# QuickTime and ISO Base Media File Formats and Spatial and Immersive Media

## Format additions

Version 1.9.8 (Beta)

June 9, 2025

Note: The information contained within this document is preliminary and is subject to change.

# Introduction

This document describes Apple extensions of, or specialized use of, the ISO Base Media File Format (a.k.a. ISOBMFF) to support spatial media. These extensions also apply to the QuickTime File Format. Spatial media is intended to produce a richer experience for the user; whether a richer audio experience, a richer visual experience, or a combination of the two.

This document also introduces features that can be referenced from *movie profiles,* which are collections of allowed media essence carried in tracks and metadata, associated media signaling, and required relationships among the former, to ensure suitability for playback to devices or production for such delivery. Production, delivery and other aspects of profiles indicate use of particular *tools* such as encoded media with their own media profiles and levels, required metadata, or other signaling. If an implementation supports the movie profile's tools individually, they are well positioned to support the movie profile required in a broader ecosystem. This document introduces new format tools, and also movie profiles that make use of existing and new format tools. Movie profiles are documented separately.

Like the QuickTime File Format (QTFF) upon which it is based, the ISOBMFF format is meant to serve as a container of media using tracks and movie-level structures. The movie format of that media continues to evolve, from the earliest “postage stamp” (i.e., very low-resolution) video, with one- and two-channel uncompressed or barely compressed audio, to modern formats performing remarkable levels of visual compression for 4K and even 8K video, with very rich multichannel, ambisonic and object-based audio encoding. This is coupled with facilities to carry and present captions, such as WebVTT text tracks or closed captions embedded with video. Static and timed metadata can be carried to augment the presented media. Still other kinds of media tracks have been supported and will likely get added.

To support media that delivers rich spatial experiences, the QTFF and ISOBMFF foundations are being extended with new media formats, extensions to supported media formats, and new constructs to inform relationships among the new and earlier supported media. Some of these extensions are specific to their spatial nature, whereas others are fundamental and used by the former. This is all intended to be done in a way—where possible—so an existing ISOBMFF or QTFF player or processor can interact with the spatial media, possibly in a reduced but compatible form, while allowing new playback or processing to take fuller advantage of the newly afforded richness.

This document describes new and updated file format structures to support spatial media. Some of these structures are accessible through Apple AVFoundation and CoreMedia framework interfaces, and those serve as the preferred alternative to direct structural access when running on a platform with Apple frameworks available. Those reading or writing the format directly—pursuant to relevant licensing—should however be able to accomplish their goals with the structural descriptions in the following sections.

Another consideration for ISOBMFF is that it is used in a fragmented movie form for HTTP-based delivery technologies such as HTTP Live Streaming. The support in standalone MP4 files and fragmented MP4 resources is much the same.

Note: The words “may," “should," and "shall" are used in the conventional specification sense—that is, respectively, to denote permitted, recommended, or required behaviors.

Note: All coordinate systems used in this specification are right-handed unless otherwise specified. Following conventions, +X is to the right and +Y upward, as in quadrant 1 of a 2D Cartesian coordinate system. The Z-axis corresponds to the optical axis, and +Z points into or toward the camera.

# References

[QTFF] QuickTime File Format (QTFF), 2016

[ISOBMFF] ISO/IEC 14496-12:2020 ISO Base Media File Format

[ISONALU] ISO/IEC 14496-15:2019 “Carriage of network abstraction layer (NAL) unit structured video in the ISO base media file format”

[HEVC] ISO/IEC 23008-2:2020 “High efficiency video coding”

[HEIF] ISO/IEC 23008-12:2022 "Image File Format"

[METADATA] “Video Contour Map Payload Metadata within the QuickTime Movie File Format—Format Additions”

[OMNI] C. Mei and P. Rives. Single View Point Omnidirectional Camera Calibration from Planar Grids. In ICRA, 2007.

# Stereo Video

## Stereoscopic, Stereopsis and Stereo Media

Just as stereo audio indicates different audio for the left and the right ear, visual media can be stereoscopic, in which a view is available to be presented to the left eye and another view is available to be presented simultaneously to the right eye. The presentation of both the left and right views allows for an effect known as *stereopsis*, which can be defined as:

> *the perception of depth produced by the reception in the brain of visual stimuli from both eyes in combination; binocular vision*.

The production and display of this is sometimes referred to in cinema as *3D*, and the implementation and storage of the views can vary. This cinema use of *3D* should be distinguished from *3D rendering*, involving a framework like Apple's Metal framework, where geometry, materials, lighting and cameras are modeled and rendered by a GPU or CPU. In the latter case, such three-dimensional rendering might produce a view as seen from the left eye and a simultaneous view seen from the right eye and therefore be stereoscopic. Rendering of a scene might however produce a single view that is not stereoscopic. This is sometimes called *monoscopic*.

Stereoscopic media can also be captured photographically, where two cameras are offset horizontally to produce video where the left-eye view and the right-eye view are each encoded. In this case, there’s not necessarily any modeling of the scene or any GPU rendering. Playback presents the left-captured view to the viewer’s left eye and the right-captured view to the viewer’s right eye. These left- and right-captured views might also have been processed.

Although storage strategies can vary, this document describes how to store stereoscopic content using standardized ISO/ITU formats and extensions to QTFF/ISOBMFF.

## Stereoscopic Video Tracks

QTFF/ISOBMFF standalone and fragmented movies can include a single video track associated with both the left and the right eye. This video track's access units carry both a base and secondary layer that correspond to a primary stereo eye view (left or right) and the complementary stereo eye (i.e., right if the primary is the left, left if the primary is the right). Another option, available to production processes only, is to use frame packing, in which both left and right stereo eye views are carried adjacently in the decoded image in a single layer (either side-by-side or over-under).

The expectation is that both stereo-eye views (i.e., the left-eye view and right-eye view) are shown to their corresponding eyes simultaneously. Both stereo views are available and synchronized according to the movie timeline. When played, stereopsis is achieved.

> **Note:** The ability to produce a movie with just one stereo eye video track (whether left or right) can be useful in production workflows. Two tracks in the same movie or in two movies might be useful. These might be combined into a new movie either by encoding with both views in one video track or, less commonly, by carrying two video tracks. Though potentially applicable to QTFF/ISOBMFF, it is not a described use case here.

This document introduces a `VisualSampleEntry` extension that can signal, among other things, whether the associated video track is stereoscopic and which stereo eyes are carried in that video track. This new signaling is referred to as *video extended usage* and is described in a later section of this document. For now, the point is that this allows a movie reader to detect stereo-related video tracks and to identify the stereo eyes contributed by that track so it can configure presentation or other processing. Whereas the video track itself uses a video-compression format and signals it has a left and right stereo view, the video extended usage is meant to be more easily parsed and to be applicable to non-multiview video (i.e., not MV-HEVC video).

## Multiview Video Tracks and MV-HEVC Compression

A QTFF movie (i.e., .mov) or ISOBMFF (e.g., .mp4) movie video track carries video as either uncompressed or encoded video media samples. In the case of compressed video, High Efficiency Video Coding [HEVC] defines extensions to encode more than one view in the compressed bitstream for each coded video frame (or access unit). Defined in Annex G of the HEVC spec [HEVC], *Multiview High Efficiency Video Coding* defines how layers corresponding to views can be encoded and associated. This is sometimes written as *MV-HEVC*, for *Multiview HEVC*.

The QuickTime `ImageDescription` or ISOBMFF `VisualSampleEntry` (each referred to as visual sample entry) shall include a video extended usage visual sample-entry extension box (described later in this document) indicating which stereo eye views—left, right, or both—are carried in the MV-HEVC video track. For MV-HEVC, both left- and right-eye views should be available. A hero eye, indicating the default stereo eye, may optionally be signaled. This construct allows a client to determine the stereoscopic nature of the video track without needing to parse for MV-HEVC bitstream details in the decoder configuration.

The video bitstream requirements of MV-HEVC coding, visual sample entry, and video media samples are described in the document *Apple HEVC Stereo Video Interoperability Profile*.

The described use with MV-HEVC encoded video should not be seen as a limitation on applicability to other multilayered or multiview encoding.

# Video Extended Usage

This specification introduces an optional visual sample-entry extension that can signal additional aspects regarding the use of the video track's decoded frames. The new extension, called the *video extended usage*, uses the box type '`vexu`' (optionally pronounced as "vex you"). Details necessary for video-frame decoding are still carried in the visual sample-entry header (e.g., the compression type or the dimensions), as well as in other visual sample-entry extensions (e.g., '`colr`' and compression type specific decoder configurations). The '`vexu`' extension describes aspects beyond fundamental decode. For instance, it may specify that the video frames are stereoscopic or otherwise organized, requiring the video frames to be processed or displayed in a special way before presenting to the viewer.

Traditionally, a video track within a movie file or movie fragments can be decoded and immediately presented with little additional processing, other than perhaps scaling, cropping and placement. For video captured in the real world, such as from a camera or computer, this is the norm. Even non-linear editing mostly works with video as stored in the movie files, perhaps applying effects, but otherwise encoding video that is directly presentable.

Increasingly today, a video track may be used as input into a rendering process and may not be suitable to show a viewer immediately. For example, a stereoscopic "3D" movie should present the left-eye view to the viewer's left eye, and the right-eye view to the viewer's right eye. To do this, it is important to know first that the video track delivers stereoscopic views, and secondly which of those views are available. For MV-HEVC video, the presence of [HEVC] and [ISONALU] constructs (such as the '`hvcC`' and '`lhvC`' extension data) might seem sufficient, but unfortunately requires all readers to parse significant HEVC detail. Generally, a guiding principle is not to rely upon codec-level signaling to understand things that are needed at the container level. Here, the video track's '`vexu`' visual sample-entry extension serves the role of easy-to-interpret signaling that the video is stereoscopic. The 'vexu' visual sample-entry extension must be consistent with what is signaled in the decoder configuration.

Beyond MV-HEVC, it may be desirable to use other video-compression formats (e.g., non-Multiview HEVC) or uncompressed video to carry stereoscopic video without requiring their fundamental decoded bitstream to signal the stereo use—something the format might not support. A video extended usage extension can be added, indicating that a video track carries two stereo eyes or is for only one of the two stereo eyes. A '`vexu`' extension can also be added, indicating that the decoded video is organized in some other way described in a future version of this specification. This can be combined with stereoscopic detail that there is both a left and a right stereo eye view. Here, playback and processing need to understand the video track uses this alternative organization so it can route the left-view and the right-view portions of the decoded frame to the respective viewer eye.

## Video Extended Usage Box Hierarchy

The video extended usage extension box specifies the usage of the decode of the video samples, and details relevant to that usage. This is an optional extension and needed only when

special or useful interpretation of the video in playback or processing is required. If the state signaled is not required for playback or processing, the extension may still be present, but there is no expectation the reader understands it.

Note: VisualSampleEntry extensions parallel to this new video extended usage extension box may exist for additional, sometimes legacy, signaling (e.g., the horizontal field of view extension box with 'hfov' box type, described in the section "Horizontal field of view box" later in this document).

This extension box is a box hierarchy, and contains further boxes signaling particular aspects of the video. These contained boxes may be leaf boxes—typically a FullBox—or box hierarchies themselves. There is also a mechanism to indicate that contained boxes must be understood by a reader and, if not, that that part of the box hierarchy has failed to be processed. That error can propagate upwards, failing within a local subtree or even in the entire video extended usage box extension. This can in turn indicate the video should not be presented or processed, as the reader's implementation lacks sufficient support.

