The Open Compound Eye Standard: A glTF Extension Definition Supporting Compound Eye Structures

File Version 0.3.0 [2023.09.18]

THIS DOCUMENT IS UNDER DEVELOPMENT. UNFINISHED OR UNDEFINED WORK IS HIGHLIGHTED IN RED.

Introduction

Over the years many ways of measuring the structure and receptive features of compound eyes have been introduced and used, ranging from simple spherical measurement of acceptance angles or ommatidial density, all the way to complete mappings of ommatdia in 3D space derived from microtomographic X-ray data. Through these methods an abundance of data has become available to the compound eye specialist. However, the lack of a unified standard of representation makes cross-comparison and even communication and distribution of compound eye data difficult. Further to this, as a new class of computer models arise to reconstruct the visual perception of compound eyes this lack of standardisation will result in incompatibilities between models and burden data collection teams with the task of making their data available in a number of formats to ensure that it has sufficient reach.

This document proposes an open compound eye standard built on top of the glTF standard for storing 3D scenegraphs. The new standard is designed to be capable of describing compound eye structures at three levels of representation of increasing fidelity, with methods to convert (through a lossy conversion) between each, with allowances for additional data to be stored, enabling the storage of as-yet-unknown properties of the compound eye in the future. It also describes supporting conventions, structures and implementation details to fully realise compound eye storage, finally supplying a fully-defined glTF extension JSON schema and example parsing libraries.

This document outlines the scope of the standard at a conceptual level (section: “OCES Primitives”), and then outlines the technical information describing the GlTF-based file format (section: “GlTF 2.0 Extension Specification”), before finally finishing with a detailed description and discussion of other technical information related to the manipulation and parsing of OCES-Augmented GlTF files, such as describing inter-primitive conversion methods, parser requirements and links to reference implementations (section: “Additional Implementation Reference Information”).

It is important to make note of where the words **should**, **could**, and **optional** are used throughout this document, as they describe parts of the compound eye description that are deemed absolutely necessary as the bare minimum of information needed to define a compound eye, compared to additional data that may be added to enhance the description of the eye. Parsers will **require** necessary information, but **will be required to** not change (destroy, move or otherwise alter such that it becomes inaccessible) additional eye property information found within an OCES-augmented glTF file.

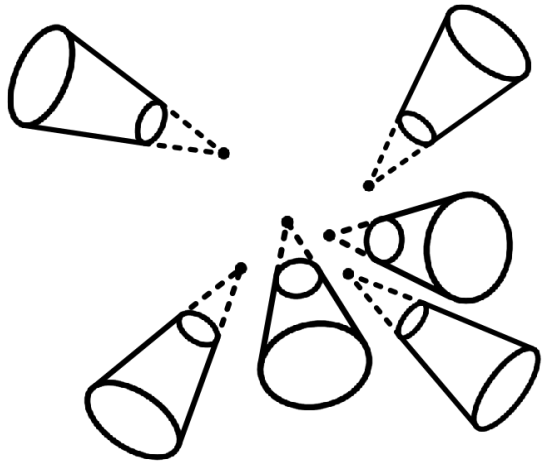
Glossary of Technical Terms

Scalar	A single real number
Integer	A single whole real number
Vector	A tuple of N real numbers defining a point in an N-dimensional space.
Coordinate Space	A named N-dimensional space. Specifically, a coordinate space can be defined by taking the translation, rotation or scale (or any matrix transformation) of the unit coordinate space (whereby each XYZ axis is of length 1). All coordinates defined in this space are subject to the same transformation. Examples of coordinate spaces used in this text are: ● Head Coordinate Space (HCS): The coordinate space defined by the transformation of a given **head** node. ● Eye Coordinate Space (ECS): The coordinate space defined by the transformation of a given **eye** node (which itself will be in HCS) ● Local Ommatidial Coordinate Space (LOCS): The coordinate space defined by the local axes at each ommatidium in the model.
Ommatidium / Ommatidia	An ommatidium is a single photosensing assembly (consisting of a light sensor and lensing arrangement), many of which (ommatidia) make up a compound eye.
Corneal Surface / Corneal Lens	When taken together, the many lenses of all ommatidia in a compound eye can be described as a corneal surface or corneal lens.

