Incremental Snapshot Management for Real-Time Spatial Scene Graphs in AR and VR

Seungjae Lee[1], Woontack Woo[1,2,*]

[1]KAIST UVR Lab, 291 Daehak-ro, Yuseong-gu, 34141, Daejeon, Republic of Korea
[2]KAIST KI-ITC Augmented Reality Research Center, 291 Daehak-ro, Yuseong-gu, 34141, Daejeon, Republic of Korea

Abstract

AR and VR applications often require access to more than the latest scene graph. Replay, restoration, comparison, and temporal queries depend on how a spatial scene evolves over time. This paper studies the downstream management problem that begins after complete 3D scene-graph states have already been produced. We present an incremental snapshot manager that filters unchanged states, stores node- and edge-level differences in an append-only JSONL history, and materializes accepted states and transitions offline from an externally supplied initial scene. A Unity/Python prototype validates the approach on four controlled workloads totaling 126 seconds, producing 61 accepted transition groups and 155 operations while reconstructing every final state. The results show that incremental histories provide temporal detail efficiently only when scene changes are sufficiently sparse, and that higher capture rates preserve more intermediate transitions without affecting final-state recovery in the tested workloads.

Keywords

Dynamic scene graph, 3D scene graph, Snapshot management, Temporal reconstruction, AR, VR

1. Introduction

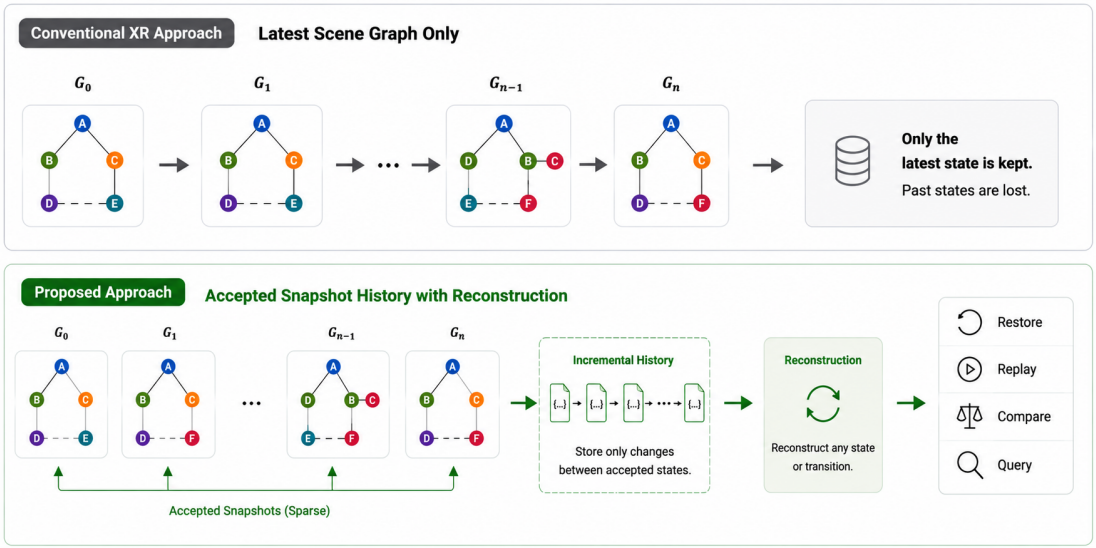

Figure 1: Overview of the proposed temporal scene manager. A conventional XR runtime exposes only the latest scene graph, making previous states unavailable once the scene evolves. The proposed manager captures sparse accepted scene-graph snapshots, stores only the changes between them as an external-baseline-plus-incremental history, and reconstructs temporal scene states for restoration, replay, comparison, and temporal queries.

AR and VR systems increasingly expose their environments as structured scene graphs. Nodes may

APMAR'26: The 18th Asia-Pacific Workshop on Mixed and Augmented Reality, Aug. 3-5, 2026, Taipei, Taiwan
*Corresponding author.
sjlee_7104@kaist.ac.kr (S. Lee); wwoo@kaist.ac.kr (W. Woo)
0009-0005-6459-1756 (S. Lee); 0000-0002-5501-4421 (W. Woo)
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-09-17

CEUR-WS.org/Vol-4267/paper15.pdf

represent physical objects, virtual content, users, anchors, or zones; edges encode relations such as containment, adjacency, attachment, or support. Prior work shows how such graphs can be constructed and updated from spatial observations [1, 2]. This paper focuses on the next step: managing the stream after those graphs already exist.

Latest-state storage is enough for immediate rendering, but not for replay, restore, comparison, or temporal query [3, 4, 5, 6]. Storing every complete graph is simple but repeats unchanged content and leaves downstream consumers to infer transitions themselves. Figure 1 makes this contrast explicit: the top row keeps only the latest runtime state, whereas the proposed manager below preserves accepted states as incremental history that can later be reconstructed for restore, replay, comparison, and query.

This gap matters because XR scene graphs are often consumed by more than one process. A live runtime may need only the newest state for rendering, but review tools, authoring utilities, analytics processes, or shared-space services may later need to know what changed, when it changed, and how one accepted state led to the next [7, 8, 9, 3, 4, 10]. Without an explicit history layer, each consumer must either keep its own copy of the stream or recompute differences from stored full states.

For example, in a collaborative XR authoring session, one user may move a chair, another may rotate a lamp, and a third may add a table to the shared scene. A latest-state runtime can render the final arrangement, but a review or replay tool needs the accepted transitions: which object moved, which property changed, and which entity was introduced. The manager stores those accepted changes directly rather than storing every complete graph again.