Note: The structure of the video extended usage box is made up of boxes, each described in following sections. Each box description indicates the mandatory nature of that box. In some cases, a box may be marked as No for Mandatory, in that the structure of the container box does not require this child box to exist. However, a particular movie profile (described in a parallel Movie Profiles document) may indicate that the box is required for conformance with that profile. This document does not indicate within each box definition which profiles require its presence. A profile feature table serves that purpose.

**Table 1.** Current box types defined in the 'vexu' box hierarchy

| FourCC | FourCC | FourCC | FourCC | FourCC | FourCC | Box syntax element |
|---|---|---|---|---|---|---|
| vexu | | | | | | VideoExtendedUsageBox |
| | must$_1$ | | | | | RequiredBoxTypesBox |
| | eyes | | | | | StereoViewBox |
| | | must$_1$ | | | | RequiredBoxTypesBox |
| | | stri | | | | StereoViewInformationBox |
| | | hero | | | | HeroStereoEyeDescriptionBox |
| | | cams | | | | StereoCameraSystemBox |
| | | | blin | | | StereoCameraSystemBaselineBox |
| | | cmfy | | | | StereoComfortBox |
| | | | dadj | | | StereoComfortDisparityAdjustmentBox |
| | proj | | | | | ProjectionBox |
| | | must$_1$ | | | | RequiredBoxTypesBox |

| FourCC | FourCC | FourCC | FourCC | FourCC | FourCC | Box syntax element |
|---|---|---|---|---|---|---|
| | | prji | | | | ProjectionInformationBox |
| | | . . . | | | | One of different per projection type boxes if needed |
| | pack | | | | | ViewPackingBox |
| | | must$_1$ | | | | RequiredBoxTypesBox |
| | | pkin | | | | ViewPackingInformationBox |
| | lnsc | | | | | CameraSystemLensCollectionBox |
| | | lens | | | | CameraSystemLensBox (one or more) |
| | | | lnhd | | | CameraSystemLensHeaderBox |
| | | | rdim | | | CameraSystemLensReferenceDimensions |
| | | | lnin | | | CameraSystemLensIntrinsicsBox |
| | | | ldst | | | CameraSystemLensDistortionsBox |
| | | | lfad | | | CameraSystemLensFrameAdjustmentBox |
| | | | lnex | | | CameraSystemLensExtrinsicsBox (one per lens) |
| | | | | must$_1$ | | RequiredBoxTypesBox |
| | | | | corg | | CameraSystemOriginSourceBox |
| | | | | cxfm | | CameraSystemTransformBox |
| | | | | | uqua | CameraSystemUnitQuaternionTransformBox |
| hfov | | | | | | HorizontalFieldOfViewBox |

**Note:** must$_1$: A must box may occur as a child box in any box hierarchy, to indicate reader responsibility for understanding any child boxes of the must box's parent box. These locations are not enumerated in the above table. Exemplar must boxes are included in the table, but these positions should not be considered exhaustive.

## 1. Video Extended Usage (‘vexu’) box

This section describes how the video extended usage extension box is organized and the constituent boxes.

### 1.1. Definition

Box Type: ‘vexu’
Container: Visual sample entry (different coding types)

Mandatory: No
Quantity: Zero or one

The video extended usage extension is a QuickTime File Format atom [QTFF], which is the same as Box in ISO/IEC 14496-12 [ISOBMFF]. As we use the bitstream syntax from ISO/IEC 14496-12, we use *box* interchangeably with *atom*. References to ImageDescription for QTFF are also interchangeable with references to the ISO 14496-12 VisualSampleEntry.

The video extended usage box is held in a `VideoExtendedUsageBox` and has the ISO box type of '`vexu`', for *video extended usage*.

As a box, it can contain zero or more child boxes that together signal the nature of the associated track samples' extended usage. Having no child boxes is valid but likely not useful. Having only child free-space boxes (i.e., a FreeSpaceBox) is appropriate if the intention is to reserve space in the `VisualSampleEntry`.

To allow new or otherwise unknown `VideoExtendedUsageBox` child boxes to be introduced while allowing older readers to know they do not understand enough to process or present the video track, a mechanism is introduced to indicate mandatory and, by implication, optional child boxes. Additionally, child boxes can indicate whether their own structure can be optional, so readers can recognize versions they do not support.

New boxes should not be introduced into the VideoExtendedUsageBox unless documented in this specification or in a successor version of this specification.

## 1.2. Syntax

```
aligned(8) class VideoExtendedUsageBox extends Box('vexu') {
   RequiredBoxTypesBox();          // optional if no required boxes
specified
   StereoViewBox();                // optional
   ViewPackingBox();               // optional
   ProjectionBox();                // optional
   CameraSystemLensCollectionBox() // required if lens signaling present
   Box() any_box;                // other optional boxes with FreeSpaceBox()
reserved for its expected use
}
```

## 1.3.Semantics

The `VideoExtendedUsageBox` contains zero or more child boxes that signal something about the use of the video. Child boxes will be defined in this specification now or in the future. Child boxes might be defined in external specifications, but the box_type used there should be registered so as not to collide with boxes introduced in this or related specifications. The order of child boxes in the `VideoExtendedUsageBox`, and in all contained boxes recursively, is not prescribed. A reader should be prepared to find boxes in any order.

> **Note:** As `FreeSpaceBox` ('free') has a very common meaning in ISOBMFF and QTFF, one or more `free-space boxes` may occur among the child boxes and should be interpreted as having no other meaning than taking up space. There is no guarantee that the payload of a FreeSpaceBox contains exclusively zero (0) bytes, but that is encouraged.

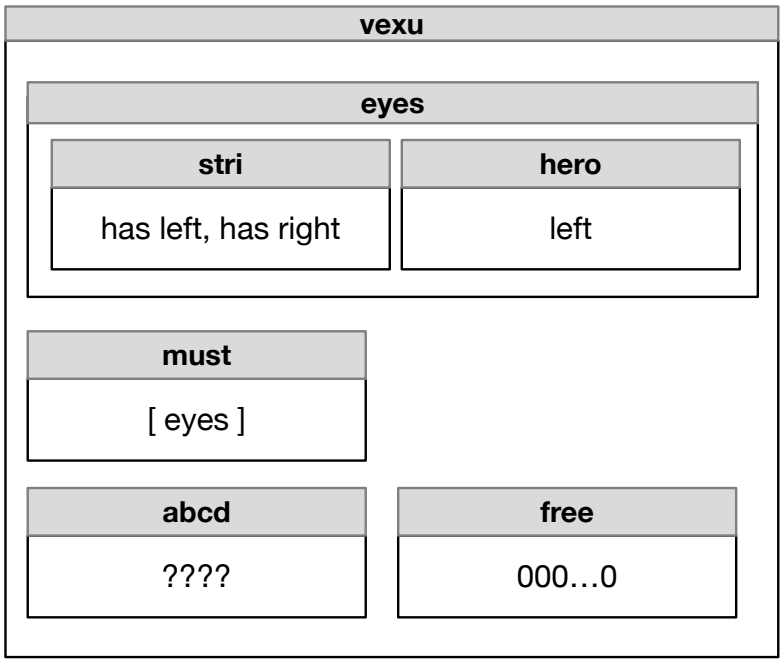
**Note:** An empty `VideoExtendedUsageBox` (i.e., containing no child boxes) is allowed but should generally not be included in the VisualSampleEntry. It may however be useful to reserve space by including a `VideoExtendedUsageBox` in concert with a contained `FreeSpaceBox` (‘`free`’).

New child boxes may be introduced in the future that are not described in this spec. There is a mechanism in the structure of VideoExtendedUsageBox to signal the set of child boxes that an implementation must understand in order to usefully process the video track. This allows future boxes to be introduced, and older implementations to know they should not present the video with newer signaled features.

Besides standard boxes, the `VideoExtendedUsageBox` may contain zero or more boxes that describe specific kinds of signaling. In the following section, each kind of box is described.

Note that a `VideoExtendedUsageBox` should carry only one child box for a specific feature. So, for example, there should not be more than one feature box for stereo-view signaling.

A `VideoExtendedUsageBox` with contained boxes might look like the following:




This ‘`vexu`’ box contains an ‘`eyes`’ box and another box. The second box, the '`abcd`' box, is optional, but if its type is included in the ‘`must`’ box, that indicates that it must be understood by the reader. ‘`free`’ boxes allow space to be reserved. The *other box* (i.e., '`abcd`') represents an unknown but not required box.

The order of child boxes within a box may vary. Readers should not expect a fixed order of child boxes at any level. A writer should not include a child box of a particular type more than once if it is documented to occur only once.

## 1.2. Required box types (‘must’) box

### 1.2.1. Definition

Box Type: ‘`must`’
Container: A video box within the video extended usage box (‘`vexu`’) or within contained boxes
Mandatory: No
Quantity: Zero or one

If a parent box at any level within the `VideoExtendedUsageBox` has a '`must`' box, that '`must`' box contains a list of box types corresponding to boxes that are peers to the '`must`' box and that the reader must successfully interpret in order for the parent box to be successfully interpreted. In other words, if the reader does not recognize a required box type, or if it fails to parse that box or any *required* child box of that box, the reader must consider that to be a failure to parse the parent box. If the

VideoExtendedUsageBox box is considered failed, the track is to be ignored by the reader.

Each kind of child box within a VideoExtendedUsageBox might serve to signal a feature according to this specification. The set of boxes is interpreted by the reader to understand what the video represents (e.g., it uses stereo views). Some of the signaling is necessary for further processing. Other boxes may be informative but not strictly required for interpretation. It's important to understand all required boxes. RequiredBoxTypesBox enables the adding of new boxes in the future that may be required for interpretation and further processing.

Each box within the VideoExtendedUsageBox may contain any hierarchy of boxes suitable to signal some aspect about the video. Some of these may be boxes with a hierarchy of other boxes, and some may be full boxes. The RequiredBoxTypesBox enumerates the box types of its sibling boxes, corresponding to required boxes. If not enumerated within the RequiredBoxTypesBox, the child box's interpretation is optional.

The RequiredBoxTypesBox contains an array of FourCCs, corresponding to box types. If an entry is 0, the entry is reserved and is not interpreted as a required box type. Free-space boxes of box type 'free' should not be included in a RequiredBoxTypesBox. If a RequiredBoxTypesBox includes a box type that doesn't correspond to a child box, the reader can ignore the absence but might want to log this for diagnostic purposes. The use of box types of missing boxes within a RequiredBoxTypesBox is, however, discouraged.

The FreeSpaceBox box type of 'free' should not be referenced from a RequiredBoxTypesBox.

The RequiredBoxTypesBox can also occur within other boxes within the VideoExtendedUsageBox box hierarchy that are themselves box hierarchies. These uses of RequiredBoxTypesBox serve to indicate local requirements on boxes that must be recognized and understood for local parsing to be valid. A local box can fail, and that influences the validity of the parent box if the parent box itself is referenced from another RequiredBoxTypesBox that is a sibling of the parent box.

