ChatGPT

AI+XR Spatial Media Unified File Format Specification

Introduction

The convergence of **Artificial Intelligence (AI)** and **Extended Reality (XR)** is driving demand for a new class of media format. Next-generation spatial experiences – from AR overlays to VR worlds – will blend dynamic 3D content with generative AI intelligence. To meet this need, we propose an **open, universal file format for AI+XR spatial media**. This format is designed for **real-time, interactive 3D/4D content** that can be shared across devices (mobile, AR glasses, VR headsets, wearables) and platforms (native apps and web). It draws on lessons from existing standards (glTF, USD, MPEG-I, WebXR) while extending them to support **AI-driven and procedural content**. The overarching goal is to enable rich, immersive "metaverse" scenes that **many users can access simultaneously** with consistent experiences, regardless of hardware differences [1] [2].

Key requirements include: **low latency rendering**, support for **massive, dynamic scenes** (e.g. MMO-scale worlds or volumetric video), and seamless integration of **AI-generated content**. AI is increasingly seen as the "intelligence backbone" of XR, with generative models creating world content on the fly from text prompts and powering NPC behaviors [3] [4]. Our format must therefore accommodate both **traditional assets** (meshes, textures, audio) and **AI-inferred content** (neural fields, procedural generation rules, etc.), with hooks for real-time AI services. It should remain **lightweight and open**, akin to glTF's philosophy of being the "JPEG of 3D" [5], yet extensible and future-proof like OpenUSD for evolving use cases. In the sections below, we detail the technical specification of this format, including its architecture, encoding pipeline, supported data types, and example use cases.

Design Goals and Requirements

1. Real-Time Performance Across Platforms: The format is optimized for **real-time, low-latency decoding** and rendering. Content encoded should stream or load quickly on a range of devices – from high-end VR rigs to power-constrained mobile AR glasses – without sacrificing too much fidelity. Techniques like level-of-detail, progressive streaming, and GPU-friendly compression are employed to minimize load times and frame drops. The target is to **sustain interactive frame rates (60+ FPS)** even for complex scenes, by allowing clients to easily skip or defer non-essential content based on device capability or network conditions [2] [6].

2. Heterogeneous 3D/4D Content Support: The format supports a **wide variety of spatial media types**, both static and dynamic. These include:

- **Geometric scenes and assets:** Standard 3D models (meshes, primitives) with PBR materials, animations, skinning, etc., as in glTF. Also point clouds and **volumetric meshes** for captured scenes.

- **Volumetric video & dynamic geometry:** Time-sequenced geometry or point clouds (e.g. for volumetric video captures or MMO worlds with continuous updates).
- **Neural scene representations:** AI-based encodings like **Neural Radiance Fields (NeRFs)** and **Gaussian Splatting** point-based radiance models. These allow photo-realistic view-dependent rendering via neural networks or point splats [7].
- **Procedural content:** Parametric or algorithmic content generation such as fractal landscapes, L-systems for plants, cellular automata, and physics simulations. The format can embed the **declarative rules or parameters** for these generators (rather than pre-baked geometry), enabling worlds that **evolve over time or respond to user interaction**.
- **Spatial audio:** Support for positional and **ambisonic audio** streams, including potentially AI-generated audio. Audio can be static clips or procedurally generated soundscapes, with metadata for 3D spatialization.
- **Shader programs & materials:** Custom shader code (GLSL, WebGPU WGSL, etc.) or node-based materials to allow special visual effects. The format can include shader source or references, with safety restrictions for cross-platform use (similar to how WebGL handles shader content).
- **4D simulations and animations:** Time-evolving simulations (particle systems, fluid or physics simulations) and interactive animations. Keyframe data, skeletal animations, morph targets as well as procedural animation scripts are supported.

3. Modular Structure: The file uses a **modular container structure** to accommodate these diverse content types. It is not a single monolithic blob but composed of sections/modules (possibly in a JSON-like manifest plus binary payloads). Creators can include only the needed modules (e.g. a purely volumetric video file would have the dynamic geometry module and audio, but might omit procedural logic or AI prompts sections). This modularity also means decoders can **gracefully ignore** sections they don't understand, ensuring forward compatibility. The design echoes the extensibility of glTF (which allows adding custom JSON sections as extensions [8]) and the layered composition of USD (where different aspects of a scene can be isolated in layers) [9] [10].

Each module has a clear specification (schema) and can evolve independently. For example, one module defines how neural fields are stored (it could even allow multiple alternative encodings of NeRFs), another defines how interactive behaviors are described. This structure will be governed by a community extension process (inspired by Khronos glTF extension registry) to allow new content types to be added in a consistent way.

4. Streaming and Progressive Loading: The format is built to **stream efficiently**. Rather than requiring the entire file to be downloaded before use, the file can be partially loaded – critical for cloud-based XR and large scenes. We leverage concepts from MPEG's **immersive streaming** work, such as providing a **manifest/index** of media segments and allowing streaming of time-sequenced buffers [11] [12]. For dynamic content, the format can specify multiple **alternative encodings** (e.g. high vs low bitrate) and let the runtime choose the best based on network or device performance [13] [6]. It also supports **progressive refinement**: an initial coarse scene can display quickly, then detail data streams in (similar to progressive mesh or image refinement).

5. AI Integration and Extensibility: A core differentiator of this format is native support for AI and generative content. The file can embed **metadata and hooks for runtime AI services**. For example, a scene file might include an *"NPC behavior"* block that contains a prompt or script for an LLM (Large Language Model) defining the character's personality and dialogue patterns. At runtime, the engine can use that prompt with a local or cloud AI service to generate dynamic dialogues. Similarly, an *"ImageGenerator"*

block could contain a Stable Diffusion prompt or seed that instructs an on-demand generation of a texture or skybox when the scene loads. These AI hooks are optional (the file can always fall back to default assets if offline), but they enable **prompt-based generation, memory/context awareness in scenes, and view-dependent or user-dependent behaviors**. Crucially, the format does not hardcode any specific AI model; rather it provides a general mechanism to reference external AI models or services (via URI, model IDs, or protocol) and to supply them with prompts or parameters. This ensures **future AI models** (including those we can't predict yet) can interface with the content. Security is considered: by default, AI calls might be disabled or require user permission, and the format can indicate whether external network access is needed for full experience.

6. Cross-Platform and Toolchain Compatibility: The format is intended to be **engine-agnostic and widely adoptable**. It should be decodeable in common engines and frameworks: e.g. Unity and Unreal via plugins or built-in support, Web engines like Three.js, Babylon.js, PlayCanvas, and WebXR frameworks (A-Frame, React Three Fiber) via JavaScript libraries, and content creation tools like Blender, Maya, Cinema4D through import/export add-ons. To ease adoption, the format will be **open and royalty-free** (under governance of a standards body or consortium). We plan to provide reference **open-source SDKs** for reading/writing the format in multiple languages (similar to how glTF has reference loaders in C++ and JavaScript). Exporters could be implemented in popular authoring tools – for instance, a Blender exporter that can pack a scene (including geometry, materials, physics settings, and even trained neural assets) into the new format, or a Unity plugin to save a game level as a deliverable AI+XR file. Likewise, conversion tools should exist to go to/from related formats (e.g. **convert glTF or USD into this format** and vice versa where feasible). This ensures existing content can be leveraged.