Our scope is deliberately narrow. We assume an upstream AR/VR runtime, shared-space service, simulation, or authoring tool supplies complete renderer-independent 3D scene graphs. We do not perform tracking, mapping, or scene-graph construction. The manager compares accepted graph states, records node- and edge-level changes after an externally supplied initial graph, and materializes temporal histories offline.

We implement the manager with a Unity recorder and Python materializer. The recorder filters unchanged states, computes differences from the last accepted graph, and appends JSONL operation records. The materializer checks order, groups records into accepted transitions, reconstructs final state, and computes aggregate differences. As summarized in Figure 1, Unity then applies those transitions to GameObjects.

The contributions of this work are:

- an incremental snapshot-management pipeline that transforms complete XR scene-graph states into durable, replayable transition histories; and
- a feasibility study of the resulting histories under varying change density and capture rates, highlighting the trade-off between temporal resolution and storage overhead.

2. Related Work

2.1. Scene graphs and temporal representations

Three-dimensional scene graphs represent entities and relations in immersive environments. 3DSSG and SceneGraphFusion show how such graphs can be constructed from spatial observations and updated over time [1, 2]. Dynamic scene-graph work adds metric, semantic, and agent-centric structure to evolving environments [11, 12, 13]. Related temporal methods such as Action Genome, STTran, TEMPURA, and Neural Scene Graphs model or infer how entities and relations evolve across time [14, 15, 16, 17], while spatio-temporal mapping systems such as Khronos extend temporal modeling into dynamic metric-semantic environments [18]. Dense dynamic representations such as 3D Gaussian Splatting preserve geometry and appearance at a different abstraction level [19]. Taken together, these lines of work address how spatial structure is estimated, represented, or predicted as a scene evolves. Some emphasize semantic graph construction from observations, others encode temporal dependencies between entities and relations, and still others preserve dense appearance and geometry for dynamic views.

2.2. XR history and incremental storage

XR authoring and replay work motivates keeping spatial history accessible after initial creation. RealitySketch, RealityCrafter, and collaborative VR authoring systems create evolving spatial arrangements whose intermediate states matter for review and reuse [7, 8, 9]. Experience Graphs and Origin-Centric Graphs show how graph history can support mixed-reality replay and adaptive playback [3, 4]. SceneGit and VRGit make the history problem explicit through diffing, comparison, commits, and merging in 3D or VR authoring workflows [5, 6]. Our prototype does not provide those user-facing replay or revision interfaces; it preserves the lower-level ordered snapshot history on which they could build. These systems also differ in when temporal structure is introduced. Replay and authoring tools typically consume already-recorded histories or expose user-facing revision boundaries, whereas our manager captures accepted runtime changes before commits, branches, or editing milestones are defined.

Large or unbounded histories also need an append-friendly format. RFC 7464 and JSON Lines support incremental production and recovery of JSON sequence data [20, 21]. The design also resembles event-sourcing and CQRS in that an append-oriented history can reconstruct state while derived read products serve different access needs [22, 23]. The models are not equivalent, however: our records are differences derived from accepted complete scene states rather than authoritative domain events.

3. Proposed System

3.1. Scope and architecture

We assume that an upstream AR runtime, shared-space service, simulation, or authoring tool publishes complete renderer-independent 3D scene graphs with stable node identities. The manager does not perform tracking, mapping, or scene-graph construction. Instead, it accepts supplied graph states, filters unchanged ones, records node- and edge-level differences after an externally supplied initial scene, and exposes temporal products offline for restore, transition replay, comparison, debugging, or temporal query [3, 4, 5, 6].

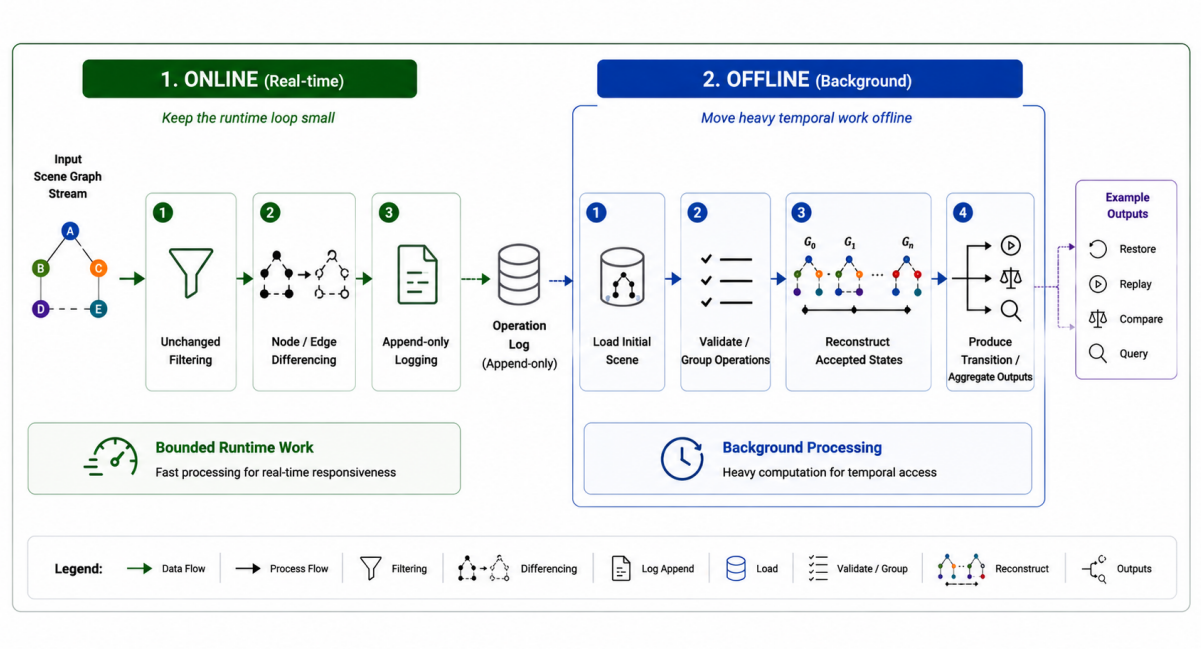

Figure 2: Online/offline management path. The runtime path performs bounded filtering, node- and edge-level differencing, and append-only logging, while background processing loads the external initial scene, validates and groups operations, reconstructs accepted states, and produces example outputs for XR applications.

