



# Technology under Consideration for ISO/IEC 23090-14

WG3 Scene Description BoG

MDS25880\_WG03\_N01691

# Table of Contents

1. Extensions .....	1
1.1. Extension of relationship information between MPEG SD nodes .....	1
1.1.1. General .....	1
1.1.2. Motivation .....	1
1.1.3. Proposal .....	1
1.1.4. Proposed extension syntax .....	2
1.1.5. Applications .....	3
1.1.6. Conclusion .....	4
1.1.7. References .....	4
2. Codec Support .....	5
2.1. Dynamic mesh support in scene description .....	5
2.2. On the support of immersive codec application in MPEG-SD .....	5
2.2.1. General .....	5
2.2.2. Background .....	5
2.2.3. Motivation .....	6
2.2.4. Proposal .....	6
3. Interfaces .....	10
3.1. Supporting Multiple Viewers in the Media Access Function .....	10
3.1.1. General .....	10
3.1.2. Proposed Updates to MAF API .....	10
4. Interactivity framework .....	12
4.1. Mapping interactivity to Khronos extension .....	12
4.1.1. Introduction .....	12
4.1.2. Overview of both Interactivity Frameworks .....	12
5. File Format .....	25
5.1. Revising carriage format for glTF in scene description .....	25
5.1.1. General .....	25
5.1.2. Issue .....	25
5.1.3. Proposal .....	25
Appendix A: Disclaimer .....	27



### 1.1.3.1. Option 1: Extension on MPEG\_scene\_dynamic

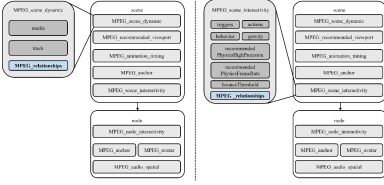


Figure 2. In case of inserting additional extension regarding Relation information, which is named MPEG\_relationships for MPEG\_scene\_dynamic and MPEG\_scene\_interactivity.

MPEG Scene Description supports dynamic interactivity and media integration through extensions such as MPEG\_scene\_dynamic and MPEG\_scene\_interactivity. The MPEG\_scene\_dynamic extension defines how objects within a scene can change dynamically, incorporating concepts like events, states, and actions to describe how a node's attributes transform under specific conditions. Similarly, the MPEG\_scene\_interactivity extension enables dynamic interactions in response to various triggers, such as user input, proximity, collisions, or visibility changes. These triggers can then execute a wide range of actions, including animations, transformations, material changes, and media processing.

To enhance these existing extensions, we propose an extension that explicitly incorporates inter-node relationship information within either the MPEG\_scene\_dynamic or MPEG\_scene\_interactivity extensions as shown in Figure 2. By introducing a new relations property as an attribute for each extension, we can clearly and explicitly define the relationships between nodes that are relevant to dynamic interactions, as shown in the accompanying figure. This extension allows for a more structured and context-aware approach to scene manipulation.

### 1.1.3.2. Option 2: Scene relationship representation using extension

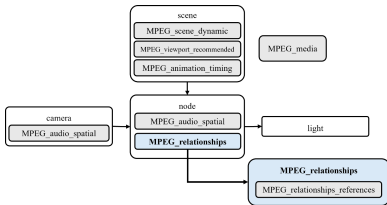


Figure 3. A MPEG-I SD scene graph showing the proposed MPEG\_relationships and MPEG\_relationships\_references extensions as external layers, independent of the main node hierarchy.

Another approach is to maintain the existing MPEG-I SD structure and describe object relations solely through extensions, such as MPEG\_relationships and MPEG\_relationships\_references. As shown in Figure 3, these extensions function as a layer external to the main node hierarchy, providing a flexible way to encode a variety of semantic and spatial links between objects.

This option is particularly advantageous as it avoids modifying the base MPEG-I SD specification, ensuring backward compatibility and a non-disruptive integration into the current ecosystem. By encapsulating relationship data within these extensions, MPEG-I SD can flexibly support advanced scene descriptions, including those with timed relations, without imposing changes on core elements.

### 1.1.4. Proposed extension syntax

This section describes the proposed syntax for the MPEG\_Relationships extension, which is

designed to define semantic relationships between nodes. The proposed syntax is detailed in Table 1.

Syntax	No. of bits	Mnemonic
MPEG_Relationships () {\		
Subject_id	16	uimsbf
Object_id	16	uimsbf
Relation_type_id	8	uimsbf
Relation_type	8	uimsbf
}		

Table 1. Proposed syntax of the MPEG\_Relationships extension.

This structure allows for a clear and efficient representation of the relationships between different objects within the scene.

### 1.1.5. Applications

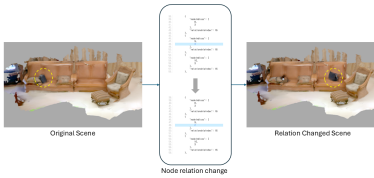


Figure 4. Application of scene reconstruction using scene relation information

Applying the proposed relationship information extension would enable its use in a variety of scenarios. First, it would allow for relationship-based 3D environment reconstruction. In this scenario, when a user manipulates an object in a 3D environment, the proposed relationship information is used to automatically adjust the position and state of related objects, thereby creating a dynamic scene.

For example, as shown in Figure 4, if a user moves a cushion on a sofa, the system will understand the "on-sofa" relationship with the sofa and the "adjacent" relationship with other cushions. It would then automatically adjust their positions to avoid collisions and maintain appropriate distances. This method allows for a more structured and context-aware approach to scene manipulation.

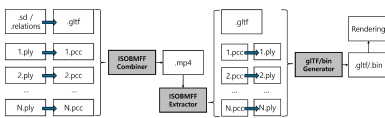


Figure 5. Application of ISOBMFF combiner with relationship information

Additionally, this approach can be extended for use in ISOBMFF, a general-purpose container format for storing digital media files. For example, as shown in Figure 5, ISOBMFF records the track locations of compressed PCCs in the scene's glTF, and before rendering, it generates glTF/bin by utilizing the track location information. At this stage, if relationship information is included through a glTF extension, the ISOBMFF extractor and glTF/bin generator can leverage this relationship information to not only restore individual objects but also reconstruct the scene while

preserving inter-object spatial relationships and interaction conditions. As a result, the framework can be extended to support context-aware scene reconstruction and efficient media synchronization.