### 1.2.2. Syntax

```
aligned(8) class RequiredBoxTypesBox extends FullBox('must', 0, 0 ) {
   unsigned int(32) required_box_types[];
}
```

### 1.2.3. Semantics

required_box_types is an array of zero or more box types corresponding to sibling boxes that must be understood by readers, to properly process the video associated with the VideoExtendedUsageBox. For each non-zero entry in required_box_types[], the reader should confirm the box type is recognized. A value of zero (0) in a required_box_types[] entry can be ignored, allowing for space for entries to be reserved.

### 1.2.4. Reader Behavior and RequiredBoxTypesBox

A reader of a movie-file video track with an associated `VideoExtendedUsageBox` should be able to detect whether it understands enough about the VideoExtendedUsageBox contents to process the video beyond fundamental decoding. This further processing, interpretation and/or rendering is what *extended* refers to within the identifier name "`VideoExtendedUsageBox`".

This specification is intended to be extended in future versions. A particular video track may carry several kinds of signaling that differ from other video tracks within the movie file, or in other movie files. The kind of signaling within a box can itself evolve over time. In all these cases, it is important to know if the set of child boxes of a box must be understood. Although the most obvious case is child boxes of `VideoExtendedUsageBox`, the approach can apply to any box serving as the root of a box hierarchy within the larger hierarchy.

The following describes reader behavior that is aware of `RequiredBoxTypesBox`:

1. Read (or start processing) the box hierarchy (e.g., `VideoExtendedUsageBox`).
2. Retrieve the contained `RequiredBoxTypesBox` child box, if any.
    1. If present, confirm all non-zero entries of `required_box_types[]` are recognized box types, and if not, treat the parent box of the `RequiredBoxTypesBox` as not processable.
    2. Ignore all zeroed entries of `required_box_types[].`
3. Enumerate each non-zero box type in the `required_box_types[]` of the child `RequiredBoxTypesBox` of the `VideoExtendedUsageBox`, using an index from 0 to the length of `required_box_types[]` minus 1, and confirm the referenced box is understood.
    1. For each successive `index`, retrieve the child box with `required_box_types[index]` and confirm understanding of its structure.
        1. For a child `FullBox`, the reader should consider the version and flags to confirm understanding, as well as anything else that may be relevant to its interpretation.
        2. For child boxes that are box hierarchies themselves and allow `RequiredBoxTypesBox`, the reader should descend into the box, retrieve the optional contained `RequiredBoxTypesBox` and perform this algorithm recursively.
        3. If the parsing of the `FullBox` or the child box hierarchy fails, the reader should treat the current level as invalid and propagate that failure upwards. Any semantics discovered at the current level should not be propagated upwards, as partial semantics is

misleading (e.g., stereo view and something else both being required should not signal stereo views if the other parsing fails).

The `VideoExtendedUsageBox` parsing follows this same algorithm. In this case, however, failing to parse required child boxes of `VideoExtendedUsageBox` means the track has failed to parse. This can best be interpreted as though the entire video track is unavailable.

## 1.3. Video stereo-view signaling

The `StereoViewBox` signals if the video track represents stereo 3D content. This can take the form of a track that delivers both a left and a right stereo-eye view, or a track that carries only the left or the right stereo eye.

If both left and right stereo eyes are carried, the views might be combined in one image and organized in some way, or they might be contained in some kind of multiview coding.

If the left stereo eye is in one video track and the right stereo eye is in a second video track, each carries its own `VideoExtendedUsageBox` with a `StereoViewBox`. The indication of which eye is carried is appropriate for each corresponding video track.

For completeness, it is also possible to signal monoscopic video, which is to say no stereo-view carriage. If this is the case, however, the `StereoViewBox` can be eliminated from the `VideoExtendedUsageBox`. If the `VideoExtendedUsageBox` would be left with no child boxes, the `VideoExtendedUsageBox` can be eliminated from the `VisualSampleEntry` as well.

If the recorded stereo video has a designated “hero” eye, the `StereoViewBox` carries a `HeroStereoEyeDescriptionBox`. There are rules that require signaling when the stereo eye video is separated into two video tracks, with each track carrying only one of the stereo eyes.

### 1.3.1. Definition

Box Type: `‘eyes’`
Container: Video extended usage box (`‘vexu’`)
Mandatory: No
Quantity: Zero or one

### 1.3.2. Syntax

```
aligned(8) class StereoViewBox extends Box(‘eyes’) {
   RequiredBoxTypesBox();                 // as needed
   StereoViewInformationBox();
   HeroStereoEyeDescriptionBox();         // optional
   StereoCameraSystemBox();               // optional
   StereoCameraSystemBaselineBox();       // optional
   Box[];                                 // other optional boxes
}
```

#### 1.3.3.Semantics

`StereoViewInformationBox` is a required box indicating which stereo eyes are present.

`RequiredBoxTypesBox` indicates the box types for other boxes that must be understood to interpret the current version of the `StereoViewsBox`. The `StereoViewInformationBox` box type of 'stri' is required within `RequiredBoxTypesBox` if a `RequiredBoxTypesBox` is used.

Other boxes indicate additional detail about the stereo-view representation and are described in later sections of this document. The set of boxes may evolve.

### 1.4. Stereo-view information ('stri')

#### 1.4.1. Definition

Box Type: `'stri'`
Container: Video stereo-view box (`'eyes'`)
Mandatory: Yes
Quantity: One

The `StereoViewInformationBox` can carry the stereography-related information, indicating the presence of particular stereo eyes (i.e., left stereo eye, right stereo eye), as well as some other flags.

#### 1.4.2.Syntax

```
aligned(8) class StereoViewInformationBox extends FullBox('stri', 0, 0) {
   unsigned int(4) reserved;       // reserved, set to 0
   unsigned int(1) eye_views_reversed;
   unsigned int(1) has_additional_views;
   unsigned int(1) has_right_eye_view;   // video contains a right-eye view
   unsigned int(1) has_left_eye_view;    // video contains a left-eye view
}
```

#### 1.4.3.Semantics

`has_left_eye_view`: Indicates the stereo left eye is present in video frames.

`has_right_eye_view`: Indicates the stereo right eye is present in video frames.

`has_additional_views`: Indicates that one or more additional views may be present beyond stereo left and stereo right eyes (e.g,. a "centerline" view).

`eye_views_reversed`: Indicates that the order of the stereo left eye and stereo right eye is reversed from the default order (left being first and right being second).

`reserved`: 4 bits reserved for future versions of this specification; for this version of this specification, writers should set it to 0, and readers should treat any non-zero values as if this box is invalid.

Because there is a flag field for the left eye and a field for the right eye, both fields should be set to indicate that both eyes are represented in video frames. Moreover, both

has_left_eye_view and has_right_eye_view can be set to 0 to indicate that the frame is monoscopic.

**Note:** If the video is monoscopic, the StereoViewBox can also be absent from the VideoExtendedUsageBox. If the only signaling is of monoscopic video, the VideoExtendedUsageBox can be absent from the VisualSampleEntry, too.

If an alternative organization is signaled in the future, the default order of stereo eyes in video will be left eye first, then right eye. Setting the eye_views_reversed field reverses the order, so the right view appears to the left of the frame, and the left view appears to the right of the frame. For multiview coding, there is no implied ordering, and the eye_views_reversed field should be set to 0.

**Note:** It may be useful to signal in a multiview coding approach the presence of the left stereo eye, the right stereo eye and a third view, which is the centerline or "down-the-nose" view between these and is neither the left nor the right. It may not be possible or appropriate to use the left or the right eye for this central view. There may be coding efficiencies from being able to include such a view in multiview coding.

The has_left_eye_view and has_right_eye_view fields specify the presence of the left and right stereo eye views, but the fields do not signal how those are stored. That is accomplished with other child boxes of StereoViewBox.

Note that the has_additional_views field indicates that views beyond those for the left eye and the right eye are present. One example of this might be a centerline view. Note that signaling the presence of the centerline is not necessary if both the left and right eye flags are zeroed, indicating a monoscopic view.

## 1.5. Hero Stereo-Eye Description

### 1.5.1. Definition

Box Type: 'hero'
Container: Video stereo-view box ('eyes')
Mandatory: No
Quantity: Zero or one

The HeroStereoEyeDescriptionBox indicates which stereo eye, if any, has been denoted as the hero eye. If so signaled, this indicates that the specified stereo eye view may be useful when choosing which eye to use in a monoscopic viewing environment. If neither eye is the hero eye, the HeroStereoEyeDescriptionBox does not need to be included in the StereoViewBox. If the hero eye is not known, a HeroStereoEyeDescriptionBox might not appear in the StereoViewBox.

It is possible to include a HeroStereoEyeDescriptionBox but set the flags to indicate that neither the left nor the right stereo eye is set. Though unconventional, this allows an implementation to reserve space for the box, to potentially set later in

processing. Readers should be prepared to recognize such a HeroStereoEyeDescriptionBox that signals no hero eye.

#### 1.5.2. Syntax

```
aligned(8) class HeroStereoEyeDescriptionBox extends FullBox('hero', 0, 0)
{
   unsigned int(8) hero_eye_indicator; // 0 = none, 1 = left, 2 = right, >=
3 reserved
}
```

#### 1.5.3. Semantics

HeroStereoEyeDescriptionBox is used to indicate which of the left or right stereo eye is the hero eye, if any.

hero_eye_indicator: Used in the HeroStereoEyeDescriptionBox, to signal which hero eye, if any, is specified. Defined values are:

0: The hero eye is not specified.

1: Indicates the left eye is the hero eye.

2: Indicates the right eye is the hero eye.

>= 3: Reserved, and should not be used for implementation of this version of this specification. If a reserved value is read, a reader should treat the signaling as though no hero eye is specified. If the hero eye is not specified, it is recommended that HeroStereoEyeDescriptionBox not be included in the StereoViewBox. The value of 0 is allowed, as it can be used to reserve space for the HeroStereoEyeDescriptionBox that might be adjusted later.

The HeroStereoEyeDescriptionBox signals the left or the right stereo eye independently of whether or not the stereo-view box's StereoViewInformationBox indicates that the order of the stereo eyes is reversed. The hero left eye is always the left stereo eye, and the hero right eye is always the right stereo eye.

## 1. Stereo Camera-System Box

### 1.1. Definition

Box Type: 'cams'
Container: Stereo-view box ('eyes')
Mandatory: No
Quantity: Zero or one

Stereo views are typically produced through capture by a camera system of some kind. Characteristics of that camera system may be useful for rendering or other processing. Moreover, the kinds of information associated with a camera system may vary now or in the future. As such, it seems prudent to allow one or more kinds of information to be carried for consideration by the processing client. It is also the case that this information is optional, as some stereo recording might not use a physical camera system. It may, however, be the case

that the camera system is virtual, so being able to include information that a physical camera system produces also seems useful to allow.

To signal information about the stereo nature of the camera system, the `StereoCameraSystemBox` is introduced. This is an optional `Box` instead of a `FullBox` so that a hierarchy of boxes and full boxes related to the camera system can be carried.

### 1.2. Syntax

```
aligned(8) class StereoCameraSystemBox extends Box('cams') {
   RequiredBoxTypesBox();               // optional
   StereoCameraSystemBaselineBox();     // optional
   Box[];   // other boxes that signal information about the camera system
}
```

### 1.3. Semantics

`StereoCameraSystemBaselineBox` is used to describe the baseline dimension between centers of the stereo lenses of the camera system. It is optional. There is no default interpretation when this box is absent.