Figure 2 shows the system boundary. The online path does only the work needed to preserve a new

accepted state: signature construction, unchanged-state filtering, differencing against the last accepted graph, and append-only logging. The offline path validates ordering, groups operations into accepted transition groups, reconstructs graph states, and produces derived products such as transition replay, aggregate comparison, and temporal queries. This split keeps long-history reconstruction and other heavy temporal access outside the real-time XR loop. In the current prototype, a state is accepted only when its canonicalized graph content differs from the last accepted state; otherwise the sample is skipped without producing log records.

Incremental management depends on identity continuity. A changed pose should remain a property patch to the same node, whereas a genuinely new entity becomes an added node. The current manager expects stable node identifiers within one history and edge identities derived from source, relation type, and target. Figure 2 reflects this division of labor by showing that identity-preserving differencing remains in the bounded online path, while reconstruction and other temporal products remain offline.

3.2. Scene-graph schema

The manager assumes that the external initial scene follows the public SceneGraph schema draft provided by the kaist-arrc/SceneGraph repository [24]. Baseline scene documents use six top-level fields: `schema`, `version`, `metadata`, `nodes`, `edges`, and `extensions`. In this prototype, `schema` must be `scenegraph`. Each node stores `id`, `type`, and a free-form `properties` object, while each edge stores `id`, `type`, `source`, `target`, and `layer`. The prototype uses object-centered node properties such as `asset`, `position`, `rotation`, `scale`, and `name`. Rotation is stored as a normalized quaternion `[x,y,z,w]` so that the recorded state is directly reusable during materialization and replay.

Table 1
Scene-graph and incremental-log schema used in the prototype.

Structure	Fields
Scene graph	`schema`, `version`, `metadata`, `nodes`, `edges`, `extensions`
Node	`id`, `type`, `properties`
Edge	`id`, `type`, `source`, `target`, `layer`
Log record	`seq`, `time`, `snapshot_id`, op
Node patch	`node_id`, optional `node_type`, sparse `properties`, `changed_properties`, `remove_properties`
Structural payload	full `node` for `add_node`; full `edge` for `add_edge` and `remove_edge`

Table 1 summarizes the two linked representations. The baseline scene graph is intentionally simple: nodes and edges carry the semantic content, while the append-only log carries only the information needed to recover accepted changes. During differencing, nodes are matched by stable `id`, and edges are matched by the triplet (`source`, `type`, `target`). This keeps the accepted-state interface compact while allowing sparse node-property patches and explicit relation changes.

3.3. Incremental snapshot model

Each history begins from an externally supplied initial scene rather than a manager-generated checkpoint. Figure 3 visualizes this structure: starting from external G_0, each accepted transition is stored as one grouped boundary containing several append-only node- and edge-level records. For every accepted state, the manager emits only five operation types: `update_node`, `add_node`, `remove_node`, `add_edge`, and `remove_edge`. Node updates store only changed properties together with explicit `changed_properties` and `remove_properties` lists, so omitted fields are not treated as authoritative

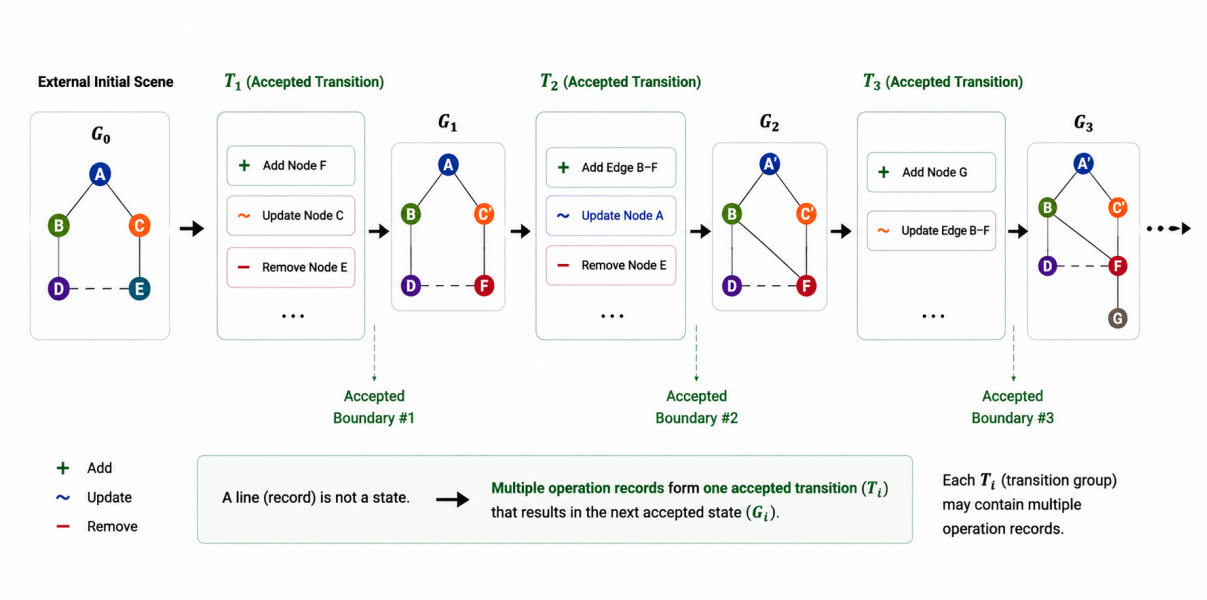

Figure 3: Accepted transition groups and appendable operation records. Starting from an externally supplied initial scene G_0, each accepted transition T_i is represented as a group of node- and edge-level operations. Records in the same group share an accepted-state boundary and together produce the next accepted state G_i.

defaults. In the current Unity recorder, transform properties include position, normalized quaternion rotation `[x,y,z,w]`, and scale.

