Design and Implementation of an AR System for Real-Time Urban Model Editing and Visualization

Cephas B. Mwimangire[1], Mohammad Hosseingholizadeh[2], Hamidreza Ostadabbas[2], and Zhihang Yao[1]

[1]Centre for Geodesy and Geoinformatics, Stuttgart University of Applied Sciences (HFT Stuttgart), Stuttgart, Germany – (42mwbi1mpg, zhihang.yao)@hft-stuttgart.de
[2]Geoinformatics Department, die STEG Stadtentwicklung GmbH, Stuttgart, Germany – (mohammad.hosseingholizadeh, hamidreza.ostadabbas)@steg.de

Keywords: Augmented Reality, CityGML, WebSocket, Real-Time Visualization, 3D Tiles, HoloLens 2.

Abstract

Urban planning increasingly relies on 3D city models and geospatial databases for sustainable development. However, no complete system currently offers immersive 3D visualization of building attributes in augmented reality (AR), bidirectional attribute editing propagated back to an authoritative geodatabase, and real-time synchronization across multiple clients. In this paper we present a full-stack, multi-level architecture that bridges Microsoft HoloLens 2 AR visualization with a CityGML/3DCityDB geospatial backend via a Django middleware layer. A hybrid WebSocket+HTTP polling communication layer, built on Django Channels with Redis pub/sub and a platform-abstracted Unity client, delivers sub second server-push updates (200-500ms latency) while automatically falling back to HTTP polling when WebSocket connections are unavailable, reducing per-client bandwidth by over three orders of magnitude compared to periodic polling. The architecture has been validated on 5,005 buildings in the German municipality of Bisingen. For future research and development plans, it would be valuable to focus on geometry editing workflows and the integration of AI-driven interaction. In particular, enabling voice-based commands would significantly improve usability on HoloLens 2, where typing is inconvenient for most users.

1. Introduction

Urban planning increasingly relies on 3D city models and geospatial databases for sustainable development, making interactive visualization of building attributes a key enabler for stakeholder engagement (Beck and Mitkiewicz, 2025). In Germany, the STEG company in Stuttgart is working on integrating 3D visualization in an augmented reality (AR) environment and Unity Engine to complement its existing web-based GIS application for municipal energy consulting. During on-site workshops with municipal planners, the ability to modify a building’s renovation parameters and immediately see the updated CO_2 classification overlaid on the physical streetscape has proven decisive for accelerating consensus on retrofit priorities. Without such a closed loop system, planners must alternate between desktop GIS tools and paper maps, losing spatial context and delaying decision cycles (Ketzler et al., 2020).

Despite the growing use of AR and game engines in visualization (Schlüter et al., 2025), and recent work on open geospatial data in game engines (Rantanen et al., 2023), three specific gaps remain unaddressed in the literature: no existing system provides bidirectional attribute editing in AR that is propagated back to an authoritative geodatabase (CityGML/3DCityDB) and synchronized across multiple clients in real time; per-building runtime coloring of batched OGC 3D Tiles meshes using glTF metadata extensions has not been demonstrated on resource-constrained AR hardware (HoloLens 2, Snapdragon 850, 4GB RAM); and the reliability of WebSocket-based push delivery in enterprise Wi-Fi environments, where proxies and firewalls frequently block the HTTP Upgrade mechanism, has not been addressed with a formally defined fallback strategy (Guha and Francis, 2005; Pimentel and Nickerson, 2012).

Conventional HTTP-polling approaches introduce a fundamental trade-off between update latency and bandwidth consumption (Pimentel and Nickerson, 2012): in a scenario with approximately 5,000 buildings, a 60-second polling interval generates 1.7-3.5GB of redundant transfer per client per hour, even when no data has changed. The WebSocket protocol (RFC 6455) addresses this by establishing a persistent, full-duplex TCP connection with sub-second latency and negligible per-message overhead (Fette and Melnikov, 2011; Lubbers and Greco, 2010).

In this paper, we present a full-stack architecture that bridges Microsoft HoloLens 2 AR visualization with a CityGML/3DCityDB geospatial backend via a Django middleware layer, directly addressing all three gaps. The principal contributions are: a hybrid WebSocket+HTTP polling communication layer with a three-tier automatic fallback strategy that delivers sub-second push updates with degradations in restrictive networks; per-building vertex coloring of OGC 3D Tiles at runtime, using glTF metadata extensions with frame spread batching to maintain 60fps on HoloLens 2; a two-tier disk/memory caching strategy enabling offline-capable startup and O(1) rendering lookups for thousands of buildings; and a dual-mode interaction system (desktop mouse + 2 HoloLens 2 XR gestures) with API-driven CRUD editing propagated back to the geodatabase. The architecture has been validated on 5,005 buildings in the German municipality of Bisingen.

2. System Architecture

The system architecture establishes a seamless pipeline from authoritative geospatial data to immersive AR-based interaction (see figure 1).