### **1.1.6. Conclusion**

In this contribution, we analyzed the limitations of the current MPEG-I SD specification in representing semantic relationships between objects and proposed a relationship extension to address these gaps. Two possible approaches were discussed: Option 1, which integrates relationship information into existing extensions such as `MPEG_scene_dynamic` or `MPEG_scene_interactivity`, and Option 2, where relationships are represented through new external extensions (`MPEG_relationships`, `MPEG_relationships_references`) that preserve the integrity of the core MPEG-I SD structure.

Both approaches demonstrate the feasibility of systematically managing static and dynamic relations within 3D scenes. Based on the use cases presented, including scene reconstruction and ISO/BMFF-based integration, the proposed extensions can enhance MPEG Scene Description by enabling context-aware scene manipulation and consistent inter-object semantics. We therefore recommend that the MPEG SD group consider adopting a relationship extension framework for MPEG-I SD and further evaluate the two options through test assets and experimental validation. This will help determine the most suitable integration path for ensuring interoperability, backward compatibility, and support for advanced interactive media applications.

### **1.1.7. References**

- [1] Moon, H., Kim, M., Jeong, J., Kim, S., Park, S., [SD] Consideration of semantic representation in MPEG-I SD,” ISO/IEC JTC1/SC29/WG3, m64402, Jul. 2023.
- [2] Hong, J., Lee, D, Song, C., Park, S., Moon, H., Kim, M., Jeong, J., & Kim, S., “[SD] Report on possible asset for scene description,” ISO/IEC JTC1/SC29/WG3, m64815, Oct. 2023.
- [3] Lee, D, Jeong, J., Kim, S. & Park, S., “[SD] A Need of Scene Description Extension to Represent Relationship Information,” ISO/IEC JTC1/SC29/WG3, m64098, Jul. 2024.

## Chapter 2. Codec Support

## 2.1. Dynamic mesh support in scene description

V-DMC is considered for future Amendment

## 2.2. On the support of immersive codec application in MPEG-SD

Source: [m73419](#)

### 2.2.1. General

The proposal focuses on the support of immersive codec applications in MPEG-I Scene Description, ISO/IEC 23090-14.

An immersive codec application refers to an application with video content designed to engage the viewer in a more interactive or enveloping experience, often by simulating a sense of presence within the scene.

### 2.2.2. Background

The MPEG-I Scene Description group has adopted glTF as a scene graph format for ISO/IEC 23090-14. The glTF format has been further enhanced to support time-based content, including 2D and dynamic volumetric media by introducing the `MPEG_texture_video` extension to the texture property, as shown in [Figure 6](#). The `MPEG_texture_video` extension provides information such as the resolution of the texture, sample format information, and buffer information. ISO/IEC 23090-14 also describes a sampler extension to sample textures in the commonly used YCbCr format for video applications.

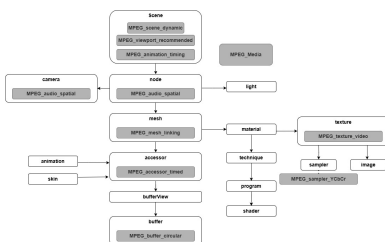


Figure 6. *glTF 2.0 object hierarchy with MPEG extension vendor.*

Below are examples of applications enabled by immersive codecs:

- “multiview”: Multiview video delivers multiple synchronized views of the same scene from different viewpoints, enabling flexible rendering for applications.
- “stereo”: Stereo video delivers a sense of depth by presenting two slightly different views (one for each eye).
- “3D”: 3D video delivers a perception of depth by combining visual views, such as stereoscopic views and depth maps.

- “surround”: Surround video (e.g. 360 videos) delivers an immersive experience by capturing the entire surrounding environment in all directions.

### 2.2.3. Motivation

Immersive codec applications are commonly used in scenarios where synchronized video streams (color maps, depth maps, etc.) captured from multiple cameras are needed to reconstruct or navigate a scene from different angles. The reconstruction process involves combining these textures to estimate the 3D representation of the scene.

The different video streams are typically carried within a bitstream with the use of NAL unit layers. The bitstream is composed of a base layer (e.g. low resolution, base texture) and enhancement or non-base layers (higher resolution, improved quality, or carry additional information such as depth or alpha data).

As MPEG\_texture\_video refers to one texture, it may not be sufficient to express the combination of such immersive textures for immersive codec applications.

On the other hand, glTF uses “material” property which uses a set of properties to define material representations such as Physically Based Rendering (PBR). This is achieved using a declarative representation of materials which are to be rendered consistently across different platforms. Each property in a material can be defined using textures. A rich representation of a model can be achieved by the combination of the different textures for each property.

The glTF2.0 ecosystem can benefit from the new immersive video formats such as 3D-HEVC, MV-HEVC, etc. Such immersive video formats describe different components to express artistic intent which enable rich experiences for the user. The different components of immersive video formats can be declared in a glTF2.0 document with semantic meaning and associate texture information for the final rendering. In addition to the texture information, supplementary information such as camera calibration, stereo information, projection type and more can be described to accurately perform the final rendering of the immersive media.

To maintain the flexibility of texture combinations and ensure consistent visual fidelity across immersive experiences, a material extension could be enriched to support immersive representations.

### 2.2.4. Proposal

To integrate the support of immersive codec application in MPEG Scene Description, the proposal is to add an extension to the “material” in glTF 2.0, named “MPEG\_Material”, as shown in [Figure 7](#).