## 2. Stereo Camera-System Baseline Box

### 2.1. Definition

Box Type: `'blin'`
Container: Stereo camera-system box (`'cams'`)
Mandatory: No
Quantity: Zero or one

Stereo camera systems may provide a number of characteristics that may prove useful to downstream rendering and processing of stereo frames captured by a camera system. One such characteristic is the distance between the optical centers of the left and right stereo eye camera lenses.

To signal information about the camera-system baseline, the `StereoCameraSystemBaselineBox` is used. This is an optional FullBox that holds a field with the distance in micrometers between the lenses. As distances in a camera system are typically expressed in millimeters, this use of micrometers can also be seen as being expressed in thousandths of a millimeter. The value is a fixed-point number expressed as an integer (i.e., 63123 micrometers is 63.123 millimeters).

> **Note:** Although `StereoCameraSystemBaselineBox` is not mandatory in the overall box structure of `VideoExtendedUsageBox`, its presence is required for the containing movie file to be considered "spatial media" by visionOS 2.

### 2.2. Syntax

```
aligned(8) class StereoCameraSystemBaselineBox extends FullBox('blin') {
```

```
   unsigned int(32) baseline_value;
}
```

### 2.3. Semantics

`baseline_value` holds the baseline dimension between centers of the stereo lenses of the camera system. It is an unsigned 32-bit integer that is interpreted in micrometers or thousandths of a millimeter (e.g., 63123 micrometers is 63.123 millimeters).

## 3. Stereo-Comfort Box

### 3.1. Definition

Box Type: `'cmfy'`
Container: Stereo-view box (`'eyes'`)
Mandatory: No
Quantity: Zero or one

Stereo views are presented to the left and the right eye in a display system. How that is performed can influence comfort. To allow different kinds of information related to stereo comfort to be carried with the stereo video frames, it is useful to allow one or more pieces of information to be carried.

To signal information influencing viewer comfort, the `StereoComfortBox` is introduced. This is a `Box` instead of a `FullBox`, so that a hierarchy of boxes and full boxes related to stereo comfort can be carried. This box is optional unless there are contained boxes. Only `StereoComfortDisparityAdjustmentBox` is currently defined.

### 3.2. Syntax

```
aligned(8) class StereoComfortBox extends Box('cmfy') {
   RequiredBoxTypesBox();                    // optional
   StereoComfortDisparityAdjustmentBox();    // optional
   Box[];   // other boxes that signal information about stereo comfort
}
```

### 3.3. Semantics

`StereoComfortDisparityAdjustmentBox` is used to describe any adjustment in the disparity between the stereo views of the current frame. The absence of this box or a zero value in the `disparity_adjustment` field of the box indicates no change in stereo disparity.

# 4.Stereo-Comfort Disparity Adjustment Box

## 4.1. Definition

Box Type: `'dadj'`
Container: Stereo-comfort box (`'cmfy'`)
Mandatory: No
Quantity: Zero or one

Stereo view comfort can be improved by allowing an adjustment in disparity to be specified. The source of this disparity is not specified, but the value's carriage is. As this adjustment is optional, this information may not be present.

To signal information about changes to the stereo comfort disparity, the `StereoComfortDisparityAdjustmentBox` is introduced. This is a FullBox holding a field that indicates the amount of disparity to apply. If the value is 0, there is no disparity adjustment. The value is a signed 32-bit integer that is interpreted as a uniform number over the range [-1.0...0.0...+1.0]. The valid range of the integer is from -10000 to +10000, which maps from -1.0 to +1.0. The interval of 0.0 to 1.0 and 0.0 to -1.0 are each mapped over the width of a view's image. Half of this value is applied to each stereo eye view.

The value is interpreted this way:

- Half the disparity is added to pixels in the left stereo eye view.
- Half the disparity is subtracted from pixels in the right stereo eye view.

As the disparity value is signed, adding a negative value to the left stereo eye is equivalent to subtracting the absolute value of the disparity value. The right stereo eye's subtraction of a negative disparity value is likewise equivalent to adding the absolute value of the disparity value.

Another interpretation of the sign of the disparity value is that positive denotes increased disparity with respect to the parallel view direction (e.g., horizontal) and negative denotes increased negative disparity with respect to the parallel view direction. Negative disparity is toward the viewer.

If the disparity adjustment value is 0, the `StereoComfortBox` need not contain a `StereoComfortDisparityAdjustmentBox`. If the `StereoComfortBox` would be left with no child boxes, the `StereoComfortBox` can itself be missing. There are rules, however, requiring the `StereoComfortBox` to exist.

> **Note:** Although `StereoComfortDisparityAdjustmentBox` is not mandatory in the overall box structure of `VideoExtendedUsageBox`, its presence is required for the containing movie file to be considered "spatial media" by visionOS 2.

## 4.2. Syntax

```
aligned(8) class StereoComfortDisparityAdjustmentBox extends
FullBox('dadj') {
   int(32) disparity_adjustment;
}
```

### 4.3. Semantics

`disparity_adjustment` holds a value describing an optional adjustment in stereo disparity. A value of 0 indicates there is no disparity adjustment and can also be represented by not including a `StereoComfortDisparityAdjustmentBox` in `StereoComfortBox`. The value is a signed 32-bit integer measured in the range of -10000 to 10000, mapping to the uniform range [-1.0...1.0]. The interval of 0.0 to 1.0 or 0 to 10000 maps onto the stereo eye view image width. The negative interval 0.0 to -1.0 or 0 to -10000 similarly maps onto the stereo eye view image width.

**Note:** This is not measured as a percentage but instead as a uniform value. To express 1.5%, the value 0.015 as a uniform value, or as the signed integer 150 (expressed over 10000), can be used.

**Note:** If the video is frame packed, for example, as side-by-side, the image width is for one stereo eye rather than the total image width of the side-by-side views. Other kinds of packings or arrangements also use the image width of the view, however stored or represented.

## 2. Video projection signaling

### Projections

A *projection* is a mathematical model mapping 3D world points onto 2D points such as an image or sensor. This mapping in a camera system can be achieved through optical elements such as lenses, through software or through a combination of the two.

### 2.1. Definition

Box Type: `'proj'`
Container: Video stereo-view box (`'eyes'`)
Mandatory: No
Quantity: Zero or one

If the video frame has some form of projection or other mathematical transform, the projection box may be used to indicate the algorithm and the approach taken. Downstream rendering likely needs to perform a mapping to present the frame in a way the viewer understands.

`ProjectionBox` signals that a form of projection is required to present the video track. Its contents signal one kind of projection.

### 2.2. Syntax

```
aligned(8) class ProjectionBox extends Box('proj') {
   RequiredBoxTypesBox();
   ProjectionInformationBox();
   FullEquirectangularProjectionBox();// optional: one of the kinds of the
possible projections
   HalfEquirectangularProjectionBox();// optional: one of the kinds of the
possible projections
```

```
   FisheyeProjectionBox();   // optional: one of the kinds of the possible
projections
   ParametricImmersiveMediaProjectionBox(); // optional: one of the kinds
of the possible projections
   RectilinearProjectionBox();  // optional: one of the kinds of the
possible projections (default if absent)
   Box[];                       // other kinds of boxes
}
```

### 2.3. Semantics

`ProjectionInformationBox` indicates the kind of projection described by the `ProjectionBox`. Each possible projection kind may be associated with a sibling box with additional parameters for that projection kind.

`RequiredBoxTypesBox` indicates the box types for other boxes that must be understood to interpret the current version of the `ProjectionBox`.

`FullEquirectangularProjectionBox` indicates that the projection is an equirectangular projection covering 360 degrees. More detail is found later in this specification.

`HalfEquirectangularProjectionBox` indicates that the projection is an equirectangular projection covering 180 degrees. More detail is found later in this specification.

`FisheyeProjectionBox` indicates that the projection is a fisheye projection. More detail is found later in this specification.

`ParametricImmersiveMediaProjectionBox` indicates that the projection uses the parametric lens projection.

`RectilinearProjectionBox` indicates that the projection is a rectilinear projection. More detail is found later in this specification.

Other kinds of projections may be introduced in future versions of this specification. Current versions of projections may also be extended in the future.

> **Note:** The projection kinds may not require any additional parameterization, so child box types for a particular projection kind may not exist. Nevertheless, the `box_type` for such projection kinds is reserved.

## 3. Projection information (‘prji’)

### 3.1. Definition

Box Type: `‘prji’`
Container: Projection box (`‘proj’`)
Mandatory: No
Quantity: One

The kinds of projections described by a ProjectionBox can vary according to this specification, and in future versions of this specification. The ProjectionInformationBox indicates the kind of projection carried.

### 3.2. Syntax

```
aligned(8) class ProjectionInformationBox extends FullBox('prji', 0, 0) {
   unsigned int(32) projection_kind;     // a FourCC for the kind of
projection
};
```

### 3.3. Semantics

projection_kind is a FourCC corresponding to the kind of projection. These currently include 'rect', 'equi', 'hequ' and 'prim' with the reserved 'fish' kind.

The ProjectionInformationBox can specify one kind of video projection for a video track. With this version of the specification, it should be one of the following projection_kind FourCCs with the corresponding projection kind of box:

- 'equi' : with optional FullEquirectangularProjectionBox, indicates a 360-degree projection;
- 'hequ' : HalfEquirectangularProjectionBox indicates a 180-degree projection;
- 'fish' : FisheyeProjectionBox indicates a projection of a fisheye lens, but reserved for legacy use;
- 'prim' : ParametricImmersiveMediaProjectionBox indicates a flexible approach to describing parameterized projection; and
- 'rect' : RectilinearProjectionBox indicates that the projection is rectilinear (with straight features), and serves as the default in the absence of the ProjectionBox within the VideoExtendedUsageBox. Because it is the default, this is likely not seen in movie files.

The ProjectionBox containing the ProjectionInformationBox may optionally have a child box, using a box type that corresponds to the projection_kind. This child box is a sibling of ProjectionInformationBox. It signals any parameters specific to that kind of projection, and reserves the corresponding box types. Although no occurrences of such boxes are defined in this specification, such usage is reserved for the future.

## 4. Rectilinear projection

### 4.1. Definition

Box Type: 'rect'
Container: Projection box ('proj')
Mandatory: No
Quantity: Zero or one

The rectilinear projection is the default for video. There is no further processing necessary to present video marked with just this projection. As the default, this can be

signaled by not including a `ProjectionBox`, or even by not including a `VideoExtendedUsageBox` if there is no other signaling.

If this projection is used, the sibling `ProjectionInformationBox` `projection_kind` should be set to ‘`rect`’.

It is possible to use this box to make it obvious what the intention is.

### 4.2. Syntax

```
aligned(8) class RectilinearProjectionBox extends FullBox(‘rect’, 0, 0) {
   // fields reserved for future use
};
```

### 4.3. Semantics

`RectilinearProjectionBox` signals a rectilinear projection. No fields are currently defined within the box, but a child box of type 'rect' is reserved for future addition to this specification.

## 6. Full Equirectangular projection

### 6.1. Definition

Box Type: ‘`equi`’
Container: Projection box (‘`proj`’)
Mandatory: No
Quantity: Zero or one

The Full`EquirectangularProjectionBox` indicates that the type of projection is a 360-degree projection using the equirectangular type. If no constraint on the angular limits is specified elsewhere, the limits are 0° to 360°.