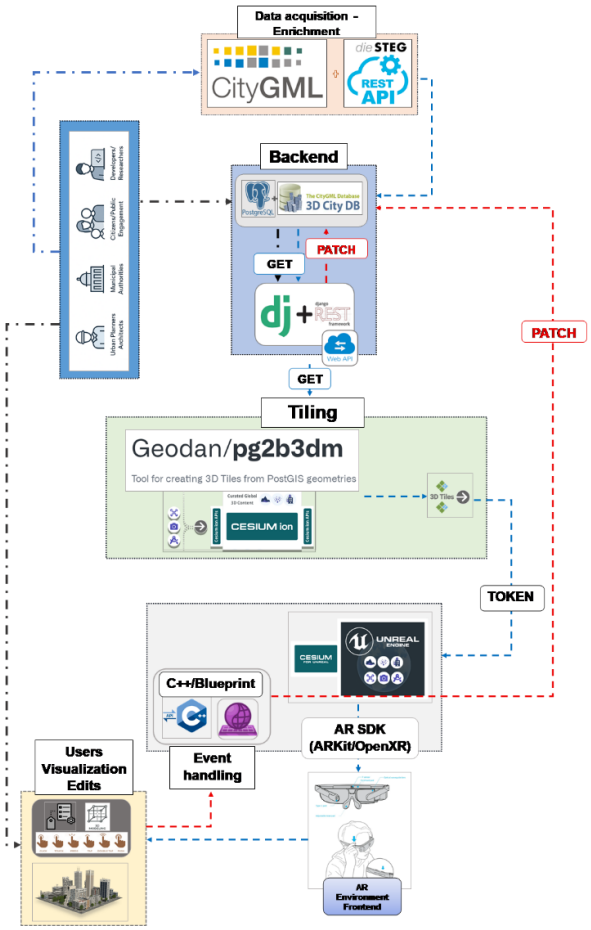

Figure 1. System architecture and workflow

2.1 Data Layer

As the primary data source, the official LoD_2 3D building models from the State Agency for Geoinformation and Land Development of Baden-Württemberg (LGL-BW) are used. These datasets follow the international CityGML standard (Kolbe, 2009; Kutzner et al., 2020), which supports the management and exchange of semantic urban information, including thematic, spatial, topological, and appearance attributes. All data are stored and maintained in a PostgreSQL/PostGIS geospatial database using the 3D City Database (3DCityDB) solution (Yao et al., 2025). Regarding Level of Detail (LoD), the current deployment uses LoD_2 buildings (extruded footprints with standardized roof shapes), which provide a practical balance between geometric fidelity and rendering performance on HoloLens 2. The architecture is not restricted to a single LoD: since the 3DCityDB stores CityGML data at any LoD and Cesium Ion’s tiling pipeline preserves the input geometry, switching to LoD_1 (block models) or LoD_3 (detailed façade models) requires only re-tiling the dataset without changes to the client-side code. The attribute editing and WebSocket push pipelines operate on semantic properties attached to building features, which are LoD independent.

2.2 Middleware Layer

STEG has developed a Django-based server application that enables efficient client-server communication and real-time CRUD operations via REST APIs secured with JWT authentication (*rest_framework_simplejwt*). The same Django application concurrently manages long-lived WebSocket connections through Django Channels (Django Software Foundation, 2024), sharing the authentication system, ORM, and business logic without code duplication.

2.3 Visualization Layer

The CityGML data stored in the 3DCityDB is exported and converted into OGC 3D Tiles (Cozzi and Lilley, 2023) via the Cesium Ion platform, then integrated into Unity Engine using the Cesium for Unity plugin (Cesium, 2025). Within Unity, individual building features are accessed thematically via the glTF metadata extensions *EXT_mesh_features* and *EXT_structural_metadata* (CesiumGS, 2026). Interactive access to and editing of individual feature attributes through feature selection is implemented using C# scripts that drive custom UI panels for attribute display and modification.

2.4 Technology Stack

Table 1 summarizes the technology stack spanning all three architectural layers. The system comprises approximately 6,550 lines of C# across ten Unity scripts, a Django REST + Channels backend, and PostgreSQL/PostGIS with 3DCityDB.

Layer	Component	Technology
Client	Runtime	Unity 2022.3 LTS, IL2CPP, ARM64
	HMD	HoloLens 2 (Snapdragon 850, 4GB)
	WS (.NET)	ClientWebSocket (.NET Std 2.1)
	WS (UWP)	MessageWebSocket (UWP SDK)
	JSON	Newtonsoft.Json 13.x
Server	Framework	Django REST Framework
	WebSocket	Django Channels (ASGI)
	ASGI server	Daphne/Uvicorn
	Auth	rest_framework_simplejwt
	Broket	Redis (pub/sub channel layer)
Data	Geodatabase	PostgreSQL + PostGIS
	Schema	3DCityDB 5.0
	3D Tiles	Cesium Ion (OGC 3D Tiles 1.1)

Table 1. Technology stack for the hybrid real-time architecture.

3. Real-Time Communication Architecture

3.1 Hybrid Design

Figure 2 provides an overview of the real-time update mechanism that replaces the original periodic polling approach. The original system relied on periodic HTTP polling (default 60s interval) to detect building attribute changes. While functionally correct, polling introduces well-documented limitations in real-time data delivery scenarios (Pimentel and Nickerson, 2012): with a 60-second interval, a building modification requires up to one minute to propagate to the HoloLens 2 client, an unacceptable delay during collaborative editing sessions.

Reducing the interval to 5-10s would generate 1,440 redundant API requests per hour per client, each transferring the complete ~4,800-building JSON payload (~2.4MB), resulting in 1.7-3.5GB of redundant transfer per hour. Rather than a WebSocket-only replacement, a hybrid architecture was adopted. WebSocket connections are inherently less reliable than stateless HTTP requests because network interruptions, proxy timeouts, and server restarts can sever the persistent connection without the client’s immediate knowledge (Grigorik, 2013). Enterprise network environments, particularly hospital and municipal Wi-Fi networks where HoloLens 2 devices are commonly deployed, may employ HTTP proxies or firewalls that block the WebSocket

Upgrade header (Guha and Francis, 2005; Pimentel and Nickerson, 2012). The polling fallback ensures graceful degradation to fully-functional behaviour under these conditions.

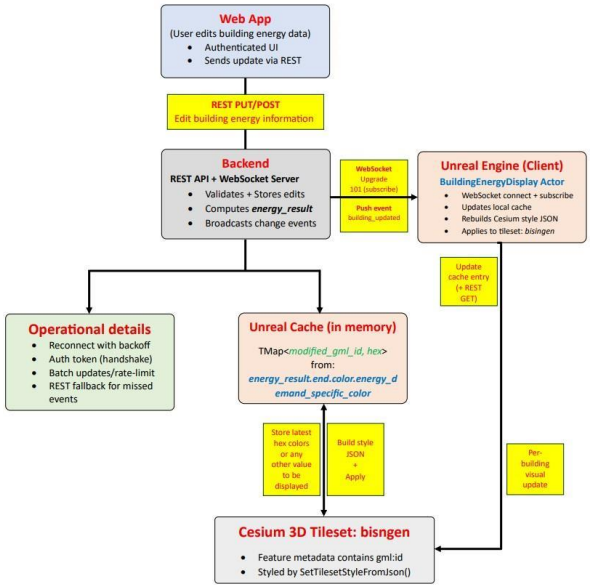

Figure 2. Conceptual schema of the real-time update mechanism.

3.2 Backend Architecture: Django Channels and ASGI

The backend WebSocket infrastructure is built on Django Channels (Django Software Foundation, 2024), which upgrades Django’s default synchronous WSGI architecture to the asynchronous ASGI protocol. The ASGI server (Daphne and/or Uvicorn) maintains persistent WebSocket connections alongside traditional HTTP request-response cycles, unified under a single deployment.