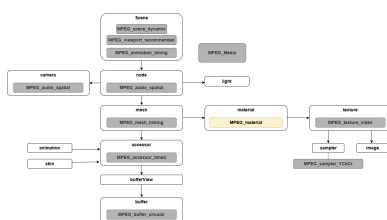


Figure 7. MPEG\_material extension definition

Each mesh primitive has a reference to a material that defines how the surface of a 3D object looks

when rendered. The MPEG\_material extension is applied to the “material” property and signals one immersive codec application among a set of supported immersive applications. The “mesh” defines the geometry of the object, whether it is a 2D plane, a sphere or a 3D mesh object, while the MPEG\_material extension aggregates all relevant information to a specific immersive media content within the same container, facilitating the scene reconstruction process. The definition of the MPEG\_material extension is shown in Table 1.

Table 1 - Definition of MPEG\_material extension.

Name	Type	Default	Usage	Description
codecApplication	string	N/A	M	Provides an information on the immersive codec application. Possible values include: "multiview", "stereo", "3D", and "surround".
layers	array[Layer]	N/A	M	Provides an array of layer objects for “multiview”, “stereo”, “3D”, and “surround” immersive codec applications
<b>Legend:</b>  For attributes: M=mandatory, O=optional, OD=optional with default value, CM=conditionally mandatory.				

In Table 1, a “layer” is an object which combines a texture with its associated rendering properties. Below are some examples of how layers can be used in the context of an immersive codec application:

- In the case of “multiview” video, each view is represented by a separate layer, identified by a unique nal\_layer\_id within the NAL units; typically, a base layer provides a primary view, while non-base (enhancement) layers deliver additional views.

- In the case of “3D” video, layers are used to separate views and depth information, where a base layer provides the primary view and non-base (enhancement) layers add stereoscopic or multiview data.
- In “surround” video, layers are used to separate the full-scene base view and optional stereo components.

The definition of Layer is described in Table 2.

Table 2 - Definition of Layers in the MPEG\_material extension.

Name	Type	Default	Usage	Description
primaryReference	boolean	N/A	M	Indicates whether this layer component is a primary component among other layers. For example, it can be used to refer to base (True) layer or non-base (False) layer. In case of stereo, it refers to the hero (True) or non-hero (False) layer.
texture	integer	N/A	M	Provides index of a texture.
cameraNode	integer	N/A	M	Provides index to a node containing a camera property used for reprojection. The camera property describes the internal characteristics of the capture camera (intrinsics). A camera node is a floating node with transform information (extrinsics) of the capture camera. It shall be only used for reprojection in the context of immersive codec applications.
eyeChannel	integer	N/A	O	Indicates if the information refers to left (0) or right (1) eye used in the case of stereoscopic application.

Name	Type	Default	Usage	Description
projectionType	integer	N/A	O	Indicates the type of projection used for deprojection in the case of surround video. It can be equirectangular(0), cubemap(1), pyramidal(2), parametric(3).
<p><b>Legend:</b></p> <p>For attributes:  M=mandatory,  O=optional,  OD=optional with default value,  CM=conditionally mandatory.</p>				

# Chapter 3. Interfaces

## 3.1. Supporting Multiple Viewers in the Media Access Function

Source: [m58510](#)

### 3.1.1. General

In the Presentation Engine of the MPEG-I Scene Description architecture, the viewer's view of the scene is determined by the camera used for rendering the scene from the viewer's viewpoint. In many use cases, the Presentation Engine runs on the end user's device and therefore there is only one viewer for the scene and one camera object is used at any given point in time for composition and rendering. Using the camera information provided by the Presentation Engine, the MAF can identify which objects in the scene are within the viewing frustum of the camera at a given time instance.

However, in some scenarios multiple cameras are used for rendering the scene from a number of viewpoints corresponding to different viewers of the same scene (e.g., in multi-viewer applications such as online conferencing applications with multiple users). In such scenarios, information about the cameras used to generate each viewer's view of the scene, including both intrinsic and extrinsic camera parameters, are required by the MAF to identify and request the appropriate media or media parts for each viewer.

Since a media pipeline is tightly coupled with the type of the media, it may not be desirable to have multiple media pipelines for the same content for different viewers. Rather, the MAF should allow a single media pipeline for a media content to be used for composition and rendering for different viewers.

### 3.1.2. Proposed Updates to MAF API

To support media fetching for multi-viewer applications, where each viewer may have their own extrinsic and intrinsic camera parameters, relevant methods in the MAF API and their definition should be updated as follows (updates are in **bold**).

#### 3.1.2.1. Methods

*Table 1. n/a*

Methods	State after success	Description
startFetching()	ACTIVE	<p>Once initialized and in READY state, the Presentation Engine may request the media pipeline to start fetching the requested data.</p> <p><b>The initialization may be performed using view information for one or more viewers.</b></p>
updateView()	ACTIVE	<p>Update the current view information. This function is called by the Presentation Engine to update the current view information, if the pose or object position have changed significantly enough to impact media access. It is not expected that every pose change will result in a call to this function.</p> <p><b>A call to this function shall include the view information for only those views whose parameters have significantly changed.</b></p>

### 3.1.2.2. IDL for media pipeline

```

interface Pipeline {
    readonly attribute Buffer          buffers[];
    readonly attribute PipelineState  state;
    attribute          EventHandler  onstatechange;
    void    initialize.  (MediaInfo mediaInfo, BufferInfo bufferInfo[]);
    void    startFetching (TimeInfo timeInfo, ViewInfo viewInfo[]);
    void    updateView.   (ViewInfo viewInfo[]);
    void    stopFetching. ();
    void    destroy.      ();
};

```

# Chapter 4. Interactivity framework

## 4.1. Mapping interactivity to Khronos extension

Source: [m72365](#)

### 4.1.1. Introduction

In this contribution, we present a detailed mapping between the MPEG\_interactivity extension (defined in ISO/IEC 23090-14) and the KHR\_interactivity extension (the Khronos Group’s glTF 2.0 interactivity extension). We begin with an overview of each extension, including their structure and purpose. Next, we define a one-to-one mapping of core interactivity constructs – such as triggers, actions, and variables in MPEG\_interactivity – to their equivalents in KHR\_interactivity (event nodes, flow/logic nodes, variables, operations, etc.). We then discuss the processing rules for each system, explaining how events are triggered, conditions evaluated, and actions executed in MPEG’s model versus Khronos’s behavior graph model. Finally, we illustrate the mapping with a fully detailed example scenario: a user interacts with a “remote control” object in a 3D scene to turn on a TV and start video playback. The scenario is described narratively and shown in both MPEG\_interactivity JSON syntax and KHR\_interactivity JSON (behavior graph) syntax, accompanied by diagrams of the event flow. Throughout the report, we use clear section headings and concise explanations for readability.