7. Open Governance and Evolving Standards: We embrace the principles of modern open standards. The format is **lightweight**, uses modern data encoding (like binary buffers, JSON or similar text for descriptors), and is **extensible** via a community-driven extension schema (comparable to glTF's extension mechanism [8]). It will participate in or align with relevant standards bodies – for example, it could be managed under the **Khronos Group or the Metaverse Standards Forum**, to gather industry-wide input and adoption. We will also track and incorporate relevant emerging standards: e.g. the ongoing **MPEG-I** standards for XR streaming, the W3C **Immersive Web** specifications (WebXR Device API for runtime, WebGPU for rendering, etc.), and **OpenXR** (the cross-platform XR device API by Khronos) to ensure compatibility with how content is presented on devices. By designing the file format in concert with these efforts, we ensure it fits into the broader ecosystem (for instance, an OpenXR application could directly consume a scene file, or a WebXR site could stream this format). Community governance will help adapt the format to new needs – whether it's **neural interface data** (if future brain-computer interfaces become part of XR) or integration with **robotics** (e.g. a robot sharing an AR scene might read the same format for spatial understanding).

Inspirations from Existing Standards

Before detailing the format, we highlight key lessons from existing 3D/XR standards that inform our design:

- **glTF 2.0 (GL Transmission Format):** glTF is often called *"the JPEG of 3D"* for its efficiency in delivering 3D models in real-time [5]. It uses a JSON descriptor plus binary buffers (.glb for all-in-one) and focuses on compactness (with techniques like mesh and texture compression) and fast load by directly matching GPU-friendly data structures. glTF's strengths are **simplicity, broad adoption, and extensibility**. It supports PBR materials, animations, skinning, etc., but notably **does not embed behavior or heavy dynamics** – it's mostly static scene description for models or small scenes [14].

We emulate glTF's lightweight approach: JSON for high-level structure and binary blobs for data, an extension mechanism for new features, and keeping things human-readable when possible. glTF's limitation is that it's aimed at singular assets or static scenes, not full interactive worlds [15] [14]. Our format extends glTF's philosophy to dynamic and AI content while remaining backward-compatible where possible (e.g. a basic static model in our format could essentially be a glTF under the hood). We also plan to support **glTF embedding**: traditional models might simply be included as glTF portions inside our file for compatibility.

- **USD (Universal Scene Description / OpenUSD):** USD (originally by Pixar, now an open standard) provides a rich **scenegraph framework for large-scale scenes**. It allows layered files, instancing, variants, and fine-grained editing of huge scenes (think entire movie sets or industrial digital twins) [16] [9]. USD's design is aimed at content creation pipelines rather than transmission – it prioritizes completeness and flexibility over compactness. A USD can describe **complex hierarchies of hundreds of objects with relationships**, whereas glTF tends to encapsulate a single object or small scene [17] [18]. However, USD files (especially ASCII .usda) can be large and not optimized for real-time loading, and they have no built-in compression beyond optional USDZ (which is essentially a ZIP container) [19] [20]. We learn from USD by adopting a **comprehensive scenegraph model** (our format can describe an entire scene with many objects, not just one model). We also like USD's **modularity**: e.g. separating geometry, shading, physics layers. Our format similarly will allow separation of concerns (one section for visual geometry, one for logic, etc.). However, we aim to avoid USD's performance penalty by using binary packing and streamlining for runtime. Think of it this way: if glTF is the JPEG and USD the PSD (Photoshop file) of 3D [21], our format tries to be **the interactive XR equivalent of a video stream** – compact enough to stream, but richer than a single model. We also ensure compatibility: since USD is becoming an industry standard (now called OpenUSD), our format could offer **interop with USD** (e.g. export a static version of the scene to USD for editing, or embed USD schemas if needed for complex properties).

- **MPEG-I and Immersive Media Standards:** The Moving Picture Experts Group (MPEG) has been developing standards for **immersive media (ISO/IEC 23090 series)**. Notably, MPEG recently adopted **glTF 2.0 as a basis for scene description** in MPEG-I, adding extensions for dynamic streaming [22] [23]. The **MPEG-I Scene Description** (23090-14) defines glTF extensions like `MPEG_media`, `MPEG_buffer_circular`, and `MPEG_accessor_timed` to handle **time-varying data, video textures, and interactive streaming** [24] [25]. This architecture cleanly separates the **presentation engine (rendering)** from the **media access function** that fetches and decodes streamed content [26] [6]. We incorporate these ideas: our format uses a *media manifest* for dynamic assets and allows a **circular buffer mechanism** for continuous streams (useful for volumetric video or live sensor input) [27] [28]. By doing so, a runtime can be "content-agnostic" – e.g. a mesh's frames might be delivered via MPEG's V-PCC point cloud codec or as a sequence of glTFs, but the engine just sees a changing buffer. We intend to stay aligned with MPEG's work so that content encoded in our format can leverage standardized compression (for example, **using MPEG V-PCC for point cloud sequences** – which can compress dynamic point clouds 125:1 with good quality [29]). Additionally, MPEG-I includes Immersive Audio and Immersive Video standards; our format can act as the *container* that ties together those streams with spatial context.

- **WebXR and OpenXR (Immersive Web Specifications):** While not file formats, the API standards for XR inform how our content is used. **OpenXR** defines a common interface for VR/AR runtime engines – by aligning with it, our format can ensure that content like hand interactions or anchor poses are

expressed in ways the runtime can apply. **WebXR** (W3C's browser API) similarly defines coordinate systems, reference spaces, and frame update loops for XR content. Our format may include metadata like "this object is anchored to real-world floor" or "this scene uses gravity aligned with Earth" that correspond to XR runtime concepts. We also consider the **security and performance model** of the web: e.g. any executable code embedded (shaders or scripts) should be sandboxable akin to how browsers handle untrusted content. The format's design will support the **3D Commerce and Khronos 3D formats** guidelines as well, ensuring things like units, coordinates, and lighting models are unambiguous across tools [30]. By drawing from WebXR, we also ensure our format can integrate with web-based experiences – for instance, an HTML webpage could include an `<model>` viewer for our file just as today they do for glTF/GLB.

- **Legacy and Niche Precedents (X3D, VRML, Tilt Brush):** Earlier attempts at interactive 3D formats, like **VRML/X3D**, allowed embedding of behaviors (via scripts and event routing in the scene). Those formats taught us the importance of **script integration** but also showed the pitfalls of overly complex XML syntax and lack of industry push. We aim to avoid X3D's verbosity while achieving its interactivity. Another example, **Google's Tilt Brush (Open Brush)**, uses a bespoke format `.tilt` for VR drawings: essentially a **ZIP with a JSON/binary payload** describing strokes [31]. Each stroke records points, orientation, pressure, etc., which is a kind of procedural geometry (the stroke can be replayed or remade at different resolution) [32]. We take inspiration here for our **procedural stroke/curve representation** – rather than storing every vertex of a drawn stroke, we can store key control points and let the viewer reconstruct the smooth curve or ribbon. This dramatically reduces file sizes for things like brush strokes or paths. Likewise, game engines often store **spline paths, particle system settings, or terrain heightfields** instead of fully realized geometry – our format will include analogous constructs. We also acknowledge **Collada** (an XML 3D format from Khronos pre-glTF) for interchange, but glTF largely supersedes it, so we focus on glTF's leaner approach.