These operations intentionally separate identity continuity from structural change. `update_node` modifies an existing node that remains the same entity across accepted states, whereas `add_node` and `remove_node` mark true appearance or disappearance in the accepted graph. The edge operations similarly add or remove relations between existing nodes without implying that the nodes themselves are replaced. This distinction lets the materializer treat pose or attribute edits as continuity-preserving patches rather than as delete-and-recreate events.

We denote the accepted reconstructed states as G_i and the grouped operation set between consecutive accepted states as an accepted transition T_i.

Each JSONL line contains a sequence number, timestamp, snapshot identifier, and one operation [20, 21]. Records that belong to the same accepted transition share the same snapshot identifier and timestamp, so the durable log distinguishes one appendable operation record from the larger transition group T_i that together produces the next accepted state G_i. This distinction matters because temporal consumers reconstruct accepted-state boundaries from grouped records rather than treating each line as an independent scene state or transition.

Operational granularity is therefore finer than accepted-state granularity. A single accepted transition may contain several records, such as multiple node updates together with one relation removal, while an unchanged sampled graph produces no transition group at all. This is also visible in Figure 3, where several operation records belong to the same accepted boundary before yielding the next reconstructed state. The log preserves only accepted change boundaries, but within each boundary it still exposes the node- and edge-level edits needed for reconstruction, inspection, or comparison.

3.4. Durability and temporal access

The recorder exports the external initial scene separately and appends each completed operation immediately. Sequence numbers must increase monotonically; unsupported operations or updates to missing nodes are rejected during load. If recording ends after only part of the final JSON object reaches storage, the loader treats a malformed final line as recoverable and ignores it, while malformed earlier lines remain errors. This yields a useful record-level recovery guarantee, although a multi-operation

transition group is not yet transactionally committed.

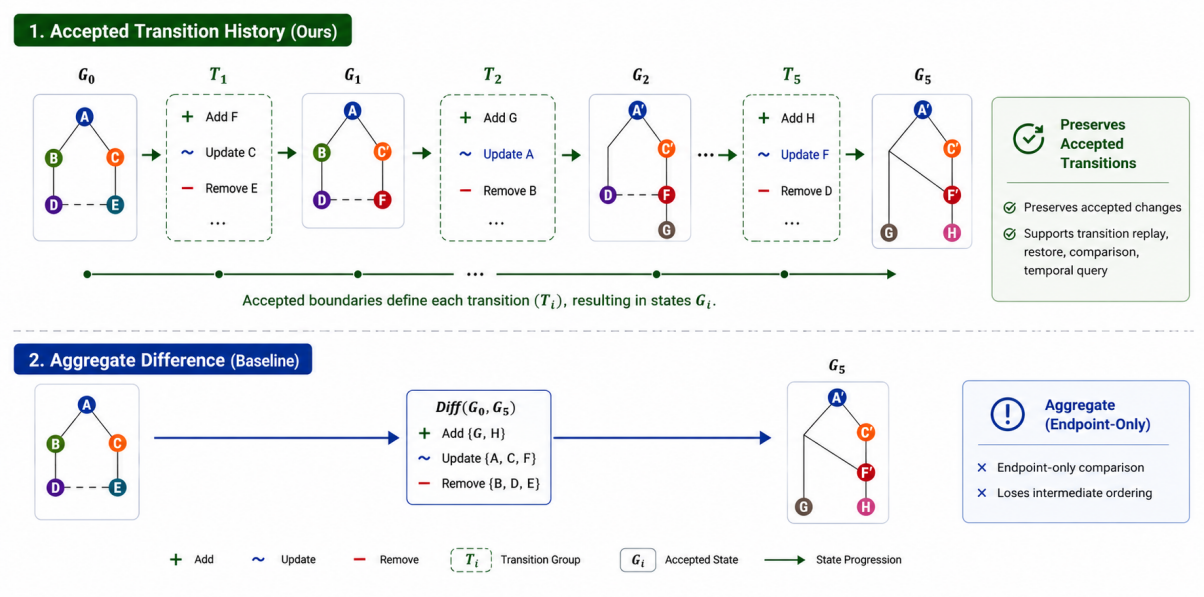

Figure 4: Accepted-transition history preserves accepted changes and their ordering, whereas the aggregate difference compares only the endpoints and loses intermediate ordering.

Offline materialization starts with a deep copy of the external initial graph and applies each accepted transition group in temporal order. Figure 4 shows the resulting distinction: the accepted-transition history preserves each intermediate change boundary, whereas the aggregate difference collapses the same evolution into one endpoint-only comparison. The processor therefore exports two complementary products: an accepted-transition sequence that preserves every accepted change boundary and an aggregate difference that compares only the beginning and end states.

The current prototype pairs a Unity online recorder with a Python offline processor. The recorder runs in Unity 6000.3.10f1, obtains complete graph states from a controlled scene extractor used only as an upstream stand-in, performs automatic or manual capture, and writes JSONL operations. The Python processor reconstructs the final graph, writes an ordered accepted-transition sequence, and writes the aggregate comparison using only standard-library components. Unity can then load the external initial scene and apply each transition to renderer objects. In Figure 4, this corresponds to the top path that preserves the ordered accepted changes rather than only the endpoint difference. Transition replay therefore demonstrates that the stored changes are explicit enough for an interactive XR consumer, while continuous motion between accepted states remains a presentation concern rather than a stored signal [3, 4].