The WebSocket endpoint is exposed at the URL path *ws/buildings/{community_id}/,* where *community_id* identifies the municipality (e.g., 08417008 for Bisingen). This community-scoped routing ensures that clients receive updates only for the geographic area they are currently visualizing, implementing the spatial partitioning strategy identified by Ketzler et al. (Ketzler et al., 2020) as essential for scalable urban digital twin systems. Authentication reuses the REST API’s JWT tokens: the client transmits its access token in a JSON message *{“type”: “authenticate”, “token”: “<JWT>”}* after connection establishment. The consumer validates this token using the *rest_framework_simplejwt* library, ensuring a single authentication authority across both channels. Upon successful authentication, the consumer joins a Redis-backed channel group named *buildings_{community_id}*, implementing *publish/subscribe* messaging. Redis serves as the in-memory message broker, providing inter-process communication when multiple ASGI workers handle WebSocket connections across different server instances.

Figure 3 details the four phases of a WebSocket connection lifecycle. After the initial TCP+TLS handshake and HTTP 101 upgrade, the client authenticates by sending its JWT token in a JSON message. The server validates the token, subscribes the client to the Redis backed channel group, and confirms with an *auth_ok* response. During steady-state operation, 30-second keepalive pings maintain the connection, while server-push messages (*building_updated, building_deleted*) arrive asynchronously. The client acknowledges each message, enabling the server to detect stale connections. Graceful disconnection is handled via the standard WebSocket close frame handshake.

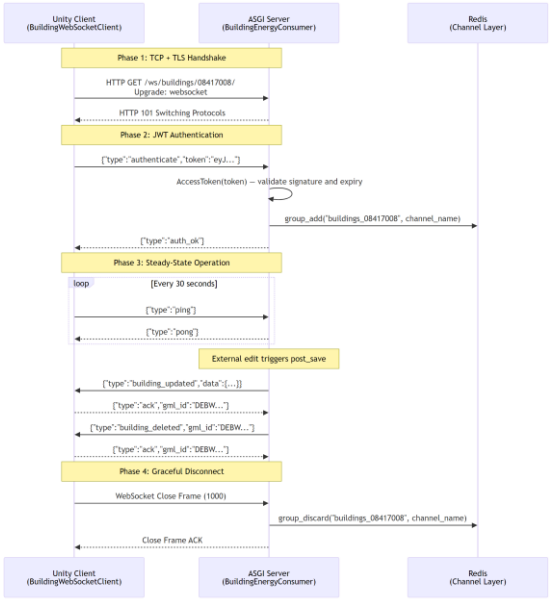

Figure 3. WebSocket connection lifecycle.

3.3 Event-Driven Push Pipeline

Data flow from database modification to client notification follows a five-stage event-driven pipeline (Figure 4) REST API write A building’s energy attributes are modified via a PUT request to */geospatial/buildings-energy/{gml_id}/.* The API view performs the database update and returns the recalculated record; Django signal dispatch Django’s *post_save* signal fires on the building model instance, invoking the registered handler *notify_building_updated()*; serialization and group send The handler serializes the building instance into JSON and invokes *group_send()* with message type *building_updated* targeting the group buildings_*{community_id};* Redis distribution Redis distributes the message to all ASGI worker processes with consumers subscribed to the target group; WebSocket delivery Each consumer forwards the serialized building data to its WebSocket client as a JSON text frame. This architecture decouples the REST API write path from the WebSocket notification path: the API view returns immediately after the database write, while the signal handler asynchronously propagates the change via the Redis message broker. For batch operations such as district-level simulation results, a *notify_bulk_update()* utility aggregates multiple records into a single *bulk_update* message.

3.4 Client-Side Platform Abstraction

A significant implementation challenge arises from the divergent WebSocket APIs on the two target platforms: the Unity Editor on Windows desktop (*System.Net.WebSockets.ClientWebSocket*, .NET Standard 2.1) and HoloLens 2 running UWP (*Windows.Networking.Sockets.MessageWebSocket*, event-driven model requiring UI-thread dispatch).

The *BuildingWebSocketClient* component abstracts this divergence using compile-time conditional compilation (#if UNITY_WSA && !UNITY_EDITOR). Both implementations Properties (*IsConnected, IsAuthenticated*) expose identical public events (*OnBuildingUpdated, OnBulkUpdate, OnBuilding Deleted*) and methods (*Connect(), Disconnect()*). The

BuildingEnergyManager interacts exclusively with this public interface, remaining agnostic to the underlying socket implementation. In both cases, complete messages are enqueued into a thread-safe Queue<string>, and Unity’s *Update()* method drains the queue each frame on the main thread, ensuring all Unity API calls (which are not thread-safe) occur on the main thread.

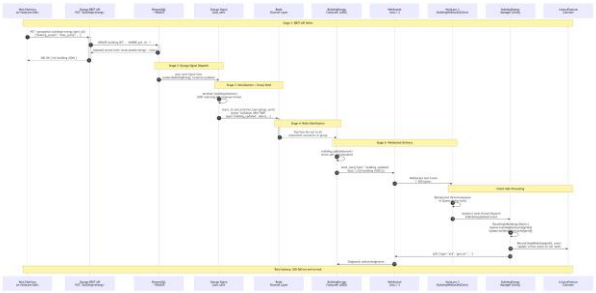

Figure 4. Sequence diagram of the event-driven push pipeline

3.5 Automatic Fallback and Reconnection

The hybrid architecture implements a three-tier reliability strategy (Figure 5) that ensures continuous data synchronization under varying network conditions. In Tier 1, the WebSocket is connected and authenticated, allowing building updates to arrive via server push with sub-second latency. In this state, HTTP polling is completely suppressed; the polling timer is not incremented while *wsClient.IsConnected && wsClient.IsAuthenticated* evaluates to true, ensuring zero redundant network traffic.

If the connection is lost, the system transitions to Tier 2, where the WebSocket is disconnected and attempting to reconnect. Upon connection loss, an exponential backoff reconnection sequence initiates with delays dn = min(d0 × n, 60) s, where d0 = 5s and n = 1,2,...,5. During this reconnection phase, the polling fallback activates automatically at 120-second intervals, ensuring continued data synchronization. If all reconnection attempts are exhausted, the system enters Tier 3, in which the WebSocket is considered permanently failed. In this state, the system operates exclusively via HTTP polling at 120s intervals, functionally equivalent to the original pre-WebSocket behaviour, ensuring that no data synchronization capability is lost even in environments that block the HTTP Upgrade mechanism. Upon successful reconnection at any stage, the counter resets and the system returns to Tier 1.