### 4.1.2. Overview of both Interactivity Frameworks

#### 4.1.2.1. 2.1 Overview of MPEG\_interactivity

The MPEG\_interactivity extension is part of the MPEG-I Scene Description standard and is designed to integrate interactive behavior into 3D scenes (built on glTF 2.0). Interactivity in MPEG-I is defined at two levels: a scene-level extension (MPEG\_scene\_interactivity) and a node-level extension (MPEG\_node\_interactivity). The scene-level extension contains the global definitions of triggers, actions, and behaviors that apply to the scene. The node-level extension can complement these by providing additional data or specialized parameters for specific nodes (for example, physics or collision properties for that node). In essence, MPEG\_interactivity allows authors to define interactive behaviors that link user or environment events to changes in the scene, all within the MPEG scene description.

**Triggers and Actions:** In MPEG’s model, a behavior is the fundamental unit of interactivity, defined as a pairing of one or more triggers with one or more actions. Triggers represent events or conditions that can occur – for example, a collision between objects, an object coming within a certain distance of another (proximity), a user input event, or an object’s visibility state. These triggers can originate from user interactions (e.g. controller input or gestures), temporal/spatial conditions, or system events. Actions represent the responses that occur when triggers are activated – for example, enabling or disabling an object, transforming or moving an object, playing an animation or media, changing a material, applying haptic feedback, etc. MPEG\_interactivity defines a fixed set of trigger types and action types. Below is a summary of the main types:

- Trigger types:

- *TRIGGER\_COLLISION*: Fires when a collision is detected between specified nodes.
- *TRIGGER\_PROXIMITY*: Fires when one node comes within a defined distance of another.
- *TRIGGER\_USER\_INPUT*: Fires on a user input or gesture (e.g. a button press or hand motion).
- *TRIGGER\_VISIBILITY*: Fires when a node becomes visible or hidden (e.g. entering/exiting the camera frustum).
- Action types:
  - *ACTION\_ACTIVATE*: Enable or disable a node's active state in the scene.
  - *ACTION\_TRANSFORM*: Apply a specified transformation to target node(s).
  - *ACTION\_BLOCK*: Lock or freeze a node's transform.
  - *ACTION\_ANIMATION*: Control an animation, e.g. play, pause, resume, stop a glTF animation by index.
  - *ACTION\_MEDIA*: Control a media asset, e.g. video or audio playback of media defined in an MPEG\_media extension).
  - *ACTION\_MANIPULATE*: Initiate a user manipulation of a node, e.g. grabbing an object with a controller.
  - *ACTION\_SET\_MATERIAL*: Swap the material of target node(s), e.g. to change an object's appearance.
  - *ACTION\_HAPTIC*: Trigger haptic feedback via haptic devices, with parameters for the type of feedback.
  - *ACTION\_SET\_AVATAR*: Invoke an avatar-specific action.

A behavior in MPEG\_interactivity ties together one or more triggers with one or more actions, along with some logic controls. The behavior definition can specify a logical combination of triggers (using AND, OR, NOT operators) that must be satisfied to activate the behavior. For example, a behavior might be set to trigger only when Trigger A AND Trigger B are true, or Trigger C OR Trigger D is true, etc. This logical expression is given as a string (e.g. "#1 & ~~#2 | (#3 & #4)" where numbers refer to trigger indices). The behavior also defines a triggersActivationControl, which indicates when the trigger condition is considered activated. This allows behaviors to fire one-time or repeatedly, and to detect both the “enter” and “exit” of a condition (e.g. when a user enters a zone vs leaves it). Additionally, a behavior can specify whether multiple actions execute sequentially or in parallel, an optional interruptAction (an action to execute if an ongoing behavior is interrupted or removed), and a priority value to arbitrate if multiple behaviors conflict.

In summary, MPEG's interactivity model is a rule-based system, where behaviors are essentially rules of the form “if these trigger conditions are satisfied, then perform these actions.”

#### 4.1.2.2. Overview of Khronos Interactivity

The Khronos KHR\_interactivity extension for glTF 2.0 introduces a behavior graph system to incorporate interactivity into glTF assets. Instead of pre-defined trigger-action lists, KHR\_interactivity uses a node graph (behavior graph) model similar to Unreal Engine's Blueprints or Unity's visual scripting. The extension allows content creators to define logic within the glTF file itself, so that the asset can respond to events in a consistent way across different viewers. The

primary motivation is to make interactive 3D assets portable and self-contained. The focus is on safety and sandboxing; by using a limited set of graph nodes and not arbitrary scripting, the behaviors remain predictable and secure across platforms.

A KHR\_interactivity behavior graph is essentially a directed acyclic graph (DAG) of nodes connected by links (edges) that pass execution flow or data. Each node performs a simple function (such as detecting an event, evaluating a condition, or performing an action), and nodes are connected such that an event triggers a flow through the graph, causing actions to happen. Nodes in the graph fall into one of several categories:

- **Event Nodes:** these are entry points that listen for certain events and start the execution flow when the event occurs. Examples of events include *lifecycle events* (like “On Start” when the scene/asset loads, or “On Tick” each frame) and *custom or user-defined events*. Custom events are essentially named events that can be triggered by external application.
- **Action/Operation Nodes:** Nodes that perform an action or change in the scene. These correspond to things like starting an animation, changing a material or variant, transforming a node, playing a sound or video, etc. They typically consume input values and produce an effect. Often, these nodes will interface with other glTF extensions or properties (e.g., a node to play an animation will reference a glTF animation, etc.).
- **Logic/Flow Control Nodes:** Nodes that direct the flow of execution or make decisions. Examples include *Branch* (an if-else node that routes the flow based on a boolean condition), loops, and sequence or delay nodes to chain or defer actions. These nodes don’t directly affect the scene; they control which actions happen and when.
- **Variable/State Nodes:** Nodes that store or manipulate state. The extension introduces the concept of variables that can hold values during the execution of the graph. Variable nodes might include setting a variable, reading a variable, or modifying it (e.g., incrementing a counter). These are useful for keeping track of state across events. There are also query nodes which can retrieve information from the glTF scene or runtime.
- **Math/Conversion Nodes:** Basic arithmetic or logic comparison nodes (e.g., add, subtract, boolean AND/OR, compare values) and type converters. These help build conditions and compute values to use in decisions or actions.