4. Feasibility Study

4.1. Study design

This section evaluates feasibility rather than deployed XR performance. We measure accepted states, skipped unchanged states, operations per transition, materialization success, transition-replay generation, reconstruction latency, and incremental-log size relative to accepted full states. The four controlled `StreamingAssets` workloads—*ShortSparse*, *MediumStaggered*, *LongComplex*, and *OriginalChoreography*—total 126 seconds and all begin from seven-node initial scenes. Using a nominal two-second capture interval, we infer observation opportunities and skipped unchanged samples from workload duration, then apply the official Python materializer to every log. The workload set intentionally spans sparse, staggered, extended, and continuously changing motion patterns so that the study covers more than one change-density regime.

The evaluation is organized around three feasibility questions: whether the manager reconstructs final graphs from accepted operations, how latency and storage vary with change density, and how capture rate affects the amount of intermediate transition history that is preserved.

The study tests the same append-and-materialize path under different temporal patterns rather than approximating one deployed application, so the emphasis is on internal consistency across filtering, transition grouping, reconstruction, and downstream reuse.

Table 2 summarizes the workload structure used by the motion driver. *ShortSparse* isolates three brief object motions with two explicit unchanged windows. *MediumStaggered* alternates single-object motion, paired-object motion, and hold intervals. *LongComplex* extends that pattern over a longer session with more alternating active and unchanged segments. *OriginalChoreography* keeps all objects active throughout and therefore provides the densest change stream without designed pause windows. These workloads approximate common XR authoring patterns at a controlled scale: isolated object manipulation, staggered multi-object editing, longer sessions with pauses, and dense continuous manipulation.

Table 2
Stage structure of the four controlled workloads.

Workload	Stage	Sec.	Active objects	Unchanged
ShortSparse (18 s)	Orb transfer	0–4	orb	No
	Unchanged observation window	4–8	none	Yes
	Pillar-only precession	8–11	pillar	No
	Second unchanged window	11–14	none	Yes
	Block-only pulse	14–18	block	No
MediumStaggered (30 s)	Left platform calibration	0–3	left platform	No
	Calibration hold	3–6	none	Yes
	Orb orbit survey	6–12	orb	No
	Pillar and block pair	12–17	pillar + block	No
	Pair-state hold	17–21	none	Yes
	Orb slalom	21–27	orb	No
	Right platform calibration	27–30	right platform	No
LongComplex (54 s)	Orb and left platform launch	0–7	orb + left platform	No
	Launch-state hold	7–11	none	Yes
	Pillar precession	11–16	pillar	No
	Block pulse train	16–22	block	No
	Intermediate hold	22–27	none	Yes
	Orb orbital scan	27–35	orb	No
	Pillar and block interaction	35–42	pillar + block	No
	Relation transition hold	42–46	none	Yes
	Orb and right platform landing	46–54	orb + right platform	No
Original Choreography (24 s)	Original launch	0–4	all objects	No
	Original central orbit	4–8	all objects	No
	Original slalom	8–12	all objects	No
	Original pillar inspection	12–16	all objects	No
	Original dive	16–20	all objects	No
	Original landing	20–24	all objects	No

4.2. Managed histories

Table 3 summarizes the four managed histories. Across 67 nominal observation opportunities, 61 changed transition groups are recorded and six unchanged opportunities are inferred as skipped. Together with four initial scenes, the materializer reconstructs 65 graph states. The logs contain 155 operations: 135 node updates and 20 edge additions or removals.

Change density differs substantially across workloads. *ShortSparse* averages 1.33 operations per accepted group, whereas continuously changing *OriginalChoreography* averages 5.23. Across all logs, position and rotation change in all 135 node-update records, while scale changes in 68. These selective patches avoid treating unchanged properties as authoritative. The official materializer reconstructs all four final scenes, emits exactly 61 accepted transitions, and all 13 snapshot-management regression

Table 3
Quantitative results from the four `StreamingAssets` workloads. Skip is inferred from a two-second nominal capture interval. Ratio compares JSONL bytes with compact reconstructed full-state bytes.

Workload	Sec.	Groups	Skip	Ops	Ops/group	Ratio
ShortSparse	18	9	1	12	1.33	0.46
MediumStaggered	30	15	1	29	1.93	0.65
LongComplex	54	24	4	46	1.92	0.67
OriginalChoreography	24	13	0	68	5.23	1.68
Total / weighted	126	61	6	155	2.54	0.86

tests pass.

The per-workload totals also show that accepted-transition count is shaped by both motion duration and pause structure. *LongComplex* produces the largest number of groups because it runs longest and alternates between active and hold intervals, whereas *OriginalChoreography* produces fewer but denser groups because multiple objects continue changing together. This distinction is important for later interpretation: a history can be long because it preserves many accepted moments, or heavy because each accepted moment carries many node- and edge-level edits.

4.3. Latency and storage

We measured the offline temporal-access path using Python 3.14.5 on the development machine. After 100 warm-up iterations, each workload was measured 2,000 times with `time.perf_counter_ns`. Final reconstruction applies all already-loaded transition groups to the external initial scene. Transition-replay generation builds the ordered transition document. End-to-end restore includes loading and grouping the JSONL file plus final reconstruction. Table 4 reports arithmetic means; these small controlled histories establish feasibility rather than large-scale performance.

Table 4
Offline temporal-access latency in milliseconds, averaged over 2,000 iterations after warm-up. Restore includes file load, grouping, and final-state reconstruction.

