf's parse time is comparable to, and on Rotterdam marginally slower than, plain JSON.parse on CityJSONSeq, consistent with a fixed per-call overhead at the JavaScript/WebAssembly boundary that only amortizes away once the dataset is large enough. This deserialization advantage no longer translates into an overall win in Experiment 1 even at scale, now that 3D Tiles/glTF is both smaller and faster to parse than FlatCityBuf itself (Table 3). The same small-scale-overhead, large-scale-advantage pattern reappears in Experiment 2's data-loading stage once FlatCityBuf's decoder is made fair to its own architecture (Section 2.4): decoding straight into the triangulator's input representation, rather than via an intermediate CityJSON object graph, makes FlatCityBuf's data-loading time 9–42% higher than CityJSONSeq's on the three smallest datasets but roughly half of CityJSONSeq's on New York and Zurich (Table 4). Because FlatCityBuf, like CityJSONSeq, still transmits un-triangulated polygonal boundaries, it inherits a comparable client-side triangulation burden in Experiment 2, 3219.7–3237.7 ms on Zurich, modestly but consistently faster than CityJSONSeq's

3613.1–3720.0 ms, plausibly because triangulating and welding directly against FlatBuffers' typed-array-backed boundary indices is cheaper than doing the same against the plain JavaScript arrays JSON.parse produces. With a fair decoder, then, FlatCityBuf is faster than CityJSONSeq at every Experiment 2 stage, not only parsing, but the gap to 3D Tiles/glTF remains an order of magnitude because triangulation, not data loading, is the dominant cost at scale for both un-triangulated formats. FlatCityBuf and 3D Tiles/glTF therefore are not simple substitutes: one is the strongest choice for fast, semantically-queryable data access; the other is the strongest choice for fast, GPU-ready visualization; a workflow that needs both would benefit from a hybrid that pairs FlatCityBuf's indexed attribute access with a pre-triangulated visual payload, a possibility Section 5 returns to.

The pronounced WebGL/WebGPU frame-rate gap referenced in Section 1 (on the order of 12–25 FPS versus 90–120 FPS, reported elsewhere for dense, tiled 3D Tiles 1.1 scenes) did not manifest in this benchmark: both APIs sustained the display refresh-rate ceiling on average across every single-mesh test, including the 4.9-million-triangle Zurich dataset, and the isolated minimum-FPS dips observed (Section 3.2) did not repeat the same dataset/format/API combinations across successive runs of this experiment, indicating environmental noise rather than a systematic pattern tied to either API. This is consistent with the qualification, present in the broader literature, that once geometry resides in GPU buffers, rendering throughput is governed by the graphics API's baseline efficiency rather than by the source data format, and extends that observation to note that without a genuine multi-tile, HLOD, or draw-call-batching workload of the kind 3D Tiles 1.1 is specifically designed to orchestrate, WebGPU's architectural advantages over WebGL2 (lower per-draw-call overhead, more efficient multi-threaded command encoding) have no opportunity to manifest against a single, unculled draw call in either format.

5. Conclusion

This study empirically benchmarked CityJSONSeq, FlatCityBuf, and a OGC 3D Tiles 1.1 (glTF) formats across five real-world city models spanning nearly three orders of magnitude in scale, using a bandwidth-controlled data-loading experiment and a from-scratch, dual WebGL2/WebGPU rendering harness designed to isolate the graphics API from confounding scene-graph library effects. Correcting the glTF export to weld shared vertices and drop stored normals, matching CityJSON's own storage discipline rather than adding a mesh-encoding overhead the format does not require, reduced its file size by 65–83% and made 3D Tiles/glTF the smallest, fastest-to-transfer, fastest-to-parse, and by far the fastest-to-render format in every dataset tested, reaching a render-ready first frame up to 47.1 times faster than CityJSONSeq and 36.9 times faster than FlatCityBuf on the largest dataset. This is the paper's central, and initially counter-intuitive, finding: the client-side triangulation that CityJSONSeq and FlatCityBuf must still perform grew more expensive once it was extended to also perform the vertex welding that 3D Tiles/glTF's exporter had already done offline. FlatCityBuf retains a genuine, literature-consistent deserialization advantage over CityJSONSeq at large scale (23.8–27.0 times faster), muted on small datasets by fixed WebAssembly overhead; once decoded fairly to its own zero-copy design (Section 2.4), this advantage was found to extend across every Experiment 2 stage as well, with FlatCityBuf's data loading roughly twice as fast and its triangulation modestly faster than CityJSONSeq's at the largest scale—but because FlatCityBuf, like CityJSONSeq, still requires the same client-side triangulation 3D Tiles/glTF avoids entirely, this narrows rather than closes the order-of-magnitude gap to a pre-triangulated visual format. Contrary to widely cited benchmarks, neither graphics API showed a systematic average- or minimum-frame-rate advantage over the other in this single-mesh, non-tiled test.