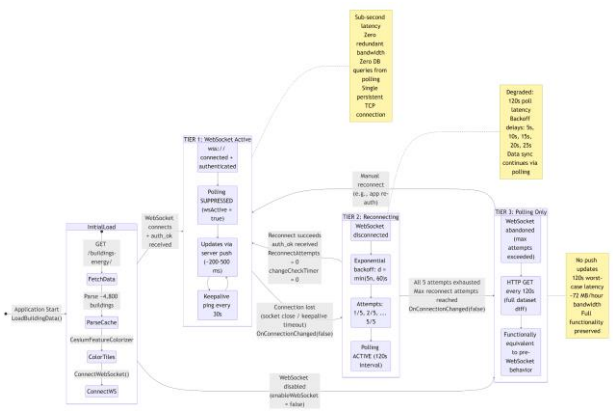

Figure 5. State diagram of the three-tier hybrid architecture.

4. Client-side implementation

This section details three novel client-side components: per-building vertex coloring of OGC 3D Tiles, a two-tier caching strategy, and an interactive building selection and editing system for HoloLens 2.

4.1 Per-Building Vertex Coloring of 3D Tiles

A key contribution of this work is the ability to colour individual buildings within batched 3D Tiles meshes at runtime, using the glTF metadata extensions *EXT_mesh_features* and *EXT_structural_metadata* (CesiumGS, 2026). The *CesiumFeatureColorizer* component (~850 lines) reads per-vertex feature IDs from the Cesium for Unity SDK (CesiumGS, 2026), resolves each feature’s *gml:id* against the energy classification colours obtained from the REST API, and writes the result into per-mesh vertex colour arrays. Figure 6. shows an example of the core coloring routine.

```
public void ColorizeMesh(MeshRenderer renderer) {
    MeshFilter mf = renderer.GetComponent<MeshFilter>();
    Mesh mesh = mf.sharedMesh;

    int vCount = mesh.vertexCount;

    if (!TryGetCesiumMeshContext(renderer, ctx)) return;

    featureColorCache.Clear(); // per-mesh cache

    Color[] colors = new Color[vCount];

    int colored = 0, buildings = 0;

    if (ctx.featureIdAttribute != null) {
        ColorizeWithAttribute(
            ctx.featureIdAttribute,
            ctx.propertyTable,
            ctx.gmlIdProperty,
            vCount,
            colors,
            ref colored,
            ref buildings,
            new HashSet<Color>()
        );
    }

    Mesh newMesh = Object.Instantiate(mesh);
    newMesh.colors = colors;

    mf.mesh = newMesh;

    ApplyVertexColorMaterial(
        renderer,
        ctx.modelMetadata
    );
}
```

Figure 6. Core per-mesh coloring: feature IDs are read from Cesium

To maintain 60fps on HoloLens 2, where frame times exceeding 16.67ms produce perceptible hologram judder, coloring is spread across multiple frames using a configurable *meshesPerFrame* parameter (default 50). A persistent coroutine re-scans every 3s to catch late-streamed and LOD-swapped tiles. When a single building is updated via WebSocket push, *RecolorSingleBuilding()* targets only meshes containing that building’s feature ID, avoiding a full tileset re-traversal. A six-level GML ID matching strategy handles format mismatches between CityGML tilesets and API identifiers: (1) direct cache lookup, (2) reverse *gmlIdCache*, (3) forward cache, (4) cached match from *idMatchCache*, (5) exact key scan, and (6) substring matching as a last resort.

4.2 Two-Tier Caching Strategy

The *BuildingEnergyManager* (~1,500 lines) implements a two-tier caching strategy to minimize application start up time and network usage. In the first tier, the disk cache persists the complete API response (~2.4MB rawJSON for 4,800 buildings) to local storage (*Application.persistentDataPath*). On subsequent launches, the *cachedJSON* is loaded and parsed entirely on the client without any network request, enabling offline-capable startup.

In the second tier, the memory cache provides fast access during rendering through three in-memory dictionaries: *buildingDataCache* (parsed *BuildingData* structs), *buildingColorCache* (pre-parsed Color values), and *gmlIdCache* (modified-original GML ID mapping), each enabling O(1) lookup performance. A smart download strategy first scans the loaded tile set to identify which building IDs are actually present, then downloads the full dataset but parses only the matched subset. This avoids wasted parsing for buildings outside the spatial bounding box defined by the *BoundingBoxTileExcluder* component, which filters tiles using ECEF-to-local coordinate transformation against a configurable geographic extent.

Figure 7. shows the hybrid polling/push selection logic in the UPDATE() method, which suppresses all polling traffic while the WebSocket connection is active.

```
void Update() {
    bool wsActive = wsClient != null
        && wsClient.IsConnected
        && wsClient.IsAuthenticated;

    websocketConnected = wsActive;

    if (enableChangeDetection
        && buildingDataCache.Count > 0
        && !wsActive) {

        changeCheckTimer += Time.deltaTime;

        if (changeCheckTimer >= changeCheckInterval) {
            changeCheckTimer = 0f;
            StartCoroutine(CheckForExternalChanges());
        }
    }
}
```

Figure 7. Hybrid polling/push selection.

4.3 Interactive Building Selection and Editing

HoloLens 2, air-tap views data and a hold gesture (≥0.5s) followed by release opens the edit form. The hold-threshold pattern avoids accidental edits during casual exploration. Upon interaction, a physics ray cast is performed (head gaze ray on HoloLens 2, screen-point ray on desktop). The hit object’s *CesiumPrimitiveFeatures* component provides the feature ID via *GetFeatureIdFromRaycastHit*(), and the parent Cesium Model Metadata supplies the property table from which the *gml:id* is extracted. This GML ID then drives a REST API fetch for the building’s full attribute set.

The *BuildingAttributesForm* (~1,600 lines) renders an API-driven CRUD form: field definitions, including choice arrays for dropdowns, are fetched from the server with *field_type=basic*, mirroring the German Energieausweisstructure (construction year class, heating system, renovation parameters). Editing submits a JWT authenticated HTTP PUT request (Figure 8.), which triggers the server-side *post_save* signal and the WebSocket push pipeline previously.

```
IEnumerator SendPutRequest() {
    string url = $"{apiBaseUrl}/geospatial/"
        + $"buildings-energy/{currentGmlId}/"
        + $"?community_id={communityId}"
        + "&field_type=basic";

    string json = BuildJsonPayload();

    using (var req = new UnityWebRequest(url, "PUT")) {
        byte[] body = Encoding.UTF8.GetBytes(json);

        req.uploadHandler = new UploadHandlerRaw(body);
        req.downloadHandler = new DownloadHandlerBuffer();

        req.SetRequestHeader("Content-Type", "application/json");
        req.SetRequestHeader("Authorization", $"Bearer {accessToken}");

        yield return req.SendWebRequest();

        if (req.result == UnityWebRequest.Result.Success)
            CloseForm();
    }
}
```

Figure 8. JWT-authenticated HTTP PUT request for saving edited building attributes back to the Django backend.