Workload	Groups	Reconstruct	Replay doc.	Restore
ShortSparse	9	0.86	0.89	3.54
MediumStaggered	15	0.98	1.86	5.10
LongComplex	24	1.54	2.66	6.07
OriginalChoreography	13	1.54	3.20	7.16

All mean restore times remain below 7.2 ms. Latency follows both transition-group count and operations per group. *LongComplex* has the most groups, but dense *OriginalChoreography* produces the slowest transition-replay generation and restore because each group carries more changes. These results support background restore and transition-replay generation for the current scale, but do not establish interactive latency for large deployed XR histories.

The four JSONL logs occupy 107,352 bytes in total, compared with 124,954 bytes for compact serialization of all reconstructed full states, giving an aggregate ratio of 0.86. This aggregate hides a strong workload effect. Incremental logs use 0.46, 0.65, and 0.67 times the full-state bytes for the sparse, staggered, and long workloads. In contrast, the continuously changing *OriginalChoreography* log uses 1.68 times the full-state bytes.

The result shows that change sparsity, rather than session duration alone, determines storage behavior. Each operation repeats sequence metadata, snapshot identifiers, timestamps, operation names, node identifiers, and JSON field names. That overhead is amortized when few nodes change, but dominates when most nodes change at every accepted time.

The results show that incremental histories are advantageous only when scene changes are sufficiently sparse. Highly dynamic scenes may instead favor periodic full snapshots, hybrid checkpoint strategies, or grouped binary encodings.

4.4. Capture-rate sensitivity analysis

As an additional validation, we evaluate sensitivity to capture rate while holding the motion driver fixed. The study uses a randomized 4×4×5 factorial design: four workloads, four rates (10, 30, 60, 120) captures per minute, and five repetitions for 80 runs. Each run exports an initial scene, incremental log, final ground-truth scene, motion manifest, and runtime metrics, and Table 6 reports the workload-specific means.

Table 5 reports means over the 20 runs at each CPM. As CPM increases from 10 to 120, accepted groups per run increase from 6.25 to 52.20 and operations increase from 19.35 to 124.15. Mean log size increases from 12.8 to 85.3 KB. The increase is sublinear relative to the twelve-fold capture rate increase because unchanged-state filtering rejects more attempts at high rates. Mean skipped-unchanged attempts increase from 0.00 to 11.80.

Table 5
Capture-rate sensitivity results. Values are means over 20 runs per rate. Capture ms is total snapshot-processing time per run.

CPM	Attempts	Accepted	Skipped	Ops	Rel. ops	KB	Capture ms
10	7.25	6.25	0.00	19.35	4.35	12.8	9.16
30	17.75	15.25	1.50	39.40	5.65	26.6	20.39
60	33.50	27.50	5.00	66.75	5.50	45.6	31.91
120	65.00	52.20	11.80	124.15	7.45	85.3	62.77

Higher CPM also exposes additional transient relation transitions. Mean relation operations rise from 4.35 at 10 CPM to 7.45 at 120 CPM, although intermediate rates vary by workload. This is not a ground-truth recall measurement, but it shows that coarse sampling can omit short-lived relations that finer sampling observes.

Every run performs a final changed-state capture after motion ends. Therefore, CPM controls intermediate temporal resolution but should not change the final state. The official materializer confirms this expectation: all 80 runs produce exactly the ground-truth final edge set. Position RMSE is zero in all analyzed runs, while maximum quaternion-angle rotation and scale RMSE are 1.87×10^{-6} degrees and 7.69×10^{-9}, respectively. Total snapshot processing remains small relative to motion duration, rising from 9.16 ms per run at 10 CPM to 62.77 ms at 120 CPM.

The workload interaction is important. For continuously changing *OriginalChoreography*, accepted groups rise from 5.0 to 49.0 and mean log size from 19.0 to 171.2 KB. For *ShortSparse*, filtering limits the increase to 4.0 to 24.8 groups and 5.1 to 20.2 KB. Higher capture rates therefore preserve more intermediate transitions at increased storage and processing cost while leaving final-state recovery unchanged in the tested workloads.

Increasing CPM does not make the final state more correct in these controlled runs; it changes how much intermediate temporal structure is retained between the same endpoints. Applications focused on final-state recovery can prefer lower rates and smaller histories, whereas applications that inspect relation changes or motion evolution may choose denser accepted-transition histories and pay the added storage cost.

Table 6
Detailed CPM means by workload. Values are averaged over five repetitions per workload-rate pair.

Workload	CPM	Accepted	Skipped	Ops	Rel. ops	KB	Capture ms
Long	10	10.0	0.0	26.0	6.0	17.4	13.92
Long	30	24.0	4.0	49.0	9.0	33.1	29.33
Long	60	45.0	10.0	80.4	8.4	55.4	45.01
Long	120	86.0	23.0	151.2	16.2	104.2	90.63
Medium	10	6.0	0.0	14.6	3.6	9.5	7.43
Medium	30	15.0	1.0	26.6	5.6	17.6	21.07
Medium	60	26.0	5.0	39.8	5.8	26.6	31.35
Medium	120	49.0	12.0	67.2	5.2	45.5	57.13
Original	10	5.0	0.0	28.6	3.6	19.0	10.65
Original	30	13.0	0.0	69.6	4.6	47.2	21.72
Original	60	25.0	0.0	128.8	3.8	88.0	34.45
Original	120	49.0	0.0	249.8	4.8	171.2	77.67
Short	10	4.0	0.0	8.2	4.2	5.1	4.64
Short	30	9.0	1.0	12.4	3.4	8.5	9.42
Short	60	14.0	5.0	18.0	4.0	12.4	16.85
Short	120	24.8	12.2	28.4	3.6	20.2	25.65

5. Discussion

The prototype demonstrates that runtime scene generation and offline temporal access can be separated by a small interface: complete graph states with stable identities. This separation permits environment-understanding and runtime components to evolve without forcing application consumers to understand low-level tracking. The resulting history is useful for restoration, transition replay, comparison, debugging, or temporal query [3, 4, 5, 6].