The table below summarizes a comparison of major formats and how they relate to the proposed format:

Format	Use Case Focus	Key Strengths	Limitations
glTF 2.0	Web and mobile delivery of 3D assets (models, small scenes) [30].	Very efficient, compact transmission; PBR materials; broad support (the "JPEG of 3D") [5]. Extensible via JSON extensions [8].	Limited to mostly static content (animations but no dynamic streaming); no built-in behavior logic; not intended for huge scenes [15].
USD (OpenUSD)	Content creation interchange; large complex scenes with many objects [17] [16].	Rich scene description (layers, variants, instancing); supports many features (geometry, rigging, physics, etc.) in one format; great for collaboration and editing.	Heavy and verbose (larger files) [19]; requires conversion or simplification for real-time use; lacks advanced compression (uses ZIP if at all) [19] [20].

Format	Use Case Focus	Key Strengths	Limitations
MPEG-I (Scene & Media)	Streaming interactive XR scenes (with timed media, point clouds, video, etc.).	Dynamic content support via glTF extensions (timed accessors, circular buffers) [25]; integrates with video/point cloud codecs for efficiency; adaptive streaming based on network [6] [13].	Still evolving (spec finalized in 2024); focuses on playback of authored content rather than user-generated or AI content; uses vendor-specific extensions to glTF (needs broad adoption in engines).
Proposed AI+XR Format	Unified delivery of interactive AI-driven XR content (games, simulations, volumetric media, etc.).	Real-time optimized (streamable, LOD support); **supports AI generative hooks and procedural logic**; modular and extensible; cross-platform by design; open governance.	New ecosystem needs to be built (requires engine support and tooling); must balance flexibility with performance; some advanced features depend on future engine capabilities (e.g. neural inference hardware).

(Table: Comparison of existing 3D/XR formats and the proposed format.)

Format Architecture and Structure

At a high level, the AI+XR spatial format consists of a **scene description** plus a set of **resource encodings**, packaged together. The scene description is a structured text (e.g. JSON or JSON-derived) that lists the objects in the scene, their properties, and references to resources (geometry data, media streams, etc.). The resources are typically binary blobs (meshes, textures, neural network weights, etc.), either embedded in the file or referenced externally (for streaming or reuse). We envision the file to be a **single package** (e.g. a binary .**XRF** – *XR Format* – file) which internally might be like an archive (similar to USDZ or glTF's GLB) so that all data stays in one file for portability. Alternatively, it could be a directory format (like an unzipped package) when used in development, but distributed as one file.

Concretely, the format could use a container like **ZIP or an OpenDocument style package**, with a manifest describing contained files. For example, an `.xrf` file could be a ZIP whose entries include `scene.json` (the main scene graph and metadata), and various subfiles like `geom.bin` (binary geometry), `textures/...` images, `nerf.dat` (neural field parameters), etc. A custom binary container (like glTF's GLB chunk system or a lightweight equivalent of MP4's atom structure) could also be used – this is an area for community input, weighing readability (ZIP+JSON is easy to inspect) vs. performance (a single binary blob may be faster to parse). **Open Brush's .tilt** approach is instructive: they use a zip with a JSON header and a binary `data.sketch` inside [31]. We generalize this: our format's **header JSON** might enumerate the modules and provide high-level metadata (like scene title, format version, and requirements), followed by contiguous binary sections per module.

Scene Graph and Entities: The scene is organized as a set of **nodes (entities)** with transform hierarchies (position, rotation, scale), similar to the node graph in glTF or Unity. Each node can reference one or more **components**: for example, a visual mesh, a collision shape, an audio source, a script, etc. This entity-

component system makes the format flexible – new component types (like “NeuralRenderVolume” or “ProceduralGenerator”) can be added without redefining the entire format. A node could even have multiple representations of content for fallback: e.g. a node might contain *either* a high-quality neural field component *or* a fallback low-poly mesh component for devices that can’t do neural rendering. The node graph supports **grouping and instancing**: instancing means the same resource can be reused under multiple nodes (like one tree model replicated 100 times with different transforms). It also naturally encodes parent-child relationships (so complex objects can be made of sub-objects, and moving the parent moves children).

Resource References: Within the scene description, references link to the actual data. For instance, a mesh component on a node might have `"mesh": "#geom/mesh12"` indicating that the geometry data is in the `geom.bin` resource with identifier 12. We plan to use a uniform referencing scheme (like URIs or IDs) for all resources, whether internal or external. External references (for streaming content) could be URLs or content-addressable hashes (if using IPFS or cloud storage). The **MPEG_media** extension concept is useful here: it defines a descriptor for external media with multiple alternatives and protocols [25] [13]. In our format, an external media reference could list options like: a URI template for DASH streaming, a WebRTC stream ID for live content, or a fallback to a local video file. The player will pick the optimal source automatically.

Modules and Extensions: The core specification will define the baseline modules: e.g. **Geometry**, **Materials**, **Animation**, **Audio**, **AI**. Each module has a schema. For example, the *Geometry module* defines how mesh data is stored (buffer formats, attribute semantics like positions/normals/UVs, possibly using glTF’s accessor concept). The *AI module* might define a structure for an “AI Agent” component (containing fields for an LLM prompt, state memory keys, etc.) or for a “Generative Texture” (with fields for prompt, resolution, style). Users can also add **custom modules** via an extension mechanism. Perhaps a company or research group wants to include **haptic feedback patterns** in a scene – they could propose an `EXT_haptics` module with its own data schema, and if a runtime doesn’t understand it, it simply ignores that component (the scene still functions without haptics). We will maintain an extension registry (similar to Khronos’) to avoid conflicts and encourage reuse of common extensions.

Metadata and Semantics: The format encourages including semantic metadata to aid AI and future-proofing. For instance, objects can have tags or descriptions (useful for an AI agent to know that a node is “a red cube that is an apple”). Spatial anchors (for AR) can be marked with real-world coordinates or geolocation. Units are specified explicitly (meters by default, following glTF’s convention). We also include fields for **environmental properties** (like global gravity vector, ambient lighting, etc.) to ensure consistent behavior across platforms.

Security Model: If the file contains any executable logic (shaders, scripts, AI prompts that could call external services), we include a manifest of required privileges. For example, a script component might be marked as requiring network access or file access, which a viewer could sandbox or prompt the user to allow. By default, content is treated as untrusted (especially important for web). This is analogous to web browsers handling WebGL shaders (which are restricted in functionality for safety) or prompting for microphone/camera access. The format will recommend sandboxing methods (perhaps using WebAssembly for any embedded code, or limiting scripting to a safe API).