4.4 Cross-Platform Abstraction

Since the application targets both desktop (Win dows10/11) and HoloLens 2 (UWP/ARM64), every network and input module uses compile-time platform directives to select the appropriate implementation. Figure 9. shows a representative import section from the WebSocket client.

```
#if UNITY_WSA
using Windows.Networking.Sockets;
using Windows.Storage.Streams;
using Windows.Security.Cryptography.Certificates;
#else
using System.Net.WebSockets;
using System.Threading;
#endif
```

Figure 9. Compile-time platform abstraction.

This pattern is applied consistently across the code base: the *BuildingWebSocketClient* (connection management, message dispatch), the *BuildingEnergyManager* (REST API requests via *UnityWebRequest* on both plat forms but with UWP certificate handling), and the *CesiumMetadataReader* (ray cast input from mouse on desktop, gaze + air-tap on HoloLens 2). The strategy ensures a single codebase compiles for both targets without runtime overhead.

4.5 Energy Visualization and Legend

Figure 10 shows the HoloLens 2 navigation panel that provides situational awareness during AR sessions. The panel displays the current municipality name, total building count, and quick-access buttons for toggling between visualization modes (energy demand, heating system type, renovation status). A minimap inset shows the user’s position relative to the tileset bounding box, helping planners orient themselves during neighbourhood walkthroughs.

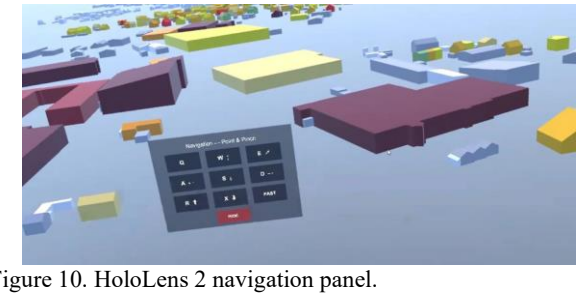

Figure 10. HoloLens 2 navigation panel.

The *EnergyDemandLegend* component dynamically generates a gradient colour bar and class labels from the currently active colour ramp (Figure 11).

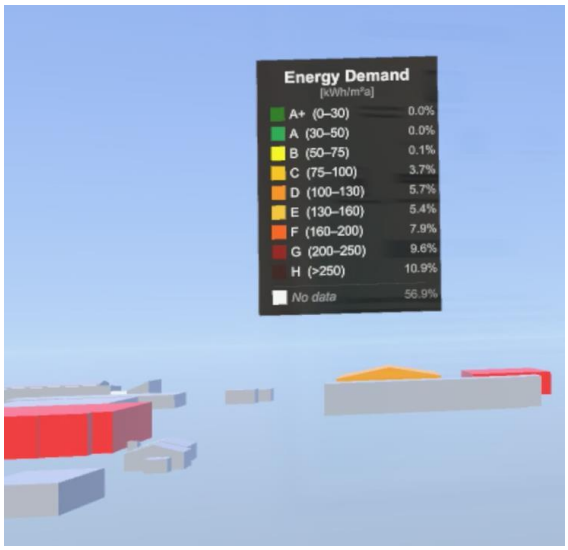

Figure 11. Dynamically generated energy demand legend showing the colour gradient and class boundaries for CO_2 emission classification in the HoloLens 2 AR view.

For each colour class, a normalized gradient texture is generated procedurally and assigned to RawImage UI elements. The legend automatically updates when the colour mapping changes, e.g. when switching between different energy indicators. An accompanying *BuildingInfoPanel* displays individual building attributes upon selection, including GML ID, construction year class, heating system, and estimated energy demand (Figure 12).

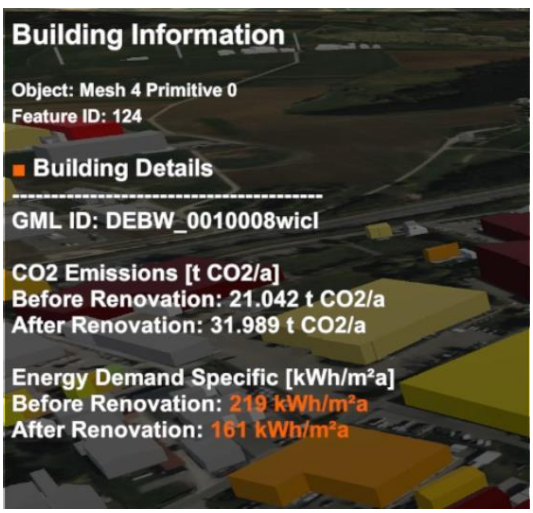

Figure 12. Building energy information panel displayed in HoloLens 2 upon air-tap selection.

5. Use case and results

The system was validated using data from the German municipality of Bisingen (5,005 buildings, ~11,000 inhabitants), where STEG GmbH is implementing a CO_2 concept evaluation for the municipal administration. The core planning task requires stakeholders to assess the energy renovation potential of the building stock by modifying individual building parameters (construction year class, heating system type, insulation status) and observing the resulting changes in the CO_2 emission classification across the municipality.

5.1 Deployment and Workflow

The Unity project was configured with Windows Mixed Reality XR settings and packaged as a UWP build targeting ARM64 with IL2CPP compilation. The application was deployed to HoloLens 2 via the Windows Device Portal. Figure 13 shows the application at launch on HoloLens 2, where the 3D tileset is initially loaded with default appearance before energy classification colours are applied. The georeferenced overlay aligns the virtual building models with the physical environment, providing an immediate sense of spatial context for the planner.

Figure 13. Application launch view on HoloLens 2: the LoD2 3D Tiles are loaded and georeferenced over the physical environment before energy classification coloring is applied.