All nodes have defined input sockets and output sockets. There are typically two kinds of connections: flow connections that pass along execution order and data connections that pass values. The use of flow sockets means the graph’s execution is explicit: an event node emits a flow, which travels through connected nodes in sequence. This is how behaviors are executed. In other words, when an event occurs, it triggers the next node via a flow link, and so on, forming a chain of execution.

KHR\_interactivity by itself defines the graph framework (nodes, events, variables), but it works in concert with other glTF mechanisms to actually affect the 3D scene. For example, to change the state of an object, an operation node might use the KHR\_animation\_pointer extension to target a specific property of a glTF node (like its translation, or a custom “visibility” flag). Likewise, an action node that plays a video might rely on a video texture extension or media extension to handle the media resource. This modular approach means KHR\_interactivity can be extended by introducing new node types via additional extensions. The initial set of node types covers common needs (UI events, animation control, variant switching, simple logic, etc.), and more specialized

interactions (like physics or complex UI) may involve additional glTF extensions.

#### 4.1.2.3. Mapping MPEG\_node/scene\_interactivity to KHR\_interactivity

Despite differences in approach (rule-based vs node-graph), MPEG\_interactivity and KHR\_interactivity address similar needs and concepts. Below, we map the key constructs one-to-one between MPEG's design and Khronos's design:

1. Triggers (MPEG) ⇨ Event Nodes (KHR): A trigger in MPEG corresponds to an event that can start a behavior. In KHR\_interactivity this is represented by an Event node in the behavior graph. For example:
  - *MPEG TRIGGER\_USER\_INPUT* maps to a Custom Event node or a specific input event node in KHR. Khronos does not hard-code user input events in the initial spec; instead, one would use a *Custom Event* node that is fired by the viewer when the user performs that input. In practice, this means an MPEG user input trigger like "left controller trigger pulled" would translate to a custom event named "left\_trigger\_pull" in the glTF behavior graph, which the viewer knows to send on that input.
  - *MPEG TRIGGER\_COLLISION* has no direct built-in equivalent in the base KHR\_interactivity since physics/collision aren't covered in the initial draft. The analogous concept would be an event triggered by a physics engine. This could be achieved with an extension: for instance, a KHR\_physics extension might generate a custom event when a collision occurs.
  - *MPEG TRIGGER\_PROXIMITY* similarly maps to a custom event from an engine when an object enters/exits a zone. In essence, a proximity trigger can be implemented in KHR\_interactivity using the building blocks of event + logic nodes, since there isn't a single "OnProximity" node by default.
  - *MPEG TRIGGER\_VISIBILITY* would map to an event or condition related to the camera view. Khronos could handle this by using an On Tick event and a query node to check if the object is in the camera frustum, combined with a branch node. Alternatively, a custom event could be fired by the viewer when an object enters/exits view. This again shows that MPEG provides a specific trigger type, whereas KHR\_interactivity might rely on general mechanisms or future extensions to achieve the same.
  - *Lifecycle triggers*: Although not explicitly named "triggers" in MPEG\_interactivity, one can consider the scene start as an implicit trigger. In KHR\_interactivity, there is a built-in OnStart event node, which fires when the asset is loaded, and an OnTick which have no direct MPEG analog, since MPEG's triggers are more content-centric.
2. Actions (MPEG) ⇨ Action/Operation Nodes (KHR): An MPEG action corresponds to a node in the behavior graph that performs the equivalent operation. Many MPEG action types have clear counterparts or ways to achieve them in KHR\_interactivity:
  - *ACTION\_ACTIVATE*: This maps to toggling a node's active state or visibility. glTF core doesn't have an "enabled" flag, but a parallel extension (KHR\_node\_visibility) is in development to allow hiding/showing nodes. In a KHR\_interactivity graph, one could use an operation node that sets a node's visibility to true or false via the KHR\_node\_visibility property.
  - *ACTION\_TRANSFORM*: This corresponds to directly manipulating a node's translation/rotation/scale. In KHR\_interactivity, this would likely be done via an Animation or Pointer node, e.g. using KHR\_animation\_pointer to target a node's transform and setting

it. The graph might use a node that takes a matrix or vector value and applies it to the target node.

- *ACTION\_BLOCK*: In MPEG, this prevents a node from moving. There isn't a direct single node in KHR\_interactivity for this, but it could be interpreted as setting certain physics or interaction constraints. If we consider a physics extension, an equivalent would be to change the body to static or disable user interaction on that node.
- *ACTION\_ANIMATION*: This maps well to KHR\_interactivity's abilities. A likely implementation is an Animation Control node that can play/pause/stop a glTF animation.
- *ACTION\_MEDIA*: Khronos is working on incorporating media (video & audio) into glTF. Using an extension like MPEG\_texture\_video and MPEG\_media for video textures, the behavior graph would control media similarly to animations. A KHR action node for media might take a media/texture ID and a play/pause command.
- *ACTION\_MANIPULATE*: This is quite interactive and likely relies on continuous input. In KHR\_interactivity, continuous interactions, like dragging an object with the mouse or a controller, might be handled outside the behavior graph logic or by a combination of event + continuous update.
- *ACTION\_SET\_MATERIAL*: glTF has a dedicated system for material variants (KHR\_materials\_variants). In a KHR\_interactivity graph, changing a material can be done by setting the active variant of an object. For instance, there could be an Action node that sets a variant index on an object.
- *ACTION\_HAPTIC*: Currently, glTF has no haptic feedback features in standard extensions. This is a specialized case where MPEG defines parameters for haptic devices. KHR\_interactivity doesn't cover this yet; a future extension or external API would be needed. One could envision a custom event node that communicates with a haptic device when triggered. Thus, an MPEG haptic action would correspond to triggering some external haptic system in the Presentation Engine.
- *ACTION\_SET\_AVATAR*: Similarly, glTF doesn't have built-in avatar systems. MPEG's avatar actions, like toggling an avatar's microphone or performing an animation on an avatar, would require an external avatar system..