If this projection is used, the sibling `ProjectionInformationBox` `projection_kind` should be set to ‘`equi`’.

### 6.2. Syntax

```
aligned(8) class FullEquirectangularProjectionBox extends FullBox(‘equi’,
0, 0) {
   // fields reserved for future use
};
```

### 6.3. Semantics

Full`EquirectangularProjectionBox` signals a 360-degree equirectangular projection. No fields are currently defined within the box, but a child box of type 'equi' is reserved for future addition to this specification for use of further details about the 360-degree projection.

## 7. Half-equirectangular projection

### 7.1. Definition

Box Type: ‘hequ’
Container: Projection box (‘proj’)
Mandatory: No
Quantity: Zero or one

The HalfEquirectangularProjectionBox indicates that the type of projection is a 180-degree projection based upon the equirectangular type. If no constraint on the angular limits are specified elsewhere, the limits should be 0° to 180°.

If this projection is used, the sibling ProjectionInformationBox projection_kind should be set to ‘hequ’.

### 7.2. Syntax

```
aligned(8) class HalfEquirectangularProjectionBox extends FullBox(‘hequ’,
0, 0) {
   // fields reserved for future use
};
```

### 7.3. Semantics

HalfEquirectangularProjectionBox signals a 180-degree half equirectangular projection. No fields are currently defined within the box, but a child box of type 'hequ' is reserved for future addition to this specification for use of further details about the 180-degree projection.

***Note:*** *The FourCC ‘hequ’ can optionally be pronounced “heck you.”.*

-

## 8.Fisheye projection

### 8.1. Definition

Box Type: ‘fish’
Container: Projection box (‘proj’)
Mandatory: No
Quantity: Zero or one

The fisheye projection is a projection from a hemisphere (typically a special lens). This kind of lens is often termed a *fisheye lens*, and so the projection uses the term *fisheye*. It is not, however, necessary to have captured with such a lens to produce a fisheye projection.

If this projection is used, the sibling ProjectionInformationBox projection_kind should be set to ‘fish’.

The projection kind 'fish' is reserved for a particular legacy implementation case and is not expanded upon here. For more general spherical projection, look at the ParametricImmersiveMediaProjectionBox ('prim') instead.

### 8.2. Syntax

```
aligned(8) class FisheyeProjectionBox extends FullBox('fish', 0, 0) {
   // fields reserved for future use
};
```

### 8.3. Semantics

`FisheyeProjectionBox` signals a fisheye projection typically covering a 180-degree field of view. No fields are currently defined within the box, but a child box of type 'fish' is reserved for future addition to this specification for further details about the fisheye projection.

-

## 9. Parametric Immersive Media projection

### 9.1. Definition

Box Type: `'prim'`
Container: Projection box (`'proj'`)
Mandatory: No
Quantity: Zero or one

The parametric immersive projection is a generalized spherical projection and is designed for significant flexibility.

If this projection is used, the sibling `ProjectionInformationBox` `projection_kind` should be set to 'prim'. A child box `ParametricImmersiveMediaProjectionBox` may be introduced to describe specific parameters needed, but currently none exist. Nevertheless, the child box of type 'prim' is reserved for future use.

The parametric immersive projection requires that the `VideoExtendedUsageBox` contains a `CameraSystemLensCollectionBox` characterizing the lens or lenses. This describes the lens intrinsics and extrinsics and other relevant information necessary for the projection to be applied.

### 9.2. Syntax

```
aligned(8) class ParametricImmersiveMediaProjectionBox extends
FullBox('prim', 0, 0) {
   // fields reserved for future use
};
```

### 9.3. Semantics

`ParametricImmersiveMediaProjectionBox` signals a parametric spherical projection, typically covering a 180-degree field of view. Unlike other current projections defined (i.e., 'rect', 'equi', 'hequ' and 'fish'), this projection does have a concrete child box of the `ProjectionBox`, specifically `ParametericImmersiveMediaProjectionBox`.

## 3. Video view packing

### 3.1. Definition

Box Type: `'pack'`
Container: Video extended usage box (`'vexu'`)
Mandatory: No
Quantity: Zero or one

A use of video tracks can be to encode more than one view in the same video frame. Different areas of the decoded frame produce different images. The View`PackingBox` can signal packing-related information. Types of packings can include frame-packed views and texture atlases.

> **Note:** Multiview encoding encodes multiple views in a way that no view is combined with another view in what is delivered. Packing, or *frame-packing*, involves multiple elements, combined in the same video frame and requiring addressing to extract just that view. Multiview coding allows individual views to be accessed without having to exclude portions of what is returned.

### 3.2. Syntax

```
aligned(8) class ViewPackingBox extends Box('pack') {
   RequiredBoxTypesBox();
   ViewPackingInformationBox();
   Box[];                    // other boxes
}
```

### 3.3. Semantics

`RequiredBoxTypesBox` indicates the box types for other boxes that must be understood to interpret the current version of the `PackingBox`.

Other boxes may occur. Some may be required by future versions of this specification.

-

## 3.4. View Packing information

### 3.4.1. Definition

Box Type: 'pkin'
Container: Packing box ('pack')
Mandatory: No
Quantity: One

The kinds of packing described by a ViewPackingBox can vary according to this specification and in future versions of this specification. The ViewPackingInformationBox indicates the kind of packing used.

### 3.4.2. Syntax

```
aligned(8) class ViewPackingInformationBox extends FullBox('pkin', 0, 0) {
   unsigned int(32) view_packing_kind;   // a FourCC for the kind of
projection
};
```

### 3.4.3. Semantics

view_packing_kind is a FourCC corresponding to the kind of packing performed. These currently include 'side' and 'over'.

The ViewPackingInformationBox can specify a single-current kind of video packing. It should be one of the following, with the corresponding value of view_packing_kind:

'side': Indicates a side-by-side arrangement of full-resolution video views arranged horizontally, where the width of each view is half the width of the containing video frame.

'over': Indicates an over-under arrangement of full-resolution video views arranged vertically, where one view is above the other and where the height of each view is 1/2 the height of the containing video frame

The absence of packing can be specified by not including a ViewPackingBox. Alternatively, a value of 0 can be specified for view_packing_kind, as a placeholder that might be specified later.

## 4.Lens collection information

Some projections and associated processing may require signaling for each of the lenses in the camera system. This information might include the lens pinhole intrinsics matrix, extensions beyond the basic matrix, and/or other parameters. The number of lenses might correspond to two for those needed in a stereo use case. Other lenses can be introduced in the future.

### 2.4.Definition

Box Type: 'lnsc'
Container: Video extended usage box ('vexu')

Mandatory: No
Quantity: Zero or one

### 2.5. Syntax

```
aligned(8) class CameraSystemLensCollectionBox extends Box('lnsc') {
   RequiredBoxTypesBox();                // as needed
   CameraSystemLensBox()[];              // one or more
   CameraSystemExtrinsicsBox();          // extrinsics for the camera
system
   Box[];                                // other optional boxes
}
```

### 2.6. Semantics

A `CameraSystemLensBox` is a required box for each relevant lens. If the camera system has lenses that are not important for the projection algorithm, a 'lens' box is not required for that lens.

`RequiredBoxTypesBox` indicates the box types for other boxes that must be understood to interpret the current version of the `CameraSystemLensCollectionBox`. The CameraSystemLensBox box type of 'lens' is required within `RequiredBoxTypesBox` if a `RequiredBoxTypesBox` is used.

If there are no lenses, the '`lnsc`' box should not be included in VisualExtendedUageBox. Boxes contained within each 'lens' box indicate additional detail about the camera-system lens and are described in later sections of this document. The set of boxes may evolve.

## 1.3. Camera system lens ('lens')

### 1.3.1. Definition

Box Type: '`lens`'
Container: Camera-system lens collection box ('`lnsc`')
Mandatory: Yes
Quantity: One or more

The `CameraSystemLensBox` carries the lens-related information for one lens in the camera system. There should be at least one 'lens' box if the camera-system projection algorithm requires information about the lenses.

### 1.3.2. Syntax

```
aligned(8) class CameraSystemLensBox extends FullBox('lens', 0, 0) {
   RequiredBoxTypesBox() must_box;             // as needed
   CameraSystemLensHeaderBox() header_box;     // identifier and algorithm
   CameraSystemLensReferenceDimensionsBox reference_dims;
   s() intrinsic_matrix;
   CameraSystemExtrinsicsBox();          // as needed
   Box[];         // other optional boxes
}
```

#### 1.3.3.Semantics

`CameraSystemLensHeaderBox`: Indicates the lens reference identifier and the algorithm associated with why lens parameters are signaled.
`CameraSystemLensReferenceDimensionsBox`: Indicates the dimensions (width and height) of the image of the lens projection (typically the entirety of the encoded buffer for multiview encoding, or the relevant horizontal or vertical half using frame-packing).

`CameraSystemLensIntrinsicsBox`: Signals the basic pinhole model intrinsics matrix, along with other values needed for the projection used by the lens.

`CameraSystemExtrinsicsBox`: Indicates the extrinsics that relate this lens to the overall camera system.
`RequiredBoxTypesBox:` Indicates the box types for other boxes that must be understood to interpret the current version of the `CameraSystemLensBox`. The `CameraSystemLensHeaderBox` box type of 'lnhd' is required within `RequiredBoxTypesBox` if a `RequiredBoxTypesBox` is used.

Though unlikely, different lenses can be described by 'lens' boxes with differing algorithms, requiring different parameters. This might be useful in the future if a camera system combines color-based lenses and depth-based lenses.

**Note:** The current spec version only supports lenses that capture color pixels. This may be extended in the future. If that is done, a new box may be introduced and signaled in the contained 'must' box.

## 3.5.Lens header information

### 3.5.1. Definition

Box Type: `'lnhd'`
Container: Camera-system lens box (`'lens'`)
Mandatory: Yes
Quantity: One

The `CameraSystemLensBox` can vary according to this specification, and in future versions of this specification. The `CameraSystemLensHeaderBox` indicates a unique identifier that can be used to reference this lens from other parts of the `VisualExtendedUsageBox`. It also signals the kind of algorithm so that the interpretation of parameters is possible.

### 3.5.2. Syntax

```
aligned(8) class CameraSystemLensHeaderBox extends FullBox('lnhd', 0, 0) {
   unsigned int(32) lens_identifier;     // an integer unique to the
enclosing CameraSystemLensBox
   unsigned int(32) lens_algorithm_kind; // a FourCC for the kind of
projection
      unsigned int(32) lens_domain;      // a FourCC for the kind of lens
(e.g., color)
```

```
      unsigned int(32) lens_role;          // a FourCC indicating which lens
this is (e.g., left or right for a stereo system)
};
```

### 3.5.3. Semantics

`lens_identifier` is a big-endian 32-bit integer that can be referenced from other structures in the `VisualExtendedUsageBox`. There is no prescribed interpretation of `lens_identifier`. The reading process can read values that differ if written out, so long as any references from other 'vexu' structures are updated consistently.

`lens_algorithm_kind`: Indicates the algorithm that makes use of the parameters held in the CameraSystemLensBox.