The LoD_2 3D Tiles datasets, hosted on Cesium Ion, were loaded and rendered natively on the device, while spatial anchors and head tracking ensure stable, georeferenced AR overlays on the physical environment. A typical planning session proceeds as follows: (1) the planner walks through a neighbourhood wearing the HoloLens 2, seeing the real buildings overlaid with their CO2 colour classification; (2) the planner selects a building via air-tap to inspect its current energy attributes (Figure 12); (3) the planner opens the edit form via a hold gesture, modifies renovation parameters such as heating system type or insulation class, and submits the change (Figure 14); (4) within 200–500ms, the building’s colour updates on all connected HoloLens 2 devices via the WebSocket push pipeline, enabling the planning team to immediately see the cumulative impact of each decision.

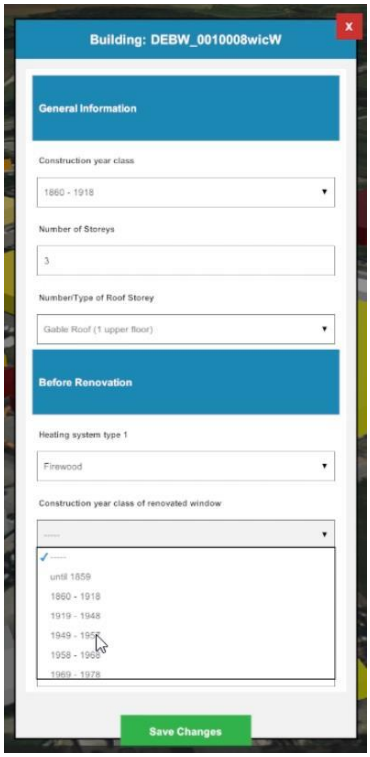

Figure 14. Interactive attribute editing form in HoloLens 2.

This closed-loop workflow, where editing, database up date, and visual feedback occur in a single interaction cycle, eliminates the traditional need to alternate between desktop GIS tools and on-site inspections. For visualization, a user-defined colour ramp representing the CO_2 class is applied, with classes corresponding to specific ranges of estimated annual emissions (see Figure 15).

Figure 15. Bisingen municipality: buildings colour-coded by energy/CO_2 class, visualized with Cesium 3D Tiles in Unity Engine.

5.2 Qualitative Assessment

During development and testing sessions conducted with domain experts from STEG GmbH and the municipal administration, several qualitative observations were recorded. Planners noted that seeing energy classifications overlaid on real buildings in AR provided significantly better spatial understanding than equivalent 2D map views, particularly for identifying clusters of high-emission buildings that are candidates for district-level retrofit programmes. The sub-second update latency enabled rapid what-if testing, allowing a planner to evaluate 15-20 building modifications in 10 minutes, compared to the previous workflow where each change required switching to a desktop application and refreshing the map view. Real-time synchronization via WebSocket push allowed two planners wearing separate HoloLens 2 devices to observe each other’s edits instantly, facilitating on-site discussion and consensus-building. Additionally, the dual-input system (tap to view, hold to edit) was learned within 2-3 minutes by users with no prior HoloLens 2 experience, with the hold-threshold pattern effectively preventing accidental edits.

5.3 Transferability to Other Use Cases

Although the primary validation was performed on the Bisingen energy consulting scenario, the architecture is designed for transferability. The REST API abstraction layer and community-scoped WebSocket routing allow the system to be reconfigured for any municipality by changing only the community_id parameter and endpoint URL. Potential additional use cases include: (1) urban heat island assessment, where building surface properties and vegetation coverage could be visualized and modified; (2) building permit review, where inspectors could com pare proposed changes against existing 3D models on-site; and (3) cultural heritage documentation, where attribute level annotations on historic buildings could be edited collaboratively in AR. The modular design ensures that adapting to these scenarios requires only server-side data model changes, with no modifications to the client-side caching, coloring, or communication infrastructure.

5.4 Performance Comparison

The hybrid WebSocket architecture provides measurable improvements over pure HTTP polling in three critical metrics. Table 2 summarizes the quantitative com parison on the production deployment over the university Wi-Fi network.

With WebSocket push, the worst-case update latency is bounded by the server-side signal processing time plus network round-trip time, empirically measured at 200 500ms, representing a 240-600× improvement over the 120s polling interval. Bandwidth consumption drops from ∼72MB/hour (30 full-dataset polls × 2.4MB each) to ∼10KB/hour for a representative session with 20 building modifications, a 7,200× reduction, consistent with the theoretical analysis by Lubbers and Greco (Lubbers and Greco, 2010) and empirical benchmarks by Puranik et al. (Puranik et al., 2013). Server load likewise drops from 30 database queries per hour per client to zero additional queries, since push notifications serialize data directly from the *post_save* signal’s already-loaded model instance.

Metric	HTTP Polling (120 s)	Hybrid WS + Poll
Worst-case latency	120 s	200-500 ms
Avg.latency	60 s	200-500 ms
BW/hour (no changes)	~ 72 MB	~ 1KB
BW/hour (20 edits)	~ 72 MB	~ 10 KB
DB queries/hr/client	30	0
Connection model	Stateless	Persistent TCP
Failure behavior	Automatic	Auto fallback

Table 2. Performance comparison: HTTP polling vs. hybrid WebSocket+polling fallback.

5.5 Limitations

Several limitations have been identified during development and testing. On enterprise Wi-Fi networks with aggressive proxy timeouts (<60s), the WebSocket connection was dropped frequently, causing the system to fall back to Tier 2 (polling + reconnection). While the three-tier fallback ensures continued

functionality, the user experience during Tier 2 is degraded compared to persistent push. After approximately 30 minutes of continuous use with full tileset coloring active, the HoloLens 2 exhibits thermal throttling, reducing frame rates from 60 to ~45fps; this can be mitigated by reducing the *meshesPerFrame* parameter at the cost of slower initial coloring. The current system supports only thematic attribute editing (energy parameters, renovation status), and geometry modifications (e.g. adding building extensions or changing roof shapes) are not implemented. The qualitative assessment was conducted with domain experts rather than end users in a controlled study, and a formal usability evaluation with a larger participant group and standardized questionnaires (e.g. SUS, NASA-TLX) would strengthen the claims of practical value. Finally, although the architecture supports arbitrary CityGML LoDs, the current deployment has been tested only with LoD_2 data, and performance with LoD3 models, which have significantly higher polygon counts, remains to be validated on HoloLens 2.

6. Discussion

6.1 Comparison with Existing 3D City Model Platforms

Several established systems demonstrate the potential of 3D city models for urban applications, yet none combines immersive AR visualization with bidirectional attribute editing. The Virtual 3D City Model of Berlin (Döllner et al., 2006), one of the earliest large-scale deployments, focused on desktop-based visualization and investor information without real time editing capabilities. Similarly, the comprehensive re view by Noardo et al. (Noardo et al., 2021) catalogued over 29 use cases for 3D city models across more than 100 applications but identified visualization and analysis as the predominant paradigms, with interactive editing being largely absent. Our system extends these paradigms by closing the loop between visualization and data modification: planners not only observe energy classifications but modify building attributes in-situ, with changes propagated back to the authoritative geodatabase within 200-500ms.