Example Structure: To illustrate, here’s a simplified outline of what a scene JSON might look like (for readability we show a JSON-like pseudo-schema):

```
{
  "formatVersion": 1,
  "modules": ["Geometry", "Material", "AI", "Audio", "Procedural", "Animation"],
  "scene": {
    "nodes": [
       { "id": "room1", "transform": [...], "children": ["table1","npc1"] },
       { "id": "table1", "transform": [...], "components": {
             "geometry": {"ref": "geom:mesh45"},
             "material": {"ref": "mat:woodMaterial"}
         }
       },
       { "id": "npc1", "transform": [...], "components": {
             "geometry": {"ref": "geom:humanoidLOD0"},
             "animation": {"ref": "anim:walkCycle"},
             "aiAgent": {
                 "behaviorModel": "gpt4",
                 "prompt": "You are a shopkeeper in a medieval VR market...",
                 "memoryRef": "aiMem:npc1memory"
             }
         }
       }
    ]
  },
  "resources": {
     "geom": "geom.bin",
     "mat": "materials.json",
     "anim": "anims.bin",
     "aiMem": "memory.json"
  },
  "externals": [
     { "id": "envSkyTexture", "type": "image/ktx2", "uri": "https://example.com/
sky.ktx2" },
     { "id": "pointCloudStream", "type": "video/pc", "dashManifest": "https://
cdn.com/pc_stream.mpd" }
  ]
}
```

In this hypothetical snippet, we have nodes representing a room, a table, and an NPC. The table uses a geometry and material; the NPC uses geometry, an animation, and an `aiAgent` component with a prompt. The resources section indicates geometry is in a binary blob, materials and memory might be in JSON, etc. External resources like a sky texture or a point cloud stream are listed separately with URIs. (The actual format syntax may differ, but this conveys the idea of referencing data and mixing local and external content.)

Encoding and Decoding Pipeline

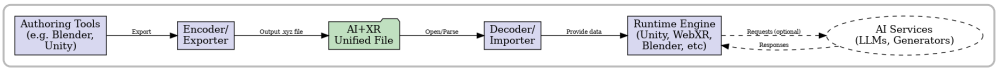

Figure: End-to-end pipeline for creating and consuming the AI+XR format. Content creators use authoring tools (Blender, Unity, etc.) with an exporter/encoder to package assets and AI metadata into the unified .xrf file. At runtime, a loader/importer in the target engine decodes the file, reconstructing the scene and its components. The engine (Unity, WebXR, etc.) then renders the scene, and if specified, connects to external AI services (e.g. an LLM or generative model) via prompt or API calls embedded in the file. The dashed arrows indicate optional real-time AI queries (for things like NPC dialogue or on-the-fly asset generation) [3] .

The **encoding pipeline** begins with content creation. Artists and developers design scenes using familiar tools: modeling in Blender or Maya, texturing, setting up animations, perhaps training a NeRF model on captured data, etc. New AI content might be authored by prompt (e.g. using tools like Lens Studio's generative design feature [33]). To export, an **exporter plugin** (or a standalone converter) takes all these elements and **serializes** them into our format. This involves: exporting meshes (with possible simplification or LOD generation), baking or exporting materials (or converting shader graphs to a portable format), saving animation keyframes or baking physics as needed, collecting neural network weights or point clouds, and gathering any scripts or prompts. The exporter also compresses data where possible (textures to KTX2, geometry through Draco/mesh compression, etc.) and splits data into the appropriate modules. The output is the single `.xrf` file (or a folder that can be zipped to .xrf). During this process, **validation** tools will ensure the file meets the spec (e.g. correct JSON, all resource refs valid, size limits not exceeded for certain real-time sections, etc.).

The **decoding pipeline** happens when a user or application loads the file. The target engine (say a web app using Three.js or a game in Unity) either natively supports .xrf or uses a provided library. The steps typically are: read the file header/manifest, determine which modules are present, load the core scene graph, then asynchronously load heavy resources (potentially from network if they are external). Because performance is critical, the format is designed such that an engine can start rendering something *quickly*. For example, the scene might contain a simple bounding box or proxy mesh that can show up while a high-detail mesh or neural field is still loading in the background.

If the file indicates **AI service usage**, the engine's loader will set up those connections. For instance, if an NPC has an AI agent component with a specified model (`behaviorModel: "gpt4"` in the example), the engine might instantiate a client that connects to an AI API (this could be a local model runtime or cloud API). The file may also include an **orchestration script** (e.g. a small state machine or prompt template) that tells the engine when to query the AI (e.g. "when player approaches within 2m, send this prompt to the LLM, and use the response to drive the NPC's dialog UI"). Such logic could be encoded in a high-level state machine or trigger system within the file. The engine would either directly execute it (if it's a known scripting schema) or rely on a provided scripting runtime (maybe a safe Lua or JavaScript environment).

Cross-Engine Compatibility: Because multiple engines need to support this format, the spec avoids any features that assume a specific render pipeline or physics engine. For instance, if physics behavior is included, it would likely be in a descriptive form (collision shapes, masses, constraints) rather than engine-specific binary data, so that Unity, Unreal, or a web physics library can all interpret it. Similarly, shaders might be provided in multiple languages or an intermediate form (e.g. SPIR-V or WGSL) to be used on

different platforms. The decoding library can handle conversion – e.g. if a material uses a certain shader not natively supported, the library might swap in a simplified PBR material or use a fallback technique.

Performance considerations: The decode pipeline is built to minimize CPU overhead. For static geometry and textures, we leverage GPU-friendly formats (like glTF does) – e.g. reading a binary vertex buffer and directly creating a WebGL/WebGPU buffer without per-vertex processing. For compressed geometry (like Draco), there might be a small decode step, but libraries exist for those. Neural network weights could be large (a NeRF might have tens of MB of parameters); these might be stored in a standardized way (perhaps using Tensor formats or ONNX). The engine could lazy-load them and use GPU acceleration (via shader or machine learning inference API) to run the network. **Memory** is also a concern; our format will support memory-efficient patterns like streaming data and releasing it when not needed (e.g. only keep a few frames of a volumetric video in memory at once, using a circular buffer as defined in MPEG extensions [27] [28]).

Real-Time Streaming and Cloud Integration

For truly large-scale or live content, the format supports a mode where the file acts as a **manifest for streaming data**. In this scenario, the initial .xrf file might be small – containing just the scene setup and references to streaming endpoints. The heavy content (like a 4D volumetric video, or a huge photogrammetry environment) is then streamed in chunks as needed.

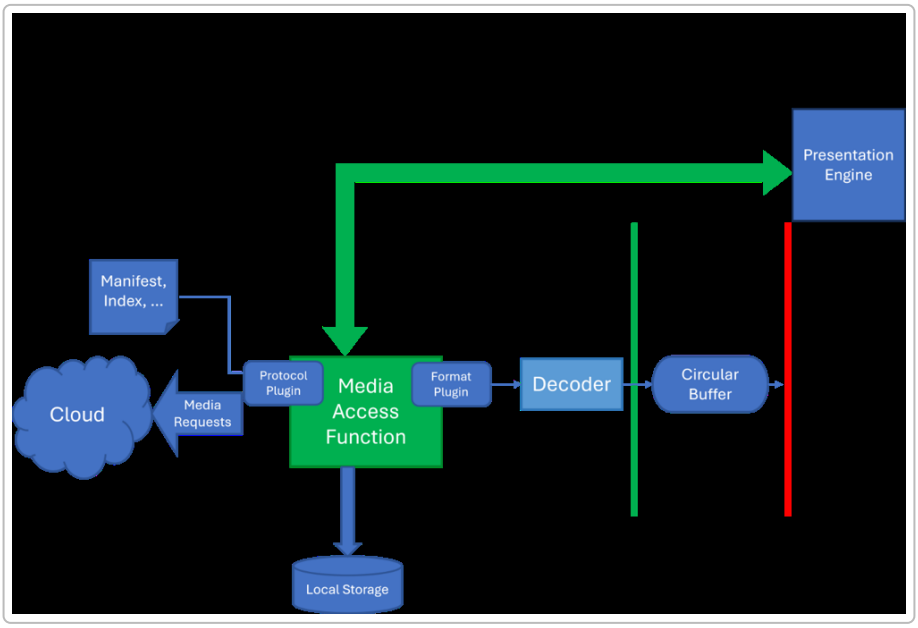

Figure: Streaming architecture (adapted from MPEG-I Scene Description). The Media Access Function (green box) handles retrieval and decoding of media streams from the cloud or local storage, providing decoded data to the engine via a buffer interface [26] [12]. The presentation engine (blue) is thus abstracted from whether content comes from network or local. This design enables adaptive streaming: the Media Access Function can choose different pipelines (protocol plugins, codecs) based on network conditions or device capability [6] [13]. For example, it might switch to a lower-bitrate point cloud stream if bandwidth drops. A circular buffer (right) is used for smooth playback of timed media, with concurrent write/read to avoid frame drops [27] [28].