In general, MPEG's action types map onto either specific KHR\_interactivity nodes or combinations of nodes. Khronos's design tends to break down effects into simpler pieces, whereas MPEG sometimes has a single action that encapsulates a multi-step process, like manipulate or haptic, which involve continuous feedback or external devices. The simpler actions, like activate, transform, play animation, set material, have clear counterparts in the graph model.

- Behavior (MPEG) □ Behavior Graph / Event Flow (KHR): An MPEG behavior object as a whole corresponds to an event flow in the KHR\_interactivity graph. In other words, a single MPEG behavior can be thought of as a little program "if [trigger conditions] then [do actions]." In a node graph, this would be represented by wiring the event nodes for those triggers through logic nodes into the sequence of action nodes. For a simple behavior with one trigger and one action, the mapping is straightforward: one Event node connected to one Action node. For a more complex behavior, i.e. multiple triggers and multiple actions with logic, the contents of the MPEG behavior need to be expanded into multiple graph nodes, e.g., several Event nodes feeding into a Logic/AND node to replicate a combined condition, then that logic node feeding

into multiple Action nodes. The `triggersCombinationControl` string in MPEG is effectively replaced by a network of boolean logic nodes in `KHR_interactivity`, AND, OR, NOT nodes linking the outputs of event nodes, which then feed into a branch. Likewise, the `triggersActivationControl` modes, such as `FIRST_ENTER`, `EACH_ENTER`, `ON`, etc., do not exist explicitly in KHR's design. Instead, achieving the equivalent behavior relies on how the graph is constructed:

- “`FIRST_ENTER`” could be mimicked by using a variable as a flag to remember it fired, or by using an Event node that only triggers once.
- “`EACH_ENTER`” is effectively how event nodes naturally behave if you use instantaneous events. A combination of conditions would need edge detection logic, which could be done with variables.
- “`ON`” could be achieved by using a continuous event like `OnTick` and inside it, use an IF (Branch) to continuously do something while a condition holds..
- “`FIRST_EXIT/EACH_EXIT`” similarly would require tracking state and using logic in the graph, like using a combination of `OnTick` + a stored boolean to detect the transition.
- **Variables and State:** `MPEG_interactivity` doesn't define general-purpose variables for behaviors; it relies on triggers and some internal state like whether an action is ongoing. Conditions are directly encoded as triggers or combinations thereof, rather than allowing arbitrary state checks. In contrast, `KHR_interactivity` provides variables as a first-class concept. This means some logic that would require a custom trigger in MPEG can be done by checking a variable in KHR.
- **Processing Model Differences:** In MPEG, behaviors are evaluated by the engine each frame: all triggers are polled/evaluated, and when conditions match, the associated actions are launched. It's a data-driven approach where the scene description lists behaviors and the engine continuously checks them. In `KHR_interactivity`, the behavior graph sits idle until an event node is invoked; then it propagates execution along the linked nodes. There isn't a concept of continuously checking a condition unless you explicitly set that up with an `OnTick` event. So effectively, MPEG's triggers that are continuously monitored map to either event nodes that are inherently continuous (`OnTick`) or to having multiple event nodes fire as needed. This means some things that are automatic in MPEG (like collision detection triggering an action) will, in a glTF context, depend on the viewer providing those events or the graph explicitly querying.

In summary, `MPEG_interactivity` and `KHR_interactivity` cover the same functionality but through different paradigms. To map MPEG to Khronos: triggers correspond to events, possibly requiring additional logic nodes in Khronos's interactivity extension. Variables and custom logic in Khronos can be used to achieve complex conditions of MPEG's interactivity.

#### **4.1.2.4. Processing and Execution Rules**

MPEG defines a clear processing model for how interactive behaviors are handled at runtime. When the scene is loaded, the Presentation Engine (the runtime) will parse the glTF and set up all the behaviors described in the `MPEG_scene_interactivity` extension. Each behavior knows its triggers and actions (and any node-level parameters from `MPEG_node_interactivity`). At runtime (typically each frame or each time the scene updates), the engine iterates through all defined behaviors and evaluates them as follows:

1. It checks the status of each trigger in that behavior – meaning it evaluates whether each trigger condition is currently active or has occurred (e.g., is collision X happening? is the button pressed? is object Y visible?). There is an underlying procedure or algorithm for each trigger type to determine its boolean state (the spec even provides a flowchart for trigger activation in the standard).
2. It then evaluates the logical combination of those triggers as specified by the behavior (applying the AND/OR/NOT from the triggersCombinationControl). This results in an overall true/false evaluation for the condition of the behavior in the current frame.
3. The engine then checks this result against the triggersActivationControl setting for the behavior. For example, if triggersActivationControl is “EACH\_ENTER,” the behavior should fire *at the moment* the condition goes from false to true. So the engine might compare the current condition state with the previous frame’s state to decide if this is a newly true event. If it’s “ON,” it would consider the behavior active continuously while the condition is true (possibly triggering actions continuously or at least keeping them active). The specifics are defined in the standard’s Table 12 (as listed in the extension): FIRST\_ENTER triggers once on true, EACH\_ENTER triggers every time it becomes true, etc. If the condition meets the criteria (e.g., it just became true for a FIRST\_ENTER, or it is true for ON), then
4. the engine launches the actions associated with the behavior. Launching actions means it executes each action in either sequential or parallel order as specified. Sequential actions might be executed over multiple frames if they have delays, whereas parallel actions are initiated together (for instance, starting an animation and a sound at the same time). Some actions (like play animation or play media) are instantaneous triggers that then proceed over time. The MPEG model accounts for actions that continue over time (e.g., an animation playing) by considering a behavior “ongoing” until those complete. If a scene update removes an ongoing behavior, the engine will execute the defined interruptAction (if any) to gracefully stop the behavior. Also, MPEG specifies that if multiple behaviors try to affect the same node simultaneously, the one with higher priority wins and the other is suppressed. This ensures determinism when two rules conflict (for example, one behavior says “move object up” and another says “move object down” at the same time – the one with higher priority will take effect). The entire cycle of checking triggers and updating actions repeats each frame or whenever the scene state changes. In effect, MPEG\_interactivity acts as a continuous rule evaluator: at any moment, it will respond to the current state of inputs by initiating the appropriate outputs.