The `ProjectionInformationBox` `projection_kind` field specifies the current projection algorithm. In the current version of this specification, extended parameters should only be used with a `projection_kind` of '`prim`'. The corresponding value of `lens_algorithm_kind` indicates what additional lens collection signaling is required:

'`prim`': Indicates the "ProIM" algorithm [OMNI]. This algorithm requires both a sibling `CameraSystemLensIntrinsicsBox` and a `CameraSystemLensDistortionsBox`, with an optional `CameraSystemLensFrameAdjustmentBox`.

0: Indicates that the lens only describes its basic intrinsic matrix. That requires only a `CameraSystemLensIntrinsicMatrixBox`.

`lens_domain`: Indicates the type of sensor readings this lens produces. The value is a FourCC with one of these values:

The `lens_domain` can hold the value 'colr' to indicate that it produces color signaling such as RGB or YCbCr samples. For convenience, a value of 0 is equivalent to 'colr'.

In the future, `lens_domain` might indicate depth or another useful representation, such as from a Lidar sensor.

`lens_role`: Indicates the particular use of the lens in the camera system. For a stereoscopic camera system where a CameraSystemLensCollection is required, there should be an identifier for the left lens and another for the right lens.

'left': Indicates that this is the left lens in a stereoscopic camera system.
'rght': Indicates that this is the right lens in a stereoscopic camera system.

'mono': Indicates that this is the main or only camera lens in a monoscopic camera system.

0: Is a reserved value that should not be written in a movie, except in the case that the containing CameraSystemLensHeaderBox is written as all zeros (0s) to reserve space.

In the future, other values for `lens_role`, for other lenses, may be introduced.

**Note:** The `lens_role` should be used to indicate what purpose a lens serves. The `lens_identifier` values should not be assigned fixed values that are inferred as to the role of that lens (i.e., do not always use 1 to mean left lens and 2 to mean right lens). There may be correlation between the writer's assignment of `lens_identifier` and `lens_role`, but that should be treated only as coincidence.

## 3.6. Lens reference dimensions

### 3.6.1. Definition

Box Type: `'rdim'`
Container: Camera-system lens box (`'lens'`)
Mandatory: Yes
Quantity: One

The `CameraSystemLensBox` corresponding to a specific lens benefits from having the horizontal and vertical dimensions that may need to be interpreted by algorithms related to that lens. The `CameraSystemLensReferenceDimensionsBox` indicates the pixel width and height for the buffer processed by the lens. This should match the view bounds and doesn't necessarily correspond to the VisualSampleEntry dimensions (e.g., for frame-packed cases).

### 3.6.2. Syntax

```
aligned(8) class CameraSystemLensReferenceDimensionsBox extends
FullBox('rdim', 0, 0) {
   unsigned int(32) reference_width;
   unsigned int(32) reference_height;
};
```

### 3.6.3. Semantics

`reference_width` and `reference_height` are big-endian 32-bit integers for the pixel width and height of the buffers that the lens algorithm processes. If this does not match the VisualSampleEntry buffer width and height, the values used can be scaled by the ratio of `reference_width` to image buffer width and `reference_height` to image buffer height.

## 3.7. Lens intrinsic matrix

### 3.7.1. Definition

Box Type: `'lnin'`
Container: Camera-system lens box (`'lens'`)
Mandatory: Yes
Quantity: One

The CameraSystemLensIntrinsicsBox describes the camera-system intrinsics for the containing lens. If the camera system is a single-lens system, this corresponds to the intrinsics of the camera.

The intrinsic matrix is a 3x3 matrix of this form for a pinhole camera:

| | | |
|---|---|---|
| $f_x$ | s | $c_x$ |
| 0 | $f_y$ | $c_y$ |
| 0 | 0 | 1 |

where

- $f_x$ and $f_y$ are the horizontal and vertical focal length in pixels. For square pixels, they have the same value.
- *s* is an optional skew factor.
- $c_x$ and $c_y$ are the coordinates of the principal point. The origin is the upper left of the image frame.

  **Note:** The HEIF specification [HEIF] describes a 'cmin' CameraIntrinsicMatrixProperty but does not signal the cells with default values (matrix[0][1] = 0, matrix[1][0] = 0, matrix [2][0] and matrix [2][1] = 0, matrix[2][2] = 1). We adopt that practice here, using named fields instead of a general matrix 3x3 form. HEIF allows matrix[0][1] to be an optional non-zero skew_factor.

  **Note:** Most cameras have square pixels, with no skew. In this case, $f_x$ and $f_y$ are equal and skew *s* is zero (0).

The values of the intrinsic matrix can be calculated as follows:

The variable *denominator* is set equal to (1 << *denominatorShiftOperand)* where *denominatorShiftOperand* is in the value of the field `denominator_shift_operand`

The variable *skewDenominator* is set equal to (1 << *skewDenominatorShiftOperand)* where *skewDenominatorShiftOperand* is the value of the field `skew_denominator_shift_operand`.

$f_x$ = `focal_length_x` * `image_width` / *denominator*

$f_y$ = `focal_length_y` * `image_height` / *denominator*

$c_x$ = `principal_point_x` * `image_width` / *denominator*

$c_y$ = `principal_point_y` * image_height / *denominator*

*s* = `skew_factor` / *skewDenominator*

`image_width` and `image_height` come from `CameraSystemLensReferenceDimensionsBox` `reference_width` and `reference_height` fields.

### 3.7.2. Syntax

```
aligned(8) class CameraSystemLensIntrinsicsBox extends FullBox('lnin',
version = 0, flags) {
   signed int(16) denominator_shift_operand;
   signed int(16) skew_denominator_shift_operand;
   signed int(32) focal_length_x;
   signed int(32) principal_point_x;
   signed int(32) principal_point_y;
   if (flags & 1) {
      signed int(32) focal_length_y;
      signed int(32) skew_factor;
   }
   if (flags & 2) {
      BEFloat32 projection_offset;
   }
};
```

### 3.7.3. Semantics

A 3x3 matrix is not recorded in the `CameraSystemLensBox` but instead the `CameraSystemLensIntrinsicsBox` carries fields that can be used to calculate the cells of an equivalent 3x3 intrinsic matrix.

> **Note:** Unlike the HEIF '`cmin`' property item box [HEIF], the denominators are explicitly carried as discrete fields instead of being "hidden" in the FullBox `flags`.

The fields are:

`denominator_shift_operand` : The number of arithmetic shifts left to calculate the variable *denominator* (i.e., 1 << `denominator_shift`).
`skew_denominator_shift_operand` : The number of arithmetic shifts left to calculate the variable *skewDenominator* (i.e., 1 << `skew_denominator_shift`).
`focal_length_x` : The horizontal focal length measured in image widths.
`principal_point_x` : The principal point x-coordinate in image widths.
`principal_point_y` : The principal point y-coordinate in image heights.
`focal_length_y` : The vertical focal length, measured in image heights. If not specified by flags, the value of $f_y$ is equal to $f_x$.
`skew_factor` : The camera-system lens skew factor. If not present, the value is implied to be zero (0).
`projection_offset`: projection_offset is sometimes referred to as *xi*.

## 3.8.Lens distortions

Distortions in a lens may need to be characterized. This requires some kind of signaling to do this. It might be a set of parameters. It might be something more complex such as a 2D map of areas of the lens. For the parametric immersive projection, the lens

distortions can currently be characterized with four parameters (k1, k2, p1, and p2) and an optional radial limit carried in a new box as fields.

### 3.8.1. Definition

Box Type: `'ldst'`
Container: Camera-system lens box (`'lens'`)
Mandatory: No
Quantity: One

### 3.8.2. Syntax

```
aligned(8) class CameraSystemLensDistortionsBox extends FullBox('ldst',
version = 0, flags) {
   BEFloat32 k1;  // radial parameter k1
   BEFloat32 k2;  // radial parameter k2
   BEFloat32 p1;  // tangential parameter p1
   BEFloat32 p2;  // tangential parameter p2
   if (flags & 1) {
      BEFloat32 calibration_limit_radial_angle;
   }
};
```

### 3.8.3. Semantics

The fields are:

k1 specifies radial parameter k1 according to the Brown-Conrady distortion model.
k2 specifies radial parameter k2 according to the Brown-Conrady distortion model.
p1 specifies tangential parameter p1 according to the Brown-Conrady distortion model.
p2 specifies tangential parameter p2 according to the Brown-Conrady distortion model.
`calibration_limit_radial_angle` specifies the outer limit of the calibration validity in degrees of angle eccentric from the optical axis.

## 3.9.Lens calibration frame adjustment

To properly render camera-captured frames, sensor-produced images may require a post calibration mapping, or adjustment, to make the generated mesh compatible with the final video dimensions.

Toward this end, the optional `CameraSystemLensFrameAdjustmentBox` allows the specification of remapping polynomial parameters to describe the adjustment.

### 3.9.1. Definition

Box Type: `'lfad'`
Container: Camera-system lens box (`'lens'`)
Mandatory: No
Quantity: One

### 3.9.2. Syntax

```
aligned(8) class CameraSystemLensFrameAdjustmentBox extends FullBox('lfad',
version = 0, 0) {
   BEFloat32 polynomialParametersX[3];   // parameters for X axis
   BEFloat32 polynomialParametersY[3];   // parameters for Y axis
};
```

### 3.9.3. Semantics

`CameraSystemLensFrameAdjustmentBox` has two fields currently:

`polynomialParametersX` and `polynomialParametersY` each specify an array of exactly three `BEFloat32` values which together satisfy these equations:

$$x' = a_x + b_x*x + c_x*x^3$$

$$y' = a_y + b_y*y + c_y*y^3$$

where:

x is the x-axis source texture coordinate
y is the y-axis source texture coordinate

$a_x$ is `polynomialParametersX[0]`
$b_x$ is `polynomialParametersX[1]`
$c_x$ is `polynomialParametersX[2]`

$a_y$ is `polynomialParametersY[0]`
$b_y$ is `polynomialParametersY[1]`
$c_y$ is `polynomialParametersY[2]`

Also, define:

`referenceDimensions`: The raw sensor image size, or the final exported video size. This is the basis of source images used for the calibration process.

`cropDimensions`: If calibrated from the sensor images, this is the center crop size from the sensor image.

`videoDimensions`: The final exported video size.

The polynomial transform origin is at the center of the frame. {x,y} are the source texture UV coordinates in the range [-0.5, 0.5]. {0.0,0.0} is the origin.

The default values of elements of `polynomialParametersX[]` and `polynomialParametersY[]` are each `[0.0, 1.0, 0.0]`. Some use cases for setting parameters may be helpful to consider:

- If there is no cropping and no downsampling, or if it is calibrated from the exported frame size, the parameters should be set to [0.0, 1.0, 0.0].
- If there is a custom UV transform, please fit your transform to the specified polynomial here.
- For calibration from the sensor image with ISP applied, a regular UV transform with centered cropping followed by downsampling:

  Define:

  - `videoAspectRatio = video.width / video.height`
    `referenceDimensionsAspectRatio = referenceDimensions.width / referenceDimensions.height`
    `aspectRatioScale = videoAspectRatio / referenceDimensionsAspectRatio`
  - If in landscape mode, or in portrait mode with a 90-degree rotation track matrix (i.e., video.width >= video.height):
    - `cropScale = referenceDimension.width / cropDimension.width`
    - `polynomialParametersX = [0, cropScale, 0]`
      `polynomialParametersY = [0, aspectRatioScale * cropScale, 0]`
  - If in portrait mode without track matrix (i.e., video.width < video.height):
    - `cropScale = referenceDimension.width / cropDimension.height`
    - `polynomialParametersX = [0, cropScale / aspectRatioScale, 0]`
      `polynomialParametersY = [0, cropScale, 0]`

Also, note that if more array elements than three are introduced in the future or new fields are required, a future version of this specification will increase the version field of `CameraSystemLensFrameAdjustmentBox` and introduce new fields. The `polynomialParametersX` and `polynomialParametersY` fields should not be treated as dynamic arrays with more or fewer than three elements.