In practice, this means our format can be used for cloud-driven experiences: Imagine a multi-user AR event where the environment is being scanned and updated in real-time – an initial file sets up the scene and then incoming data (perhaps via WebRTC or MQTT) updates object positions or streams new point clouds into the scene. The file format's role is to describe how to handle those streams. We provide constructs for **temporal content**: e.g. a "timed mesh" that changes every frame (with a pointer to a stream ID), or an "audio stream" synchronized to a global clock.

One important aspect is **synchronization**: The format can include timing information so that multiple streams (e.g. a volumetric video and its spatial audio) stay in sync when played. It may specify presentation timestamps or designate one stream as the master clock. This aligns with MPEG's approach of treating timed data uniformly and providing sync via the scene description [27].

LOD and selective streaming: To reduce bandwidth, the format supports Level-of-Detail (LOD) definitions for networked content. For example, a city-scale scene might have a low-detail model for far-away blocks and high-detail only streams in when the user comes near. The scene description can encode LOD thresholds (like a certain model has an alternative resource when closer than X meters or when screen size exceeds Y pixels). The engine could subscribe/unsubscribe from certain streams based on user position or device capability. This way, a mobile AR user might only ever request low-res data, whereas a high-end VR client can opt into the highest detail stream.

Caching and local fallback: The format can include checksums for streamed content and allow local caching. If a device has previously cached parts of the content (say, common assets), the Media Access Function can load from local storage instead of refetching【22†】. The format could even package a base version of the content inside the file as a fallback (e.g. a static low-quality capture), so that if live streaming fails, the user still sees something. A good example is an AR conference call: the file could contain a static avatar model as fallback, but if the live volumetric video stream is available, it uses that for a real-time hologram of the person. This kind of design ensures robustness: the experience degrades gracefully rather than breaking.

Data Compression and Representation Strategies

A crucial design decision is how various data types are represented and compressed. The format is **agnostic to the exact compression codecs** for things like geometry, images, and video – it relies on existing methods and allows plugging in new ones. The guiding principle is to choose a representation that balances **file size, decode speed, and fidelity**. There are broadly three categories of representations we consider: **explicit (binary) data**, **procedural descriptions**, and **neural network-based representations**. Each has pros and cons:

Representation	Description	Advantages	Challenges
Compressed Binary Data (explicit meshes, images, etc.)	Traditional explicit data encoded with compression (e.g. mesh as triangles + Draco compression; textures with PNG/ JPEG or KTX2; animation curves quantized).	*Fast to render* (often one decode step to GPU-ready data); *deterministic quality* (no runtime surprises); leverages mature codecs (Draco, MPEG video, etc.).	Files can still be large for complex scenes; fixed resolution (cannot easily increase detail beyond what's stored); lacks flexibility (content is baked, not easily changed without re-export).
Procedural Descriptors	Content described via algorithms or rules rather than raw data. Examples: a terrain generated by a noise function with a seed, a building generated by a grammar, an effect defined by particle system parameters, or an image synthesized by a mathematical formula.	*Tiny file sizes* for huge content (a whole forest generated from a few rules); *Adaptive detail* (can tesselate or refine more at runtime if needed); dynamic and customizable (parameters can be tweaked on the fly, enabling endless variety).	Requires computation at runtime (potential performance cost, especially on low-end devices); quality depends on the algorithm (may not capture real-world complexity easily, unlike captured data); determinism and sync can be an issue (procedural randomness must be controlled to ensure all viewers see the same result).
Neural / AI-Based (learned models, e.g. NeRFs, neural textures)	Data encoded as weights of a neural network or latent vectors, which a runtime network interprets to produce content (images, 3D scenes, etc.). For instance, a NeRF stores a neural field that outputs color+density given a 3D coordinate and view direction.	*High compression ratio* for certain content (neural fields can capture detailed real-world scenes in tens of MB where explicit point clouds might be gigs); *Continuously detailed* – effectively an infinite resolution within quality bounds (no discrete polygons); can capture subtle phenomena (light field effects, transparency) via the neural function.	Inference can be *computationally heavy* (requires neural network acceleration hardware or time); tools and standards for neural data are nascent (interoperability issues, as there's no single standard for "NeRF weights" yet); difficult to edit or manipulate granularly; some view-dependent networks may not generalize beyond their training (lack of dynamic flexibility).

Meshes and Geometry: For polygonal meshes, we'll use state-of-the-art geometry compression like **Draco** or **Meshopt**, which glTF already employs (reducing mesh data by 5-10x with negligible runtime cost). Point clouds (for static scenes) could be quantized and compressed similarly; for dynamic point clouds, we lean

on **MPEG's V-PCC** where sequences of point cloud frames are turned into video streams (efficiently handled by existing video decoders) [34]. If a scene uses procedural geometry, e.g. a fractal, the format might store a short **shader or script** that generates the mesh on the fly (perhaps using a compute shader or WASM module). We will allow **implicit surfaces** definitions too (like a signed distance field function) which a viewer can raymarch or tesselate as needed. This is useful for smooth organic shapes at arbitrary resolution.

Textures and Images: These will use standard image compression – KTX2 with Basis U supercompression is a likely default (as used by glTF 2.0), since it supports GPU-ready compressed formats (ASTC, etc.) and lower memory use. For dynamic or AI-generated textures, one could embed a small generative model (like a GAN or Diffusion model) or a parametric description (like "a checkerboard pattern of colors X and Y at scale Z"). However, storing a full generative model for a single texture might be overkill; more often, an AI-generated texture would be computed **ahead of time and stored as an image**. Where it gets interesting is **personalization**: imagine an object whose appearance is generated from the user's name or some prompt – the file could include a prompt template and the viewer's app locally runs a generative model to produce the texture. We will support basic 2D vector graphics as well (SVG or a simple shape grammar) to allow resolution-independent decals or UI elements in XR.

Animation and Physics: Animation data (bone animations, morph targets) can be sizable but compressible via curve fitting or even neural compression (some research uses neural nets to compress skeletal animations). We'll likely stick with tried methods: keyframe reduction, splines, maybe OpenTimelineIO integration for complex timelines. For physics, rather than storing full simulation data, we store initial conditions and let the engine simulate. If a physics simulation is pre-authored and critical (e.g. a complex destruction animation), it may be stored as baked animation or even as a **4D mesh sequence**. Those sequences can be compressed via MPEG techniques similar to point clouds or simply kept as delta keyframes.