The data encoding and streaming layer builds on the CityGML standard (version 3.0 (Kutzner et al., 2020)) and its compact alternative, CityJSON (Ledoux et al., 2019). While CityJSON reduces file sizes by approximately 80% compared to CityGML XML, neither format natively supports per-feature real-time updates without full dataset reloading. Our per-vertex coloring approach bypasses this limitation by decoupling the visual representation from the tile payload: the 3D Tiles geometry (Kolbe et al., 2021) remains static while vertex colours are updated in-place, avoiding costly mesh reconstruction. Schilling and his colleagues demonstrated glTF-based streaming of CityGML models in web browsers, achieving interactive frame rates for datasets of comparable size. Their approach, however, operates within a read-only web context. Our architecture extends the streaming model to a mixed reality client with write-back capability, where modifications flow through the Django REST API to the PostGIS database and are redistributed via the WebSocket channel layer (Schilling et al., 2016).

6.2 Significance of Bidirectional Editing in AR

The conceptual framework proposed by Wang et al. (X. Wang et al., 2013) identified the integration of BIM with AR as a promising direction for real-time on-site visualization, but their work remained at the conceptual level without implementing a functional editing workflow. Boga et al. (Boga et al., 2017) explored AR/VR integration with BIM for educational visualization in Unity, yet the system supported only passive viewing of pre-loaded models.

Our implementation advances beyond these conceptual and visualization-only approaches by providing a complete write path: the user modifies building attributes through the AR edit form, the backend recalculates colour is pushed to all connected clients. This bidirectional data flow transforms the HoloLens from a passive display device into an active planning instrument. During testing, domain experts confirmed that this capability eliminated the need to alternate between on-site inspection and desk top GIS tools, a workflow disruption consistently reported in urban planning practice (Boga et al., 2017).

6.3 HoloLens 2 as a Geospatial Platform

The HoloLens 2 hardware (Qualcomm Snapdragon 850, 4GB RAM) imposes significant constraints on geospatial applications. Wang et al. (W. Wang et al., 2019) demonstrated holographic 3D geographical scene visualization on the first-generation HoloLens but were limited to small-scale, pre-loaded datasets without real-time data synchronization. Zaccardi et al. (Zaccardi et al., 2023) benchmarked deep learning inference on HoloLens 2, reporting that computationally intensive tasks require careful optimization to maintain acceptable frame rates.

The implemented system addresses these constraints through three mechanisms: (1) the two-tier caching strategy minimizes network requests by maintaining both in-memory and disk caches, enabling instant startup from the persistent cache; (2) frame-rate-aware mesh coloring batches vertex updates across multiple frames to prevent frame-time spikes; and (3) the WebSocket push model eliminates periodic bulk downloads that would otherwise consume bandwidth and processing time on the constrained device. The thermal throttling observed after ~30 minutes of continuous operation aligns with findings by Guo (Balakrishnan and Guo, 2024) and Ungureanu et al. (Ungureanu et al., 2020), who reported similar thermal behaviour during sustained compute-intensive tasks on HoloLens 2.

6.4 Real-Time Communication Trade-offs

The hybrid WebSocket+polling architecture represents a pragmatic compromise between latency and reliability. Pure WebSocket deployments, while offering sub-second latency, are vulnerable to proxy timeouts and network interruptions common in enterprise environments. Pure HTTP polling, conversely, guarantees eventual consistency but introduces unacceptable delays for collaborative editing. The three-tier fallback strategy provides graceful degradation: the system transitions automatically from persistent push (Tier 1) through reconnecting with polling (Tier 2) to polling-only mode (Tier 3), ensuring that the planner never loses access to building data regard less of network conditions. This approach extends the findings of Pimentel et al. (Pimentel and Nickerson, 2012), who demonstrated WebSocket superiority over long-polling in controlled benchmarks, by adding an automatic reliability layer absent from their experimental setup. The measured bandwidth reduction of three orders of magnitude (~72MB/hour vs. ~10KB/hour for 20 edits; Table 2) is particularly significant for HoloLens 2, which relies on Wi-Fi connectivity with limited antenna performance compared to desktop systems.

6.5 Implications for Urban Planning Practice

The WebXR usability study by Rzeszewski and Orylski (Rzeszewski and Orylski, 2021) demonstrated that extended

reality can be effective for participatory urban planning when the interface resembles familiar practices. Our dual-input interaction model (tap to view, hold to edit) was designed following this principle, and the 2-3-minute learning time observed during the qualitative assessment supports their finding that intuitive affordances are critical for adoption.

The community-scoped architecture enables deployment across different municipalities by changing only the community_id parameter. Combined with the REST API design and WebSocket group routing, this positions the system as a reusable platform for urban energy consulting rather than a single-purpose prototype. The Noardo et al. (Noardo et al., 2021) GeoBIM benchmark highlighted software interoperability as a persistent challenge in the CityGML ecosystem; our use of 3D Tiles as an intermediate format, with semantic attributes maintained through the REST API rather than embedded in the tile payload, sidesteps many of the interoperability issues they documented.

7. Conclusion and Future work

In this paper, we presented a full-stack system architecture for enabling interactive editing and real-time visualization of semantic 3D building models in an AR environment. The deployment in Bisingen, Germany, covering 5,005 buildings, demonstrates that the system handles a real-world municipal building portfolio. The qualitative assessment with domain experts confirmed that AR based in-situ visualization improves spatial understanding of energy data, that sub-second update latency enables rapid scenario exploration (15-20 building modifications in 10 minutes), and that real-time synchronization supports collaborative on-site planning.

The four technical contributions, a hybrid WebSocket+polling communication layer, per building 3D Tiles vertex coloring, two tier disk/memory caching, and dual-mode AR/desktop interaction, collectively address the three literature gaps identified in the introduction of this paper. The community scoped, REST API-driven design enables transferability to other municipalities and use cases without client-side code changes.

Future work will address the limitations identified in this paper: a formal user study with standardized us ability metrics; geometry editing workflows to transform the system from a visualization tool to an active design platform; validation with LoD3 building models; voice-based command integration for hands-free operation; and supporting additional standard web APIs such as the “OGC API– Features” (OGC, 2022).