Future work should investigate a hybrid workflow pairing FlatCityBuf's indexed, queryable attribute access with a pre-triangulated glTF visual payload, combining each format's demonstrated strength rather than treating them as competing substitutes.

Acknowledgements

The authors would like to thank the Scientific and Technological Research Council of Türkiye (TÜBİTAK) for the financial support provided under the BİDEB 2224-A Travel Fellowship for International Scientific Events Program (Grant number: 1919B022606053).

References

Baba, H., Ledoux, H., & Peters, R. (2025). FlatCityBuf: A new cloud-optimised CityJSON format. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *48*, 17-24.

Batty, M. (2018). Digital twins. *Environment and Planning B: Urban Analytics and City Science*, 45(5), 817–820.

Biljecki, F., Stoter, J., Ledoux, H., Zlatanova, S., & Çöltekin, A. (2015). Applications of 3D city models: State of the art review. *ISPRS International Journal of Geo-Information*, 4(4), 2842–2889.

Chaturvedi, K., Yao, Z., & Kolbe, T. H. (2015). Web-based exploration of and interaction with large and deeply structured semantic 3D city models using HTML5 and WebGL. *In 34th Annual Conference of the German Society of Photogrammetry, Remote Sensing and Geoinformation (DGPF)*.

Foley, J. D., van Dam, A., Feiner, S. K., & Hughes, J. F. (1990). Computer graphics: Principles and practice (2. Baskı). Addison-Wesley.

Kolbe, T. H. (2009). Represents and exchanges 3D city models with CityGML. *In 3D geo-information sciences* (pp. 15–31). Springer, Berlin, Heidelberg.

Ledoux, H., Arroyo Ohori, K., Kumar, K., Dukai, B., Hintz, D., & Stoter, J. (2019). CityJSON: A compact and easy-to-use encoding of the CityGML data model. *Open Geospatial Data, Software and Standards,* 4(1), 1–12.

Ledoux, H., et al. (2024). CityJSON 2.0 Specification. CityJSON.org. (Sourced from the provided introduction text)

Mao, B., & Ban, Y. (2011). Online visualization of 3D city model using CityGML and X3DOM. Cartographica: *The International Journal for Geographic Information and Geovisualization,* 46(2), 109–114.

OGC. (2022). OGC 3D Tiles 1.1 Specification. Open Geospatial Consortium. (Sourced from the provided introduction text)

Vitalis, S., Arroyo Ohori, K., & Stoter, J. (2020). CityJSON in the browser: Enabling 3D web GIS with WebGL. *ISPRS Annals*

of the Photogrammetry, Remote Sensing and Spatial Information Sciences, 6(4/W1), 171–178.

Usta, Z., Akın, A.T., Ohori, K.A., & Stoter, J., 2026. Visualization of Urban Digital Twins on the web with attribute-driven adaptive tiling. *Environmental Modelling & Software*, 197, 106863. doi.org/10.1016/j.envsoft.2026.106863.

Appendix.