Spatial Audio: Audio streams can use standard codecs (AAC, Opus, etc.) for compression. We include support for **ambisonic or binaural audio** to allow true spatial sound. The file can mark an audio track with its spatialization metadata (order of ambisonics, orientation). We foresee linking audio to objects (e.g. a node has an audio component that plays a sound at that location). If audio needs to be generated (like text-to-speech for an AI character), the format might include the script or captions, and the client can generate speech via an AI voice model. Alternatively, pre-generated voice lines are included as clips.

Neural Fields (NeRF, etc.): The format can incorporate neural scene representations in two ways: *explicitly* – by storing a specific format's data (e.g. a sparse grid of features and a small MLP, as in many NeRF variants) – or *generically* – by treating it as an external model resource. For the explicit approach, we could define a schema: e.g. a NeuralRadianceField component contains a voxel grid of feature vectors (perhaps compressed) and a set of network layer weights that the engine knows how to use to reconstruct color. There are emerging implementations to reference: NVIDIA's **Instant NeRF (instant-ngp)** uses a hash-grid encoding which could be stored; others like **SMERF** propose streamable partitions for large scenes [35] [36]. SMERF's idea of splitting a large environment into cells each with a small NeRF is something our format can adopt: the scene can have multiple neural field components each covering a region, allowing **large-scale coverage with efficient streaming** (only load the cells near the user) [35] [36]. The generic approach would be to reference an external neural model file (perhaps an ONNX or a custom .nerf file) that a specific engine plugin can read. Initially, we might include basic support for one or two popular neural formats (for example, a `NeuralSplat` component that lists a bunch of Gaussian blobs with position, radius, color – since Gaussian Splatting has proven a fast, **real-time method for high-fidelity view synthesis** [7]).

Indeed, Gaussian splats reduce the gap between neural and explicit – they're kind of an explicit point cloud with a Gaussian volume rendering, which can be stored by just listing each Gaussian's parameters. That's much lighter than storing millions of points or voxel grids, and has shown excellent results (Kerbl et al.'s Gaussian Splatting achieves 1080p 30fps on a neural-rendered scene by using ~500k ellipsoidal splats instead of a heavy network [7]).

To summarize, our format doesn't invent new compression algorithms but rather **chooses the best representation per content type** and allows multiple options. An author might decide to use procedural generation for one part of the scene and explicit meshes for another, and our format will carry both seamlessly. It's similar to how a game engine might combine hand-modeled assets with auto-generated terrain and AI characters.

Schema Overview and Components

In this section we outline the proposed schema of the format, enumerating the primary components and how they relate. This is a conceptual overview – the actual implementation may use JSON, XML, or another notation, but the concepts remain.

File Header and Metadata

Each file begins with a small header containing: format version, a unique identifier, and possibly a table of contents for quick indexing. The header can also store global metadata like the coordinate system (e.g. right-handed Y-up, like glTF) and units (meters), as well as the **engine requirements** (for example, minimum API level or supported extensions needed to fully use the file). If the file uses extensions, they are declared here so a parser can quickly know if it supports them.

Example:

```
{
  "format": "XRF",
  "version": 1.0,
  "extensionsUsed": ["AI_agentic", "Physics_basic"],
  "generator": "BlenderXRF 0.9",
  "timestamp": "2025-05-15T01:54:00Z"
}
```

This indicates the file was generated by a Blender exporter, uses two extensions (AI_agentic, Physics_basic), and conform to XRF v1.0 spec.

Scene Graph (Node Hierarchy)

The scene graph defines the **hierarchical structure** of the scene. It typically contains one root node (or multiple roots if it's an unconnected scene) which can have child nodes. Each node has: a unique ID, a transform (position/rotation/scale), and a list of components. Child nodes are listed by reference. This structure is quite like glTF's `nodes` and `scenes` fields. We also allow nodes to specify certain

environmental roles – e.g. a node can be marked as “main camera” or “floor anchor” if relevant (especially for AR, linking virtual content to real world).

A node can be **active or inactive** by default (some content might be present in file but disabled unless triggered). Also, for convenience, we might allow grouping of nodes by layers or tags (like “environment”, “characters”, “UI”) to allow easy toggling or isolation by the engine.

Components and Modules

The following are key component types (modules) defined in the base spec:

- **Transform (built-in):** Every node has a transform (3D affine). Also supports animation channels to allow it to move over time.

- **Geometry:** Attaches a geometric shape to a node. There are subtypes:

- *MeshGeometry:* A reference to mesh data (one or more vertex buffers and index buffers, possibly with LODs). It could also include a reference to a collision mesh or simplified version for physics.
- *PointCloud:* A set of points with attributes (or a reference to such data).
- *ImplicitSurface:* A mathematical description (e.g. “sphere radius R” or an SDF shader).
- *NeuralField:* A reference to neural scene data (could include bounds, and reference to weight data).
- *Volume:* A 3D grid or volumetric texture (for volumetric fog or medical data etc.).

- **Material/Shading:** Defines the appearance of geometries. Could be:

- *PBRMaterial:* A standard physically-based material with textures for baseColor, metalness, roughness, normal, etc. (We base this on glTF 2.0’s material definition for compatibility).
- *ShaderProgram:* A custom shader, possibly provided in multiple language variants (GLSL, HLSL, SPIR-V). Might be limited to surface shaders or postprocess effects for security. We might also allow **material graphs** (like a small graph that can be translated to shader code).
- *VolumeMaterial:* For volumetric rendering (e.g. volume raymarching parameters, transfer functions).
- The material system will also handle **view-dependent effects** – for example, a NeuralField has an implicit material that’s view-dependent by nature, or a “mirror” material might indicate the need for planar reflections, etc.

- **Animation:** This can attach animations to a node (or globally). Animations can target transforms, morph weights, or other properties. We follow glTF’s animation model (channels and samplers). Additionally, trigger-based animations (play on event) can be declared.

- **Physics:** (If included) defines physical properties like rigid body, colliders, joints. A node with a geometry might have a physics shape (auto-generated from the mesh or a separate simplified collider). Properties include mass, friction, etc. We might allow a “PhysicsScene” global component to define gravity and other simulation settings.

- **Audio:** Attaches an audio source to a node. Properties: reference to an audio clip (or stream), volume, looping, spatial properties (mono/stereo/ambisonic, spread angle, etc.). If it's spatial audio, the node's transform is used for positioning. Could also include an **audio emitter pattern** (like audio cones or distance attenuation model).

- **Script / Logic:** A component that encodes interactive logic. This can be a simple state machine, a set of event-response pairs, or a full script. To maximize compatibility, a declarative event system is preferable (like: on trigger X, set node Y's visibility to true). For more complex behaviors, one might embed a small script. We will likely support a safe subset of JavaScript or Lua, or use a visual logic format (maybe similar to Unreal Engine's Blueprints but serialized). This area will be carefully controlled for security. Script components can also call certain engine APIs (the spec will define what they're allowed to do, e.g. move objects, play animations, send AI queries).