The KHR\_interactivity behavior graph has a different execution model more akin to an event-driven system. Rather than continuously scanning conditions, it waits for events and propagates changes through the graph. Here’s how it works: When an interactive glTF asset is loaded, the viewer will initialize the behavior graph. This likely involves creating runtime representations of all the nodes (events, variables, etc.) and possibly initializing default variable values. Some event nodes may fire immediately upon load – notably, the OnStart (or equivalent) event node will trigger as soon as the graph is ready, which can start certain behaviors (e.g., an introductory animation) without any user input. During runtime, events occur either because of user interaction or as part of the system (for example, each frame an OnTick event may fire, or a custom event is emitted when an external condition is met). When an Event node fires, it emits a flow signal that travels along its outgoing connections in the graph. The nodes connected to that event’s output will then execute in order. Execution in the graph is generally instantaneous in the sense that in a single frame tick, an event can trigger a whole chain of nodes to run sequentially. Each node, upon

execution, may do something and then pass along flow to the next. For example, if an Event node is connected to an Action node: as soon as the event happens, the action node executes. If that action node has a flow output to another node (like another action), it will then trigger that next node, and so on – all within the same event invocation. There isn't an external loop checking all nodes; instead, nodes call one another via these links. This is a reactive execution: something happens (event), then reactions propagate.

During this propagation, flow control nodes can decide to branch or loop. A Branch (if/else) node will receive the flow, check a condition (e.g., compare a variable's value), and then send the flow down one of its two outputs (true path or false path). This is how conditions are evaluated in KHR\_interactivity – the condition check is just another node in the chain. For looping, the extension avoids infinite loops by design (no direct cycle in the graph), but a node might have multiple flow outputs (like a loop node could output back to an earlier part but that's likely forbidden to keep acyclic). Instead, repeating behavior is usually achieved by using OnTick events or by an action re-triggering an event. For instance, a repeating animation loop might be handled by having an animation node that upon finishing emits a custom event that loops back to start it again (that "loop" was described in the sofa example using a custom event to repeat). So loops are achieved by scheduling events rather than actual graph cycles.

#### **4.1.2.5. Example Scenario: Remote Control Triggers TV On (Video Playback)**

To illustrate the mapping, consider a scenario in an interactive 3D scene: A user reaches out and presses a virtual remote control object, which causes a 3D television in the scene to turn on and start playing a video. We will describe this scenario and then show how it can be implemented using both MPEG\_interactivity and KHR\_interactivity, including example JSON syntax and a diagram of the event flow for each.

The scene consists of a television set and a remote control as separate objects. Initially, the TV is "off", perhaps its screen is dark and no video is playing. The remote control is an object the user can interact with. When the user performs the appropriate input, e.g. clicking on the remote control object, that interaction is detected by the system. The interactivity logic then triggers two changes: (1) the TV's power state turns on, i.e. change its material to a lit screen), and (2) a video begins playing on the TV's screen.

This scenario involves one primary event (user presses remote) and two actions (turn on TV, play video). There could also be a condition, e.g. only do it if the TV was off, but for simplicity we'll assume the TV is off and the remote always turns it on.

Below we show how the MPEG\_interactivity JSON might look like, and then the equivalent KHR\_interactivity behavior graph JSON.

#### **MPEG\_interactivity Implementation**

In MPEG\_interactivity, we define a trigger for the remote-control input, actions for turning on the TV and playing the video, and a behavior that links them. We also assume a media is defined in the MPEG\_media extension.

The JSON snippet might look like this:

```

"extensionsUsed": [
    "MPEG_scene_interactivity",
    "MPEG_media"
],
"extensions": {
    "MPEG_media": {
        "media": [
            {
                "uri": "tv_video.mp4",
                "mimeType": "video/mp4"
            }
        ]
    }
},
"scene": 0,
"scenes": [
    {
        "extensions": {
            "MPEG_scene_interactivity": {
                "triggers": [
                    {
                        "type": "TRIGGER_USER_INPUT",
                        "userInputDescription": "/user/hand/right/input/select/click"
                    }
                ]
            }
        }
    ]
],

```

```

"actions": [

  {

    "type": "ACTION_ACTIVATE",

    "activationStatus": "ENABLED",

    "nodes": [ 1 ]    // assume node 1 = TV

  },

  {

    "type": "ACTION_MEDIA",

    "media": 0,          // play media index 0 (tv_video.mp4)

    "mediaControl": "MEDIA_PLAY"

  }

],

"behaviors": [

  {

    "triggers": [ 0 ],

    "actions": [ 0, 1 ],

    "triggersCombinationControl": "",

    "triggersActivationControl": "TRIGGER_ACTIVATE_EACH_ENTER",

    "actionsControl": "SEQUENTIAL"

  }

]

},

"nodes": [ ...

  { "name": "RemoteControl", /* remote node index 0 */ },

  { "name": "TV", /* TV node index 1, initially off/inactive */ }
]

```

```

    ... ]
  }
]

```

In this example, we use a `TRIGGER_USER_INPUT` for the remote control. The `userInputDescription` is given as an OpenXR path `"/user/hand/right/input/select/click"`, which represents a generic “select” action (like pulling a trigger or clicking) with the right hand. This implies that when the user performs the select action (while pointing at or near the remote, presumably), the trigger fires. We list two actions: an `ACTION_ACTIVATE` targeting the TV’s node (index 1) with `activationStatus: ENABLED` to turn it on, and an `ACTION_MEDIA` referencing media 0 (which in `MPEG_media.media[0]` is `tv_video.mp4`) with control `MEDIA_PLAY` to start playback of the video.