Dataset	Format	API	Data load (ms)	Triangulate (ms)	GPU upload (ms)	First frame, render-only (ms)	Total (ms)	Avg FPS	Min FPS	Triangles
Rotterdam	CityJSONSeq	WebGL2	14.8	31.2	2.1	5.9	51.9	120.1	106.4	37,798
	CityJSONSeq	WebGPU	13.5	37.0	1.7	3.7	54.2	120.0	106.4	37,798
	3D Tiles/glTF	WebGL2	6.7	0.0	2.6	7.4	14.1	120.1	106.4	37,478
	3D Tiles/glTF	WebGPU	5.3	0.0	4.9	9.8	15.1	119.8	58.5	37,478
	FlatCityBuf	WebGL2	21.0	33.1	1.4	9.0	63.1	120.0	107.5	37,798
	FlatCityBuf	WebGPU	14.5	37.7	2.1	2.7	54.9	120.0	106.4	37,798
Den Haag	CityJSONSeq	WebGL2	26.1	40.5	2.0	9.3	75.9	120.1	106.4	41,183
	CityJSONSeq	WebGPU	27.1	41.6	2.0	3.0	71.7	120.0	106.4	41,183
	3D Tiles/glTF	WebGL2	5.9	0.0	2.5	6.6	12.5	120.3	106.4	41,183
	3D Tiles/glTF	WebGPU	10.8	0.0	7.8	10.0	20.8	120.0	106.4	41,183
	FlatCityBuf	WebGL2	28.4	46.8	1.6	2.5	77.7	120.2	106.4	41,183
	FlatCityBuf	WebGPU	22.9	48.8	2.1	8.6	80.3	120.0	106.4	41,183
Montreal	CityJSONSeq	WebGL2	27.4	52.9	1.5	9.3	89.6	120.0	107.5	63,265
	CityJSONSeq	WebGPU	37.9	61.7	2.3	3.7	103.3	120.0	107.5	63,265
	3D Tiles/glTF	WebGL2	4.2	0.0	1.5	10.3	14.5	120.0	106.4	62,993
	3D Tiles/glTF	WebGPU	10.9	0.0	8.7	9.5	20.4	120.0	106.4	62,993
	FlatCityBuf	WebGL2	34.1	63.7	1.5	9.8	107.6	120.3	106.4	63,265
	FlatCityBuf	WebGPU	19.8	59.6	1.4	7.8	87.2	120.0	106.4	63,265
New York	CityJSONSeq	WebGL2	351.2	1219.6	9.1	19.6	1590.4	120.0	106.4	1,930,067
	CityJSONSeq	WebGPU	322.5	1298.0	12.9	13.2	1633.7	120.0	106.4	1,930,067
	3D Tiles/glTF	WebGL2	34.0	0.0	5.5	8.1	42.1	120.0	106.4	1,930,066
	3D Tiles/glTF	WebGPU	40.9	0.0	14.1	14.7	55.6	120.0	106.4	1,930,066
	FlatCityBuf	WebGL2	163.9	1087.5	5.1	15.4	1266.8	120.0	95.2	1,930,067
	FlatCityBuf	WebGPU	138.1	1046.3	13.8	14.4	1198.8	120.0	106.4	1,930,067
Zurich	CityJSONSeq	WebGL2	933.9	3720.0	30.0	38.7	4692.6	120.0	106.4	4,928,825
	CityJSONSeq	WebGPU	874.3	3613.1	82.7	83.2	4570.6	120.0	106.4	4,928,825
	3D Tiles/glTF	WebGL2	78.2	0.0	19.4	21.5	99.7	120.0	106.4	4,928,827
	3D Tiles/glTF	WebGPU	138.6	0.0	43.8	44.4	183.0	119.9	97.1	4,928,827
	FlatCityBuf	WebGL2	436.3	3219.7	10.0	17.8	3673.8	120.0	107.5	4,928,825
	FlatCityBuf	WebGPU	468.1	3237.7	32.0	32.6	3738.4	120.2	106.4	4,928,825

Table 4. Experiment 2: rendering pipeline timing and sustained frame rate, using the welded, normal-free pipeline for all three formats. First frame, render-only" excludes data loading and triangulation by design.