- **AI/Agentic:** This is a novel component type for our format. It contains data for AI-driven behaviors. Examples include:

- *Agent Persona:* (For NPCs) A text prompt or knowledge base that an LLM can use, plus parameters like which model or how often to call. Possibly a placeholder for a dialogue tree or interactive narrative logic.
- *Generative Instruction:* Could be attached to an environment object, e.g. a procedural generator that says "whenever the player looks away, use a diffusion model to alter this painting's texture based on mood."
- *Memory/State:* We might include a mechanism for an AI agent's state that persists. Since a file is static, we cannot store evolving state across sessions (that would be dynamic), but we could allow initial state or default knowledge.

- *AI service reference:* For instance, a component that means "this object's texture is generated by calling XYZ API with these parameters" – the engine can do that at runtime if needed.

- **Networking/Collaboration:** (Future extension) Could mark which nodes are meant to be shared in multi-user sessions (so an engine knows to sync them via a network). The file itself is static, but including an identifier for networked objects helps if the scene is loaded in a multi-user engine – all clients see that object with the same ID and can sync changes to it.

- **Meta and Extensions:** A catch-all where custom or future components go. E.g. a **haptic feedback** component for XR haptics, an **eye-tracking target** for focusing, or a **robotics navmesh** to aid robot agents navigating the scene. These would be defined outside the core spec by domain experts, but our format's structure can include them.

It's important that each component type is optional and only parsed by engines that support it. This modular design ensures that, say, a standard WebXR viewer that only knows about geometry, material, animation can still open the file and render the scene, albeit without AI behaviors or whatever it doesn't understand. It would simply skip unknown components. Meanwhile, a specialized viewer (imagine an AI-enhanced XR app) would activate those components and provide the full experience.

Schema Example

To ground this, here's a brief example of what a node with multiple components might look like in JSON form:

```
{
   "id": "npc1",
   "name": "ShopkeeperNPC",
   "translation": [0,0,0],
   "rotation": [0,0,0,1],
   "scale": [1,1,1],
   "components": {
       "geometry": { "meshRef": "geom:npcMesh_hi", "meshLODs":
["geom:npcMesh_med","geom:npcMesh_low"] },
       "material": { "ref": "mat:npcMat" },
       "animation": { "ref": "anim:npcIdle" },
       "aiAgent": {
           "model": "OpenAI-GPT4",
           "prompt": "You are a medieval shopkeeper. Greet the player and offer
help.",
           "updateMode": "onPlayerApproach",
           "responseAction": "speak"
       },
       "audio": { "ref": "audio:shopkeeperVoice", "mode": "TTS" }
   }
}
```

In this snippet, the NPC has a geometry with multiple LODs (high, medium, low), a material, an idle animation, an AI agent component that specifies using GPT-4 with a certain prompt and that it should update (generate new output) when the player approaches, and an audio component indicating the voice lines are tied to a text-to-speech (TTS) system using an audio asset (or possibly generated on the fly).

Another example for a procedural object:

```
{
  "id": "treeField",
  "components": {
      "proceduralGenerator": {
          "type": "forest",
          "seed": 42,
          "area": { "shape": "circle", "radius": 50 },
          "density": 0.1,
          "template": "geom:oakTree"
      }
```

```
    }
  }
```

This could indicate: generate a forest of oak trees in a 50m radius circle area, using a density and random seed. The `template` refers to an oak tree model (which could itself be another node or just a resource). An engine reading this might instantiating many tree nodes at runtime. This shows how our format can embed concise instructions for potentially huge content.

Example Use Cases

To illustrate how the format can be used in practice, we describe a few representative scenarios:

Use Case 1: Volumetric Video in AR Telepresence
Imagine a holographic video call system where a person is captured as a 3D volumetric video (point cloud or mesh per frame with texture). Using our format, the call setup sends an initial `.xrf` file to all participants. This file contains a node for the remote participant with a **MeshSequence component** indicating it's a timed sequence. The resource for it is not a static mesh but an external stream labeled, say, `pointCloudStream` with a DASH manifest URL. The file also includes an **Audio stream component** for that participant's spatial audio. When loaded in an AR headset (or phone), the app reads the file, sees that it needs to fetch the pointCloudStream, and begins streaming it. The user sees the other person in 3D appearing in front of them in real-time. MPEG's point cloud compression (V-PCC) ensures that even if the point cloud is say 1 million points per frame, it's streamed efficiently at, e.g., 8 Mbit/s with decent quality [29]. The AR device uses the circular buffer approach to buffer a few frames ahead [28] to account for network jitter. If the network is good, a high-detail stream is used; if bandwidth drops, the Media Function automatically switches to a lower-detail stream (perhaps the .xrf provided two alternatives in the MPEG_media description). The key is that all this complexity is hidden behind the format's descriptors – the scene graph just knows there's a "person" node with continuously updating geometry. For devices that cannot handle the volumetric feed (e.g., a weaker phone), the file might also include a fallback: a flat video billboard or an avatar model to use instead. The AR engine picks the fallback if needed. After the call, the streamed frames aren't necessarily saved (unless for playback), keeping memory usage low. This use case shows the format enabling **live, high-performance streaming XR content** in a standardized way.

Use Case 2: AI-Augmented VR Game Level
Consider a VR adventure game where the environment and NPC behaviors are partly AI-generated. The level is exported as an .xrf file. In this scene, there is a medieval town square. Many assets are typical (buildings, props as static meshes). But, the NPCs have AI agent components: each has a unique prompt/personality in the file. There is also a **global AI director** entity that has a script or prompt describing the overall plot, so it can dynamically generate events or dialogue. When a player plays this level, their VR engine loads the file, spawns the environment, and then initializes the AI systems. Perhaps it loads a local LLM (if on a powerful PC) or connects to a cloud AI for dialogue. When the player talks to an NPC (via voice), the engine uses the NPC's prompt and conversation history (context) to query the LLM, and then generates the NPC's response (potentially doing text-to-speech on the fly). The NPC's audio component might be set to TTS mode, meaning no pre-recorded lines, but instead synthesize from text. All the while, the rest of the scene renders with good performance because the environment assets are optimized. Some environmental details might be procedural: e.g., the file could say that weather is procedural – a script that changes the sky texture and plays thunder sounds based on random but controlled timing. If the user's device doesn't

support the fancy sky shader, it could fall back to a static sky texture. The format ensures that even with all these AI and procedural pieces, the game level is encapsulated in one deliverable package. This makes it easier to share user-created levels (they don't have to ship code separately; the logic is inside the file). It also means someone could edit that .xrf (with an appropriate editor) to modify the AI prompts or swap models, enabling a new kind of modding.

Use Case 3: Multi-User Shared World with Cloud Backend
Envision a shared XR experience – e.g. a virtual art gallery that multiple users (on different devices) can visit together. The gallery space, artwork, and interactive elements are packaged in our format. When deployed, the .xrf is placed on a cloud server. Users' clients download the file to get the static content. The file contains special markers for network-synced objects, say an object that represents a collaborative drawing canvas. That object might have a component marking it as network-synced and also perhaps an **OpenBrush-style** format for strokes. As users draw in VR on that canvas, their apps send the new stroke data to a server which relays to others. The initial .xrf had the canvas with maybe some initial state or blank state. Because each stroke is encoded (like OpenBrush uses a list of control points for each stroke [32]), the network can broadcast those small datasets and each client integrates it. The format ensures consistency by using the same procedural stroke definition for all – so everyone sees the same smooth curve drawn. For things like user avatars, the format might not handle live position (that's purely network), but the avatar models themselves could be specified in the file (so everyone has the same set of avatar options defined). If the gallery uses generative art – say each artwork is an AI-generated painting based on the day's news – the .xrf can include the generator descriptors (model type, prompt template). A cloud service might periodically update a texture via that descriptor. All clients either fetch the updated image or generate it locally if they have the model. Thanks to the standardization, a new client that joins knows exactly how to do that because it's in the spec (e.g. "if you see a GenerativeTexture component with model=StableDiffusion, prompt=X, you either have an SD model to run or you call a service").