8. Acknowledgements

We gratefully acknowledge Prof. Dr. rer. nat. Michael Mommert for providing access to the high-performance workstation used in this research, and Prof. Dr. Volker Coors as well as Mr. Muhammad Alfakhori for providing access to the Microsoft HoloLens 2 device.

9. References

Balakrishnan, P., Guo, H. J., 2024. HoloLens 2 Technical Evaluation as Mixed Reality Guide. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).

Beck, D., Mitkiewicz, J., 2025. A systematic literature review of citizen science in urban studies and regional urban planning: policy, practical, and research implications. Urban Ecosystems, 28(2).

Boga, S. R. C., Kansagara, B., Kannan, R., 2017. Integration of augmented reality and virtual reality in building information modeling: The next frontier in civil engineering education. Mobile Technologies and Augmented Reality in Open Education, 233–261.

Cesium., 2025. Cesium For Unity Logo. https://cesium.com/learn/cesium-unity/ref-doc/

CesiumGS., 2026. glTF Extensions for 3D Tiles Next. https://github.com/CesiumGS/glTF/tree/3d-tiles-next

Cozzi, P., Lilley, S., 2023. 3D Tiles Specification Community Standard Approved (Issues 22-025r4).

Django Software Foundation, 2024. Channels Documentation (latest).

Döllner, J., Kolbe, T. H., Liecke, F., Sgouros, T., Teichmann, K., 2006. The Virtual 3D City Model of Berlin-Managing, Integrating, and Communicating Complex Urban Information. Proceedings of the 25th UDMS 2006 in Aalborg, Denmark, 15-17 May 2006, May, 15–17.

Fette, I., Melnikov, A., 2011. Websocket protocol.txt. 1–71. https://datatracker.ietf.org/doc/html/rfc6455

Grigorik, I., 2013. High Performance Browser Networking, Special Edition. In High Performance Browser Networking,. O’Reilly Media.

Guha, S., Francis, P., 2005. Characterization and measurement of TCP traversal through NATs and firewalls. Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC, 199–211.

Ketzler, B., Naserentin, V., Latino, F., Zangelidis, C., Thuvander, L., Logg, A., 2020. Digital Twins for Cities: A State of the Art Review. Built Environment, 46(4), 547–573.

Kolbe, T. H., 2009. Representing and exchanging 3D city models with CityGML. Lecture Notes in Geoinformation and Cartography, 15–31.

Kolbe, T. H., Kutzner, T., Smyth, C. S., Nagel, C., Roensdorf, C., Heazel, C., 2021. City Geography Markup Language (Citygml) Part 1: Conceptual Model Standard Standard Approved. 25–27.

Kutzner, T., Chaturvedi, K., Kolbe, T. H., 2020. CityGML 3.0: New Functions Open Up New Applications. PFG - Journal of Photogrammetry, Remote Sensing and Geoinformation Science, 88(1), 43–61.

Ledoux, H., Arroyo Ohori, K., Kumar, K., Dukai, B., Labetski, A., Vitalis, S., 2019. CityJSON: a compact and easy-to-use encoding of the CityGML data model. Open Geospatial Data, Software and Standards, 4(1).

Lubbers, P., Greco, F., 2010. HTML5 Web Sockets: A Quantum Leap in Scalability for the Web. SOA World Magazine, 1.

Noardo, F., Arroyo Ohori, K., Biljecki, F., Ellul, C., Harrie, L., Krijnen, T., Eriksson, H., van Liempt, J., Pla, M., Ruiz, A., Hintz, D., Krueger, N., Leoni, C., Leoz, L., Moraru, D., Vitalis, S., Willkomm, P., Stoter, J., 2021. Reference study of CityGML software support: The GeoBIM benchmark 2019-Part II. In Transactions in GIS (Vol. 25, Issue 2).

OGC., 2022. OGC API - Features - Part 1: Core corrigendum (Issues 17-069r4).

Pimentel, V., Nickerson, B. G., 2012. Communicating and displaying real-time data with WebSocket. IEEE Internet Computing, 16(4), 45–53. https://doi.org/10.1109/MIC.2012.64
Puranik, D. G., Feiock, D. C., Hill, J. H., 2013. Real-time monitoring using AJAX and WebSockets. Proceedings of the International Symposium and Workshop on Engineering of Computer Based Systems, 23(1), 110–118.

Rantanen, T., Julin, A., Virtanen, J. P., Hyyppä, H., Vaaja, M. T., 2023. Open Geospatial Data Integration in Game Engine for Urban Digital Twin Applications. ISPRS International Journal of Geo-Information, 12(8), 310.

Rzeszewski, M., Orylski, M., 2021. Usability of webxr visualizations in urban planning. ISPRS International Journal of Geo-Information, 10(11). https://doi.org/10.3390/ijgi10110721
Schilling, A., Bolling, J., Nagel, C., 2016. Using glTF for streaming CityGML 3D City Models. Proceedings of the 21st International Conference on Web3D Technology, Web3D 2016, February, 109–116.

Schlüter, M., Kwasnitschka, T., Bernstetter, A., Karstens, J., 2025. Rendering Large Volume Datasets in Unreal Engine 5: A Survey.

Ungureanu, D., Bogo, F., Galliani, S., Sama, P., Duan, X., Meekhof, C., Stühmer, J., Cashman, T. J., Tekin, B., Schönberger, J. L., Olszta, P., Pollefeys, M., 2020. HoloLens 2 Research Mode as a Tool for Computer Vision Research. 1–7.

Wang, W., Wu, X., He, A., Chen, Z., 2019. Modelling and visualizing holographic 3D geographical scenes with timely data based on the Hololens. ISPRS International Journal of Geo-Information, 8(12), 1–20.

Wang, X., Love, P. E. D., Kim, M. J., Park, C. S., Sing, C. P., Hou, L., 2013. A conceptual framework for integrating building information modeling with augmented reality. Automation in Construction, 34, 37–44.

Yao, Z., Nagel, C., Kendir, M., Willenborg, B., Kolbe, T. H., 2025. The New 3D City Database 5.0 - Advancing 3D City Data Management based on CityGML 3.0. ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences, X-4/W6-2025(September), 241–248.

Zaccardi, S., Frantz, T., Beckwée, D., Swinnen, E., Jansen, B., 2023. On-Device Execution of Deep Learning Models on HoloLens2 for Real-Time Augmented Reality Medical Applications. Sensors (Basel, Switzerland), 23(21), 1–15.