### KHR\_interactivity Implementation

Now, we implement the same scenario with `KHR_interactivity`. We need to create a behavior graph that listens for the remote press event and triggers the TV on + video play actions. In the glTF, this would be done inside the `KHR_interactivity` extension object. We will use a Custom Event node for the remote press, and two Operation nodes for the actions. We’ll also assume we have an extension for visibility or activation, and an extension to play the video. We assume the following:

- The Presentation Engine will send a custom event named `"remote_pressed"` when the user clicks the remote object.
- The TV’s visibility is controlled by a boolean property visible on the TV node.
- The video playback can be started by setting a parameter on a video texture.

Given those assumptions, the behavior graph JSON could look like:

```

"extensionsUsed": [
  "KHR_interactivity",
  "KHR_node_visibility",
  "EXT_texture_video",

"extensions": {
  "KHR_interactivity": {
    "nodes": [
      {
        "id": 0,
        "type": "Event",
        "eventType": "custom",

```

```

    "name": "remote_pressed",

    "outputs": {

        "flow": [ 1 ]

    }

},

{

    "id": 1,

    "type": "Operation",

    "operation": "setVisibility",

    "target": \{ "node": 1, "property": "visible" \},

    "value": true,

    "outputs": \{

        "flow": [ 2 ]

    }

},

{

    "id": 2,

    "type": "Operation",

    "operation": "playMedia",

    "target": { "node": 1, "media": 0 },

    "outputs": {}

}

]

}

}

```

This example highlights how the same interactive outcome is achieved through each extension. In MPEG\_interactivity, we relied on the predefined trigger and action types and simply listed them in a behavior object. In KHR\_interactivity, we manually built the logic using behavior graph nodes. Both realizations require integration with a media extension for the video and (in Khronos's case) a visibility/activation mechanism.

[1] WG03 N01221, 23090-14 2<sup>nd</sup> Edition, MPEG Scene Description

==

# Chapter 5. File Format

## 5.1. Revising carriage format for glTF in scene description

Source: [m75527](#)

### 5.1.1. General

ISO/IEC 23090-14[1] section 7.3.2 has defined how to store glTF as non-timed items in the ISOBMFF. However, it has two limitations in sharing scenarios: first, it does not support other features (such as HEIF) that require occupying file-level meta boxes; second, it lacks support for the glTF binary format (.glb). Therefore, this proposal puts forward a targeted solution.

### 5.1.2. Issue

ISO/IEC 23090-14 clause 7.3.2 specifies the following:

- “It shall use a **HandlerBox** with the `handler_type` set to 'gltf'”

Occupies the only `hdlr` under the unique file-level meta box, making it incompatible with other features (such as HEIF) that require occupying file-level meta boxes and using `hdlr` to specify their types.

- “It shall contain a **PrimaryItemBox** which declares as primary item a resource of type 'model/gltf+json'”

Lacks support for the 'model/gltf-binary' implementation scheme. When parsing "model/gltf+json," the glTF parser first reads the glTF JSON file and then accesses binary files (.bin) through URIs in the JSON. The URIs correspond to a physical disk address, thus requiring additional disk writing. Another feasible approach is to convert "model/gltf+json" to glb during the parsing glTF process, but both methods will increase data reading latency and degrade user experience.

### 5.1.3. Proposal

1. Supplement the solution for the scenario where the `hdlr` of the meta box is not gltf, thus supporting coexistence with other features that utilize the `hdlr` of the meta box.
2. Add support for .glb (model/gltf-binary) to avoid additional disk writing or converting "model/gltf+json" to glb.

In section 7.3.2, change:

The brand '**glti**' may be used to signal the use of a **MetaBox** with the following constraints:

- It shall be present at the file level.
- It shall use a **HandlerBox** [.mark]
  - Its `handler_type` is set to '**gltf**', such that only glTF files are contained in the **MetaBox**.

- It may contain a **GroupsListBox** with a **EntityToGroupBox** with the grouping type '**gltf**', containing an array of **entity\_id** which is resolved to this item and to all the tracks in this file referenced by the glTF object stored in this item.
- Its handler\_type is not set to '**gltf**', such that the **MetaBox** can contain not only glTF files but also other data items, such as videos, images, etc.
  - It must contain a **GroupsListBox** with a **EntityToGroupBox** with the grouping type '**gltf**', containing an array of **entity\_id** which is resolved to this item and to all the tracks in this file referenced by the glTF object stored in this item.
- It shall contain a **PrimaryItemBox** which declares as primary item a resource of type 'model/gltf+json' or 'model/gltf+binary'.
- It shall not use any **DataInformationBox**, **ItemProtectionBox** or **IPMPControlBox**.
- It shall use a **ItemInfoBox** '**iinf**' with the following constraints:
  - its version is either 0 or 1;
  - each item is described by an **ItemInfoEntry** '**infe**' with the following constraints:
    - its version is set to 0;
    - its **item\_protection\_index** is set to 0;
    - if the item is referred to by a URL in the content of another item, its **item\_name** is equal to that URL.
    - for JSON format file (.gltf), its **content\_type** is set to 'model/gltf+json'; For binary format file (.bin), its **content\_type** is set to 'application/gltf-buffer'; For glTF files stored in GLB container (.glb), its **content\_type** is set to 'model/gltf-binary'.
- It shall use an **ItemLocationBox** '**iloc**' with the following constraints:
  - its version is set to 1 or 2;
  - each item is described by an entry and values 0, 1 or 2 may be used for the construction method.
- It may use any other boxes (such as **ItemReferenceBox** '**iref**') not explicitly excluded above.

# Appendix A: Disclaimer



The formatting of the document is based on the Khronos glTF specification formatting under CC-BY 4.0.



The extensions information are automatically generated using [wetzel](#) tool under Apache License 2.0.