The explicit operation model provides a reusable representation of accepted changes beyond repeated full-state differencing. Without it, every downstream consumer must repeatedly compare complete graphs and independently decide how to apply changes. With it, the manager centralizes identity-based differencing, ordering, patch semantics, and recovery behavior. Rather than preserving only archived full graphs, the manager keeps accepted changes as explicit transition groups that later tools can reuse for restore, comparison, and inspection.

This design sits between two simple baselines. Full-snapshot storage is easy to load but repeats unchanged scene content at every accepted state. Aggregate differencing is compact when only endpoints matter, but it discards intermediate ordering and cannot replay accepted transitions. Incremental snapshot management preserves those transitions at the cost of materializing operation groups during offline access. In this design, that cost is kept out of the real-time XR loop by the online/offline split in Figure 2.

The current work remains a feasibility study. It uses a controlled Unity source rather than deployed AR/VR traffic, assumes one ordered writer and stable identities, keeps one external initial scene without intermediate checkpoints, lacks transactional transition-group commits and multi-writer reconciliation, and operates on small controlled workloads with basic identity and transform properties rather than dense observations or production security features. The most immediate extensions are deployed-runtime integration, periodic checkpoints, stronger commit and integrity markers, and source-aware ordering for multi-producer histories [25].

Even within that scope, the prototype separates three concerns that are often entangled in XR systems: producing a scene graph, deciding which states to preserve, and serving temporal products derived from preserved states.

6. Future Work

Two next steps are especially important for deployed AR and VR systems. First, the controlled extractor should be replaced by broader adapters that normalize heterogeneous upstream graph sources into the accepted-state interface used here. Deployed runtimes may expose partial updates, uncertain

identities, changing labels, or multiple graph producers at different rates [25]. The storage layer should likewise become more robust through transactional commits for each accepted transition group, periodic checkpoints for longer sessions, and lightweight integrity metadata for recovery.

Second, the history model should be exercised in collaborative and deployed settings. Multi-user XR will require source-aware ordering or reconciliation rather than the single ordered writer assumed here, and longer real-world traces are needed to test restore latency, query usefulness, and review workflows beyond controlled motion scripts. These studies would clarify which downstream tasks benefit most from accepted-transition history and where richer indexing or collaboration semantics become worth adding [6, 5, 26, 10].

7. Conclusion

This paper presents an incremental snapshot manager for XR applications that require replay, restoration, comparison, or temporal access to scene history. By assuming that a runtime or shared-space service supplies complete graph states, the prototype separates runtime scene generation from offline temporal access. Starting from an externally supplied initial scene, it filters unchanged states, appends explicit node and edge operations, checks ordering, recovers from an incomplete final record, reconstructs graph state, computes aggregate comparison, and supplies accepted transitions for replay.

Across four controlled workloads, the prototype manages 61 accepted transition groups and 155 operations over 126 seconds, successfully reconstructs every final state, and passes 13 snapshot-management regression tests. The study also shows that readable incremental JSON benefits sparse-change workloads but is not automatically storage-efficient for highly dynamic graphs. A capture-rate sensitivity analysis further shows that increasing CPM captures more intermediate and transient relation transitions at higher storage cost, while every run preserves the final graph. These results support snapshot management as a practical interface between runtime scene generation and offline temporal access, while motivating stronger durability guarantees, temporal indexing, and scalable encoding.

Viewed this way, the contribution is not a new scene-understanding algorithm but a narrow systems layer that preserves temporal access after scene graphs already exist. Many XR runtimes can produce a current-state graph, but far fewer retain an ordered, reusable history once the scene changes. The prototype shows that even a transparent append-only design can bridge that gap for controlled workloads and provide a concrete baseline for more durable, queryable, and collaborative temporal scene-management systems. The main trade-off is also clear: accepted-transition history improves temporal access, but reconstruction cost and storage overhead depend on change density.

Acknowledgments

This work was supported by the Korea Institute for Advancement of Technology(KIAT) grant funded by the Korea Government(MOTIE) (RS-2025-02304167, HRD Program for Industrial Innovation). This work was supported by the IITP(Institute of Information & Communications Technology Planning & Evaluation)-ITRC(Information Technology Research Center) grant funded by the Korea government(Ministry of Science and ICT)(IITP-2026-RS-2024-00436398). This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (RS-2026-25523396, Core Technology Development for Immersive Content)

Declaration on Generative AI

During the preparation of this work, the author used OpenAI Codex in order to: inspect the implementation, calculate descriptive statistics, draft and revise text, and assist with reference formatting. After using this tool, the author reviewed and edited the content as needed and takes full responsibility for the publication's content.

References

[1] J. Wald, H. Dhamo, N. Navab, F. Tombari, Learning 3d semantic scene graphs from 3d indoor reconstructions, in: 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2020, pp. 3960–3969. doi:10.1109/CVPR42600.2020.00402.

[2] S.-C. Wu, J. Wald, K. Tateno, N. Navab, F. Tombari, Scenegraphfusion: Incremental 3d scene graph prediction from rgb-d sequences, in: 2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2021, pp. 7511–7521. doi:10.1109/CVPR46437.2021.00743.

[3] S. Kim, Experience graph using spatio-temporal scene data for replaying mixed reality interaction, in: 2024 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW), 2024, pp. 1112–1113. doi:10.1109/VRW62533.2024.00350.

[4] S. Choi, D. Kim, T. Ha, S. Kim, W. Woo, Task breakpoint generation using origin-centric graph in virtual reality recordings for adaptive playback, IEEE Transactions on Visualization and Computer Graphics 32 (2026) 3283–3292. URL: https://doi.org/10.1109/TVCG.2026.3679910. doi:10.1109/TVCG.2026.3679910.