## 2.Camera-System Extrinsics

A camera system—whether a single-lens camera, a stereoscopic system with two, or a theoretical system with more lenses—has a relationship from its lens(es) to the world coordinate

system. This relationship establishes camera location in the world and is embodied in the *camera extrinsics*. Just as there is an intrinsic matrix for a lens, typically an extrinsic matrix establishes the orientation and position of the entire camera. Other models might use an extrinsic matrix establishing the lens' position and orientation. The extrinsic matrix embodies a rotation and translation. The rotation might be represented by a matrix of alternatively by a quaternion.

Instead of a single extrinsic matrix for the camera system, this specification takes the approach of describing a lens (or camera-system element) relative to the overall camera system. An outcome of this is that the signaling for a multi lens system is that each lens may duplicate the camera-system transform while identifying its relationship to a single point around which the transform is applied.

As a matrix, the extrinsic matrix might look like:

| | | | |
|---|---|---|---|
| $r_{1,1}$ | $r_{1,2}$ | $r_{1,3}$ | $t_1$ |
| $r_{2,1}$ | $r_{2,2}$ | $r_{2,3}$ | $t_2$ |
| $r_{3,1}$ | $r_{3,2}$ | $r_{3,3}$ | $t_3$ |

where $r_{i,k}$ cells embody rotation and the t vector positions the world origin in camera coordinates.

An alternative formulation to a 4x3 matrix is to signal the extrinsics with a quaternion for the rotation transform and some other indication of the camera's origin.

As mentioned in the introduction, coordinate systems user here are right-handed.

#### 2.1.1.1.1.Definition

Box Type: `'lnex'`
Container: Camera-system lens box (`'lens'`)
Mandatory: No
Quantity: Zero or one

The camera system may need to signal extrinsics. For a single-lens system, it is unambiguous (i.e., one cameras system has one lens has one extrinsic transform). For a multi lens system, each lens exists in relation to other lenses in the overall camera system and can signal per lens extrinsics using a `CameraSystemLensExtrinsicsBox`. There is still an overall notion of camera extrinsics and this is done by carrying a single `CameraSystemLensExtrinsicsBox` within the overall `CameraSystemLensCollectionBox` that describes the camera system. Initially, the approach is restricted in what is signaled but may be extended in future versions of this specification even to the point of describing each lens' contribution by referencing lenses by their lens identifier.

#### 2.1.1.1.2.Syntax

```
aligned(8) class CameraSystemLensExtrinsicsBox extends Box('lnex') {
   RequiredBoxTypesBox();                // as needed
   CameraSystemOriginSourceBox();        // optional
   CameraSystemTransformBox();           // optional
   Box[];
};
```

#### 2.1.1.1.3.Semantics

`CameraSystemOriginSourceBox:` a box indicating how the origin for the applied transform is applied with respect to this lens.
`CameraSystemTransformBox:` a box indicating the transform to apply to the origin of the camera system.

### 2.1.1.2.Camera-system origin source

#### 2.1.1.2.1.Definition

Box Type: `'corg'`
Container: Camera-system lens extrinsics box (`'lnex'`)
Mandatory: Yes
Quantity: One

The camera-system extrinsics indicate an origin for the camera system that will be coupled with a further transform. The origin may be derived from values instead of directly specified in the extrinsics containing box. The `CameraSystemOriginSourceBox` indicates how it is determined. Initially, the approach is restricted to a stereoscopic (binocular) system where translations are inferred along the baseline distance. This approach may be extended in future versions of this specification.

#### 2.1.1.2.2.Syntax

```
aligned(8) class CameraSystemOriginSourceBox extends FullBox('corg', 0,0) {
   unsigned int(32) source_of_origin;    // e.g., 'blin'
};
```

#### 2.1.1.2.3.Semantics

`source_of_origin:` Identifies how the origin of the camera system's extrinsics are determined.

This can hold these values:

'`blin`': Indicating the center of transform is determined by the point mid way along the dimensions indicated by the `StereoCameraSystemBaselineBox` held in the `StereoCameraSystemBox`. The lens_role in `CameraSystemLensHeaderBox` affects the interpretation in that the left lens distance to the origin is 1/2 the distance indicated in `StereoCameraSystemBaselineBox` and the right lens distance to the origin is the reverse of that direction but with the same magnitude of 1/2 the distance indicated in `StereoCameraSystemBaselineBox`.

In the future, other values may be introduced to indicate other sources of direction and magnitude to the origin for the transform.

Other values for origins may be introduced with new boxes or extensions to existing boxes.

#### 2.1.1.3.Camera system transform

The rotation or other transform to orient the camera system in world coordinates may take various forms depending upon the kind of projection used. New formulations may be introduced in future versions of this specification.

##### 2.1.1.3.1.Definition

Box Type: `'cxfm'`
Container: Camera-system lens extrinsics box (`'lnex'`)
Mandatory: Yes
Quantity: One

Whereas `CameraSystemOriginSourceBox` indicates the origin of the extrinsic transform, the `CameraSystemTransformBox` indicates the transform around that origin. Currently, a quaternion is used. In the future, a matrix or other representation may be introduced.

##### 2.1.1.3.2.Syntax

```
aligned(8) class CameraSystemTransformBox extends Box('cxfm') {
   RequiredBoxTypesBox();                       // as needed
   CameraSystemUnitQuaternionTransformBox();    // optional
   Box[];
};
```

##### 2.1.1.3.3.Semantics

`CameraSystemUnitQuaternionTransformBox:` A box holding a unit quaternion describing the rotation.

`RequiredBoxTypesBox`: In future versions of the specification, new mandatory boxes may be introduced that are required to be understood. At that time, the new box types can be signaled in a `RequiredBoxTypesBox` contained within the enclosing `CameraSystemTransformBox`.

#### 2.1.1.4.Camera-system unit quaternion transform

The rotation or other transform may be modeled as a quaternion. The `CameraSystemUnitQuaternionTransformBox` holds the three fields of a unit quaternion.

##### 2.1.1.4.1.Definition

Box Type: `'uqua'`
Container: Camera-system transform box (`'cxfm'`)
Mandatory: No
Quantity: Zero or one

`CameraSystemUnitQuaternionTransformBox` describes a single unit quaternion. It exists to be mappable to a runtime representation of a unit quaternion, such as the first three elements of the vector of a `simd_quatf` data type.

#### 2.1.1.4.2.Syntax

```
aligned(8) class CameraSystemUnitQuaternionTransformBox extends
FullBox('uqua', 0, 0) {
   BEFloat32 xyz[3];
};
```

#### 2.1.1.4.3.Semantics

`xyz:` A vector of three big-endian Float32 elements corresponding to the elements of a unit quaternion (i.e., a quaternion with a length of 1).

# Use of other signaling extensions

## Horizontal field-of-view box

The VisualSampleEntry defined an extension box to signal the horizontal field of view.

### 3. Definition

Box Type: `'hfov'`
Container: Visual sample entry
Mandatory: No
Quantity: Zero or one

The horizontal field of view of the decoded video frame image may be important to know. This can be signaled with the optional `HorizontalFieldOfViewBox` extension to `VisualSampleEntry`. The horizontal field of view is an unsigned integer in 1000ths of a degree (e.g., 123.456 is represented as 123456).

The `HorizontalFieldOfViewBox` is optional, but if present, there can be at most one `HorizontalFieldOfViewBox` in a `VisualSampleEntry`.

> **Note:** Although `HorizontalFieldOfViewBox` is not mandatory in the overall `VisualSampleEntry` structure parallel to `VideoExtendedUsageBox`, its presence is required for the containing movie file to be considered "spatial media" by visionOS 2. It may be present for other use cases beyond stereo video.

#### 3.1.1.1.1.Syntax

```
aligned(8) class HorizontalFieldOfViewBox extends Box('hfov') {
   unsigned int(32) field_of_view;
};
```

#### 3.1.1.1.2.Semantics

`field_of_view:` An unsigned 32-bit integer indicating the degrees, in 1000ths of a degree. A 104° field of view would be recorded as 104000.

# Auxiliary Video Track Handler Type

To date, video tracks have used the handler type 'vide'. Other track types such as audio and metadata use their own handler types (i.e., 'soun' and 'meta', respectively). This has never been a problem because the decoded video can be presented as is, though perhaps with scaling or cropping. With stereo video tracks, the decoded video may require additional processing such as view extraction before being presented to the user.

Movies or fragmented movie files for HTTP Live Streaming may now use the *auxiliary video,* or 'auxv', handler type to "hide" the video track from naive reader decode and presentation. For example, this can be useful for a video track with an alternative layout of images, signaled using a video extended usage atom.

If the decoded video, however, displays in a backwards-compatible way when delivered—such as MV-HEVC showing just a default view from the stereo pair—there is no need to use the 'auxv' handler type. Use 'vide' in this case. Also, HTTP Live Streaming mediates the media shown so that the multivariant playlist can serve to filter display of particular video streams.

In a production workflow, where users expect to see and confirm the decoded video even if further processing might be expected when delivered to an end user, it is okay to use 'vide' so tools that already present or process video tracks can find the track.

# Spatial Audio

The experience is enhanced if audio can represent the spatial acoustic environment. Just as listening to stereo audio is richer than listening to mono audio, even richer audio representations are possible with appropriate audio coding. A number of advanced audio technologies exist, and they may be used in isolation or in combination.

QTFF/ISOBMFF audio tracks use audio codecs to encode and carry audio—uncompressed and compressed—and the codecs can use different audio technologies. Some technologies are applicable to Spatial Audio, and when used in that way, the audio might be termed *Spatial Audio*. ISOBMFF can carry a wide range of uncompressed and compressed formats, some supporting spatial playback. By introducing new audio codecs in audio tracks, the movie can carry Spatial Audio.

ISOBMFF movies may contain any supported audio format. As additional formats are supported, those may prove useful for delivering more richer experiences.

## Spatial Audio Technologies

By way of a quick summary, there are three audio technologies typically used in the Spatial Audio realm.

*Channel-based audio* can include more than one audio channel, each of which is mapped onto the speaker layout. This is called *multichannel* audio, and is typically used with 5.1 and 7.1 audio. The number and placement of these channels in the soundscape can be more varied, and the

channel count can be more or less than the six of 5.1 or the eight of 7.1. Indeed, stereo has two channels, so is in fact multichannel, but that term is almost never applied to stereo.

Another technology, termed *ambisonics*, is a modeling of three-dimensional audio in a 360-degree space. It allows for the recording, mixing, and playing back of such audio. Just as multichannel audio can vary in channel count, ambisonic audio can vary in *order,* allowing more refined audio with higher-degreed ambisonics. Audio is fixed in location but surrounds the user.

A third technology, *object-based audio*, models each sound source as an object, with associated metadata describing three-dimensional placement and other relevant characteristics. Individual objects might be fixed in 3D or might move in 3D over time.