Use Case 4: Robotics and IoT Interaction (future-looking)
As XR and AI blend, consider a household robot that can also project AR or is aware of AR content. Our format could be used to share a spatial map between a robot and an AR headset. For example, a cleaning robot scans your room and builds a 3D map (mesh or point cloud). It then creates an .xrf file of your room's layout – including semantic labels (like it recognizes "couch", "table"). Your AR glasses load this file to use as a mesh for occlusions and physics (so virtual objects can collide with your real furniture in AR). The file's semantic metadata also allows your glasses' AI assistant to reason about the space ("The keys are on the table" – and the system knows which object is the table). If the robot updates the map (furniture moved), it could stream an update to the .xrf's content – since our format can allow dynamic updates or at least be reloaded. In another scenario, a neural interface (BCI) might feed user intent to the system; while not directly a file concern, the format could include placeholders for "user attention hotspot" or similar, which a BCI could fill. For instance, an object could have an attribute "reactWhenLookedAt" which, combined with eye-tracking or neural signals, triggers an animation. This shows how the format is extensible to novel inputs and outputs as technology evolves.

Implementation and Integration

To realize this format, we propose establishing an **open-source project and working group**. The initial spec will be drafted (likely influenced by this document), and prototype implementations will be created. Key deliverables include:

- **Specification Document:** A formal specification defining the file format syntax (likely in JSON schema or similar), binary structures, and module definitions. This will be versioned (v1.0, etc.) and published openly (perhaps via Khronos or the Metaverse Standards Forum).

- **Reference SDK/Libraries:** We will provide C++ and JavaScript reference libraries for loading and writing .xrf files. Possibly C# for Unity as well. These libraries handle parsing, basic validation, and provide the data in a structured object model for the engine to use. Where needed, they might include decoder implementations (e.g. a built-in Draco decompressor, or an adapter to call existing decoders). For AI integration, since not every platform can include an AI model, the library might interface with a plugin system (engines can register how to handle AIAgent components, e.g. by hooking them to their own AI modules).

- **Tool Plugins:** Exporters for Blender and Maya (to go from creation to .xrf). A Unity exporter that can save a scene or prefab as .xrf (with certain limitations perhaps). Also converters: e.g. glTF to XRF for static models, USD to XRF for scene graphs (we can leverage USD's Python API to read a scene and output .xrf).

- **Validator and Compressor Tools:** Similar to glTF's `gltf-validator`, a tool to check .xrf files for spec compliance will be needed. Additionally, since we allow multiple ways to encode content, tools to **optimize a file** will be useful – e.g. a tool that takes an .xrf and compresses all its meshes with Draco if not already, or downscales textures for a mobile-friendly version. The format might support profiles or levels (like "Level 1 XR file" means no more than X MB, etc., for certain platforms).

- **Viewer/Player:** A minimal viewer application (perhaps web-based) that can load an .xrf and render it with basic interactions. This would help showcase the format and test compatibility. Because the format covers so much, a full implementation in one viewer is non-trivial, but initially focusing on core features (static scene rendering, simple AI text output) is enough, then build up.

- **Community and Extension Process:** We'll set up a repository for extensions, where proposals for new modules (like new AI components or new physics types) can be discussed. This will ensure the format stays **future-compatible**. As new AI breakthroughs occur (e.g. say 2026 brings a new kind of 5D video format), the format can add support without breaking older files (thanks to extension IDs and backwards compatibility rules).

Compatibility with glTF and USD: To ease transition, certain parts of our format could be directly compatible or at least mappable. We might declare that a subset of .xrf (with only static meshes, materials, and animations) is effectively glTF 2.0-compliant. Perhaps we can even structure the JSON such that it's easy to generate a glTF from it by picking relevant fields. Similarly, since USD is used by content creators, one might use USD as an intermediate or import .xrf into USD pipelines for further editing (ignore the AI parts,

but preserve geometry and layout). This interoperability is important so that creators aren't locked into one ecosystem.

Performance Testing: As part of spec development, we'll conduct tests on target devices (phone, AR headset, PC, etc.) to ensure the format's design choices meet performance goals. For instance, we'll test loading a large scene with streaming to verify that the chunking and buffer management works without stutter. We'll test AI calls, making sure that if an AI service is slow or offline, the engine can degrade gracefully (maybe using fallback behavior from the file).

Governance: Likely, we propose the format be managed under an organization like the **Khronos Group** (which has experience with glTF, OpenXR) or the **Metaverse Standards Forum** (which is a broader coalition). This helps with industry adoption. Involvement from companies working on XR, game engines, AI, and web would be sought so the format addresses real needs. An open conformance suite would be developed to certify viewers (just as there are conformance tests for glTF).

Finally, we foresee this format becoming a **cornerstone for spatial computing content** as the industry moves towards more interactive, AI-driven experiences. By being open and extensible, it can adapt over the coming years, handling the growth of XR from today's early use cases to tomorrow's ubiquitous spatial web. Our emphasis on performance, modularity, and AI integration positions it as a forward-looking standard for the convergence of **AI, XR, and spatial media** – truly enabling the next generation of immersive, intelligent environments.

---

1 2 6 11 12 13 22 23 24 25 26 27 28 MPEG Extends and Integrates glTF 2.0 into MPEG-I Scene Description ISO/IEC 23090 - Khronos Blog - The Khronos Group Inc
https://www.khronos.org/blog/mpeg-extends-and-integrates-gltf-2.0-into-mpeg-i-scene-description-iso-iec-23090

3 4 33 Generative XR: How Will AI and Spatial Computing Collide? - AR Insider
https://arinsider.co/2024/08/13/generative-xr-how-will-ai-and-spatial-computing-collide/

5 8 9 10 14 15 16 17 18 19 20 30 glTF or OpenUSD - Which is the best 3D file format to use?
https://www.linkedin.com/pulse/gltf-openusd-which-best-3d-file-format-use-jed-fisher-s24ce

7 GitHub - graphdeco-inria/gaussian-splatting: Original reference implementation of "3D Gaussian Splatting for Real-Time Radiance Field Rendering"
https://github.com/graphdeco-inria/gaussian-splatting

21 OpenUSD instead of GLTF: Notes of the USD Roundtable at GDC ...
https://rystorm.com/blog/usd-roundtable-gdc-notes-2024

29 Point Cloud Compression in MPEG - ICIP 2020
https://2020.ieeeicip.org/program/tutorials/point-cloud-compression-in-mpeg/index.html

31 32 Open Brush File Format | Open Brush Docs
https://docs.openbrush.app/developer-notes/open-brush-file-format

34 [PDF] MPEG Point Cloud Compression First Standard for Immersive Media
https://confit.atlas.jp/guide/event-img/idw2019/DES5-3/public/pdf_archive?type=in

35 36 SMERF
https://smerf-3d.github.io/