[5] E. Carra, F. Pellacini, Scenegit: a practical system for diffing and merging 3d environments, ACM Trans. Graph. 38 (2019). URL: https://doi.org/10.1145/3355089.3356550. doi:10.1145/3355089.3356550.

[6] L. Zhang, A. Agrawal, S. Oney, A. Guo, Vrgit: A version control system for collaborative content creation in virtual reality, in: Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems, CHI '23, Association for Computing Machinery, New York, NY, USA, 2023. URL: https://doi.org/10.1145/3544548.3581136. doi:10.1145/3544548.3581136.

[7] R. Suzuki, R. H. Kazi, L.-y. Wei, S. DiVerdi, W. Li, D. Leithinger, Realitysketch: Embedding responsive graphics and visualizations in ar through dynamic sketching, in: Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology, UIST '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 166–181. URL: https://doi.org/10.1145/3379337.3415892. doi:10.1145/3379337.3415892.

[8] S. Kim, D. Kim, T. Son, W. Woo, Realitycrafter: User-guided editable 3d scene generation from a single image in mixed reality, in: Adjunct Proceedings of the 38th Annual ACM Symposium on User Interface Software and Technology, UIST Adjunct '25, Association for Computing Machinery, New York, NY, USA, 2025. URL: https://doi.org/10.1145/3746058.3758405. doi:10.1145/3746058.3758405.

[9] J. Hong, M. Baeck, S. Kim, Y. Shin, W. Woo, Collaborative scene mood authoring with voice-driven multimodal feedback design in virtual reality, in: SIGGRAPH Asia 2025 XR, SA '25, Association for Computing Machinery, New York, NY, USA, 2025. URL: https://doi.org/10.1145/3761667.3761960. doi:10.1145/3761667.3761960.

[10] S. Kim, D. Kim, J.-E. Shin, W. Woo, Object cluster registration of dissimilar rooms using geometric spatial affordance graph to generate shared virtual spaces, in: 2024 IEEE Conference Virtual Reality and 3D User Interfaces (VR), 2024, pp. 796–805. doi:10.1109/VR58804.2024.00099.

[11] A. Rosinol, A. Gupta, M. Abate, J. Shi, L. Carlone, 3d dynamic scene graphs: Actionable spatial perception with places, objects, and humans, 2020. doi:10.15607/RSS.2020.XVI.079.

[12] A. Rosinol, A. Violette, M. Abate, N. Hughes, Y. Chang, J. Shi, A. Gupta, L. Carlone, Kimera: From slam to spatial perception with 3d dynamic scene graphs, The International Journal of Robotics Research 40 (2021) 1510–1546. URL: https://doi.org/10.1177/02783649211056674. doi:10.1177/02783649211056674.

[13] N. Hughes, Y. Chang, L. Carlone, Hydra: A real-time spatial perception system for 3d scene graph construction and optimization, 2022. doi:10.15607/RSS.2022.XVIII.050.

[14] J. Ji, R. Krishna, L. Fei-Fei, J. C. Niebles, Action genome: Actions as compositions of spatio-temporal scene graphs, in: 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2020, pp. 10233–10244. doi:10.1109/CVPR42600.2020.01025.

[15] Y. Cong, W. Liao, H. Ackermann, B. Rosenhahn, M. Y. Yang, Spatial-temporal transformer for dynamic scene graph generation, in: Proceedings of the IEEE/CVF international conference on

computer vision, 2021, pp. 16372–16382.
[16] S. Nag, K. Min, S. Tripathi, A. K. Roy-Chowdhury, Unbiased scene graph generation in videos, in: 2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2023, pp. 22803–22813. doi:10.1109/CVPR52729.2023.02184.
[17] J. Ost, F. Mannan, N. Thuerey, J. Knodt, F. Heide, Neural scene graphs for dynamic scenes, in: 2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2021, pp. 2855–2864. doi:10.1109/CVPR46437.2021.00288.
[18] L. Schmid, M. Abate, Y. Chang, L. Carlone, Khronos: A unified approach for spatio-temporal metric-semantic slam in dynamic environments (2024). doi:10.15607/RSS.2024.XX.081.
[19] B. Kerbl, G. Kopanas, T. Leimkuehler, G. Drettakis, 3d gaussian splatting for real-time radiance field rendering, ACM Trans. Graph. 42 (2023). URL: https://doi.org/10.1145/3592433. doi:10.1145/3592433.
[20] N. Williams, JavaScript Object Notation (JSON) Text Sequences, RFC 7464, Internet Engineering Task Force, 2015. URL: https://www.rfc-editor.org/rfc/rfc7464.html. doi:10.17487/RFC7464.
[21] JSON Lines, Json lines documentation, https://jsonlines.org/, 2026. Accessed: 2026-06-15.
[22] M. Fowler, Event sourcing, https://martinfowler.com/eaaDev/EventSourcing.html, 2005. Accessed: 2026-06-16.
[23] M. Fowler, Cqrs, https://martinfowler.com/bliki/CQRS.html, 2011. Accessed: 2026-06-16.
[24] KAIST ARRC, Scenegraph: A unified scene graph schema draft, https://github.com/kaist-arrc/SceneGraph, 2026. GitHub repository, accessed: 2026-06-22.
[25] Y. Chang, N. Hughes, A. Ray, L. Carlone, Hydra-multi: Collaborative online construction of 3d scene graphs with multi-robot teams (2023) 10995–11002. doi:10.1109/IROS55552.2023.10341838.
[26] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski, Conflict-free replicated data types, in: X. Défago, F. Petit, V. Villain (Eds.), Stabilization, Safety, and Security of Distributed Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 386–400.