This specification does not prescribe which audio encoding formats, or which technologies within those formats, are used to realize a richer experience.

# Timed Metadata and Spatial Media

Spatial media tracks such as video may benefit from having associated timed metadata. Although this might be injected in AVC or HEVC SEI signaling, an alternative is to use a parallel metadata track. This timed metadata track can use the ISOBMFF ‘`mebx`’ format’s ability to carry a number of metadata items for a time range. Metadata item keys need not be related to other item keys, allowing a flexible way to signal a variety of structural or descriptive information.

We consider one kind of timed metadata payload related to describing the parallax of decoded stereoscopic video frames.

## New data types

Some new data types are introduced as composite payloads to support timed metadata introduced here. Although it would be possible to introduce keys for elements of these composite structures, it is deemed more efficient to treat each payload as a self-contained value.

### BEFloatVector3

```
aligned(8) class BEFloatVector3 {
  BEFloat32 vector_elements[3];    // three elements that are each a
BEFloat32
}
```

**Semantics:**

vector_elements[3]: An array of three big-endian IEEE 754 single floating-point values indexed from 0 to 2 for three elements. Each element is typed as a BEFloat32. The interpretation of each element is not prescribed, but a typical mapping might be x, y and z coordinates.

### BEFloatQuaternion

```
aligned (8) class BEFloatQuaternion {
```

```
  BEFloat32 quaternion_elements[4]; // four elements making up quaternion
}

aligned (8) class BEFloatUnitQuaternion {
  BEFloat32 unit_quaternion_elements[3]; // three elements making up a unit
quaternion having length 1.0
}
```

**Semantics:**

`quaternion_elements`[4]: An array of four big-endian IEEE 754 single floating-point values, indexed from 0 to 3, for four elements that map onto the four elements of a mathematical quaternion. Each element is typed as a BEFloat32. This vector of four elements should be mappable to a runtime's mathematical representation for a quaternion. Quaternions are defined by a scalar (real) part and three imaginary parts, collectively called the *vector part*. The order of elements in quaternion_elements corresponds to the real part (labeled r), then the three imaginary parts (labeled ix, iy and iz).

`unit_quaternion_elements`[3]: An array of three big-endian IEEE 754 single floating-point values, indexed from 0 to 2, for three elements that map onto the three elements of a unit mathematical quaternion. Each element is typed as a BEFloat32. This vector of three elements should be mappable to a runtime's mathematical representation for a unit quaternion. Unit quaternions have a length of 1.0.

## Caption-parallax timed metadata items

Traditionally, captions are placed in the horizontal and vertical axes over video. With the introduction of stereoscopic video, however, there is a risk of depth collision if captions are placed in Z, so they might intersect with stereoscopic elements that have a parallax (i.e., horizontal disparity) that is less than the screen plane. This “depth conflict” can produce viewer discomfort. To account for this, captions can have their parallax adjusted to have a more negative parallax than the video elements so there is no collision.

The document “Video Contour Map Payload Metadata within the QuickTime Movie File Format—Format Additions” [METADATA] specifies the structure of a metadata payload structure that can serve to describe parallax values associated with 2D areas of a stereoscopic video frame. This metadata is specific to the time-aligned video frame.

This payload is carried as metadata items within samples, within QuickTime File Format [QTFF] timed metadata or ISOBMFF [ISOBMFF] multiplexed metadata. Note: Both of these use the '`mebx`' format type of the '`meta`' track handler type. The payloads can also occur in fragmented movie files in both ISOBMFF and QTFF.

## Motion timed metadata items

Much of this document describes the local projections needed to deliver a stereoscopic experience at a point in time, corresponding to a single video frame in an overall timeline. When played as video, a spatial experience is produced that is dynamic as time progresses. If dynamic metadata were captured in parallel with the video that also characterized the motion of the capturing device and user, that may prove useful in ways that are unanticipated or anticipated. That motion might help in understanding how to deliver better user comfort, or where user

comfort might be compromised. Aside from its role in user comfort, this motion might prove useful for purposes that may only become available long after capture or initial delivery. That might take the form of new algorithms requiring novel metadata.

To that end, this section enumerates potential timed metadata items that seem general and are agnostic to how they are used. Novel services and algorithms can be built upon the presence of some or all of these motion-related metadata items.

### Accelerator, gyroscope and magnetometer

When carried in 'mebx' timed metadata [QTFF, ISOBMFF], there are three metadata items, each for a particular kind of motion data. Each key indicates the source of the value, and items in samples represent discrete readings from that sensor. The timescale of the timed metadata track can differ from that of the corresponding video, allowing for more frequent or less frequent sensor reading than the frequency of produced video frames.

Each value is a vector, with subfields for elements needed in the interpretation of the value, rather than separate metadata items for each field or element. These data-type payloads are documented in this specification. These use the following keys and value definitions:

| Keyspace | Key | Data type[1] | Well-known data type[2] |
|---|---|---|---|
| mdta | com.apple.quicktime.motion.accelerometer | `BEFloatVector3` | TBD |
| mdta | com.apple.quicktime.motion.gyroscope | `BEFloatVector3` | TBD |
| mdta | com.apple.quicktime.motion.magnetometer | `BEFloatVector3` | TBD |

**Note[1]:** BE refers to "big-endian."
**Note[2]**: ISOBMFF does not currently support signaling data types as QTFF does.

#### Semantics

The `com.apple.quicktime.motion.accelerometer` value is a `BEFloatVector3`, encoding G's m/s$^2$ relative to earth's gravity (9.81 m/s$^2$).

The `com.apple.quicktime.motion.gyroscope` value is a `BEFloatVector3`, encoding angular velocity in radians per second (rad/s).

The `com.apple.quicktime.motion.magnetometer` value is a `BEFloatVector3`, encoding magnetic field in microtesla (µT).

### Pose orientation and orientation heading

These metadata items may be interesting.

| Keyspace | Key | Data type[1] | Well-known data type[2] |
|---|---|---|---|
| mdta | com.apple.quicktime.motion.pose-orientation | `BEFloatQuaternion` | TBD |
| mdta | com.apple.quicktime.motion.orientation-heading | `BEFloat32` | 23 |

**Semantics**
The `com.apple.quicktime.motion.pose-orientation` value is a `BEFloatQuaternion`, encoding datum in quaternions.

The `com.apple.quicktime.motion.gyroscope` value is a `BEFloat32`, encoding Euler angles (degrees).

### Motion Events

The camera may experience unanticipated motion that may be useful to accommodate processing for comfort or other purposes. To capture these *motion events*, a metadata item is introduced to signal the event(s) from a collection of known event identifiers.

| Keyspace | Key | Data Type[1] | Well-Known Data Type[2] |
|---|---|---|---|
| mdta | com.apple.quicktime.motion.eventID | `BEUInt8` | 75 |

**Semantics**
The `com.apple.quicktime.motion.eventID` key is associated with a payload value that is a big-endian unsigned 8-bit integer holding one of the values from the following table. New events may be introduced in the future. Any unrecognized event IDs should be ignored, as opposed to producing a failure.

| Motion Event Name | Motion Event ID | Description |
|---|---|---|
| Motion Nominal | 0 | No motion event currently detected. |
| Level Departure | 1 | The camera is no longer in its nominal “horizon-level" state, as calibrated by the user. |
| Tumble | 2 | The camera orientation changes from a steady state to a high rotation rate on at least two axes. |
| Drop | 3 | The camera experiences rapid vertical acceleration from a steady state. |
| Shock | 4 | The camera is physically hit by an object at high speed, from any direction. |
| Roll | 5 | The camera experiences rapid roll moment from a steady state. |
| Pan | 6 | The camera experiences rapid pan moment from a steady state. |
| Tilt | 7 | The camera experiences rapid tilting moment from a steady state. |
| Heave | 8 | The camera detects a period of cyclical up-and-down motion after a steady state period. |

| Motion Event Name | Motion Event ID | Description |
|---|---|---|
| Sway | 9 | The camera detects a period of cyclical lateral and/or longitudinal motion after a steady state period. |
| Vibration | 10 | The camera detects high- or low-amplitude vibration in any direction at a frequency. |

There is no obligation for cameras to write motion-event metadata, but if they do so, the above metadata item "`mdta/com.apple.quicktime.motion.eventID`" and event ID payloads should be used.

The absence of motion events may be written in two ways:

1. Write the Motion Nominal (0) event ID.
2. Write no motion-metadata item to the metadata track for that time period. (Recommended)

## Comfort-related motion-analysis timed-metadata items

Motion-analysis timed metadata is composed of:

- Rotational motion indicator
- Perpendicular motion indicator
- Horizontal-panning motion indicator
- Vertical-panning motion indicator
- Motion score—mean from pixel differences
- Motion score—standard deviation from pixel differences
- Mean optical-flow magnitude
- Mean optical-flow magnitude in the horizontal direction
- Mean optical-flow magnitude in the vertical direction

| Keyspace | Key | Data Type[1] | Well-Known Data Type[2] |
|---|---|---|---|
| mdta | com.apple.quicktime.motion.metric.rotational | `BEFloat32` | 23 |
| mdta | com.apple.quicktime.motion.metric.perpendicular | `BEFloat32` | 23 |
| mdta | com.apple.quicktime.motion.metric.pan-horizontal | `BEFloat32` | 23 |
| mdta | com.apple.quicktime.motion.metric.pan-vertical | `BEFloat32` | 23 |
| mdta | com.apple.quicktime.motion.metric.pixel-difference-mean | `BEFloat32` | 23 |

| Keyspace | Key | Data Type[1] | Well-Known Data Type[2] |
|---|---|---|---|
| mdta | com.apple.quicktime.motion.metric.pixel-difference-standard-deviation | `BEFloat32` | 23 |
| mdta | com.apple.quicktime.motion.metric.flow.mean-magnitude | `BEFloat32` | 23 |
| mdta | com.apple.quicktime.motion.metric.flow.mean-horizontal-magnitude | `BEFloat32` | 23 |
| mdta | com.apple.quicktime.motion.metric.flow.mean-vertical-magnitude | `BEFloat32` | 23 |

**Semantics**
All of the above metrics are per frame and of type BEFloat32. The first four metrics are indicators for different kinds of camera motion and are generated from optical-flow analysis. They quantify rotational motion, perpendicular motion, horizontal pan, and vertical pan, respectively. Pixel difference is an overall motion score generated from analyzing pixel differences based on their mean and standard deviation. The last three metrics measure the overall optical-flow magnitude between consecutive frames.

# Conclusion

This document describes extensions to the QuickTime (.mov) and ISOBMFF movie formats. These extensions are introduced by Apple to allow for the delivery of stereoscopic video, spatial audio, and timed metadata signaling, to influence parallax of any subtitles associated with the video. This is applicable to both standalone movie and fragmented movie files. It attempts to build on existing structures where that was deemed appropriate, and introduces new constructs where there was a perceived deficit or a benefit in introducing a new construct. The evolution of the QTFF/ISOBMFF extensions described here may be taken through standards processes in time. This document also introduces the notion of movie profiles, which are described separately.

# Document Revision History

| Date | Revision | Notes |
| --- | --- | --- |
| 2025-06-09 | 1.9.8 | Beta version toward version 2.0 |
| 2023-06-21 | 0.9 | First version |