OCES Primitives

This section explains the 4 primary primitives defined in the OCES standard - 3 of which can be used to represent the ommatidial structure of an eye itself, and the final (a “head” primitive) acts as a grouping of compound eyes for full-head representation. These primitives are described first in brief (subsection: “Compound Eye Primitives Overview”), and then again in detail along with standardised descriptions of required local coordinate spaces (subsection: “Minimum Requirements and Extra Properties”). Finally, requirements for consistency and standard cohesiveness are discussed (subsection: “Ensuring Data Cohesiveness”).

Compound Eye Primitives Overview



At their core, compound eyes - to the point of light collection at the lens - can be described as a set of (potentially sheared) conical frusta oriented in 3D space, which describes the full-width-at-half-maximum (FWHM) of the light sampled by each ommatidium (referred to as the ommatidium’s **oriented sampling frustum**). Alongside these basic features, additional **properties** such as light sensitivity curves, facet shape, or polarisation sensitivity can be defined. Historically when an eye is examined only some of these properties are recorded - for instance, facet diameter and acceptance angle (used to construct the oriented samping frustum) are often recorded, but more obscure properties such as the relative shear values of sampling frusta (or the way they can change with respect to time), or polarisation sensitivity are often - but not always - omitted. To summarise, in this way, a single eye can be described as a set of ommatidia defined by their oriented sampling frusta, alongside optional extra properties that further describe the sampling methodology.

We have identified three separate ways that this data could be and is being (in modern works) represented using three varying levels of detail, from high-fidelity to low:

1. At the most concrete level we have identified the **point-ommatidial representation**, as described above, where the oriented view frusta and additional properties are stored directly on a per-ommatidial basis.
2. Following this is a **surface properties representation** in which rather than storing individual ommatidial data, a 3D surface is stored along with a value map of properties across this surface where these properties can be used to re-create individual ommatidial sampling frusta (a *point-ommatidial representation*) that then will match the specified property distribution.
3. Finally, at the most abstract level is a **spherical properties representation** - like the *surface properties representation*, only assuming a spherical surface) these consist of property distributions mapped around a spherical surface, as seen predominantly in older works often in the form of, acceptance angle plots for example.

Each level from point-ommatidial, through surface to spherical property representations are gradual abstractions and it is possible to lossily convert between each. For instance, to convert from high to low fidelity, eye surface meshes can be formed from point-ommatidial representations using algorithms typical to point-cloud meshing, converting per-ommatidium property values to per-vertex properties. Converting from either point-ommatidial or surface representations to a spherical property field becomes a task of simple projection onto the sphere. Due to the lossy nature of each abstraction, reversing these conversions and transforming from low fidelity representations to high will be prone to artifacting as they have to artificially expand on the given data. However, methods such as weighted point distribution over surface meshes can be used to convert from surface property models to point-ommatidial representations, and spherical property representations can be mapped onto surface property models by using the surface normal to sample from the spherical property maps.

As a fundamental requirement of rendering from the insect perspective requires exact and complete knowledge of the sampling frusta, only **point-ommatidial** representations can be used for rendering purposes, however for the purposes of comparative analysis, any of the three representations can be used. For this reason, the data standard allows all three to co-exist in one format, with individual software (renders, comparison programs, visualisers etc) dictating which level of detail is required.

Multi-Eyed Samples

So far we have discussed compound eyes as a singular unit - i.e. defining only one eye as a collection of ommatidia. However, in practice eyes do not exist in a vacuum - we'll often want to use two or more eyes as part of an entire head-based invertebrate visual system, perhaps even including additional ocelli. For this reason, all of the compound eye representations listed above should be taken as descriptions of single eyes (or ocelli groups), to be placed relative to a **head**. This simple primitive essentially acts as a root-object for compound eye primitives to be described relative to, handling the global pose of the eyes and allowing for common simple manipulation tasks - such as mirroring eye primitives along a given axis to produce left/right eye pairs (without having to store mirrored copies of the same data, which would otherwise be prone to incongruities when converting between types), and allowing for head-level transforms such as scaling, rotation and translation to be applied.

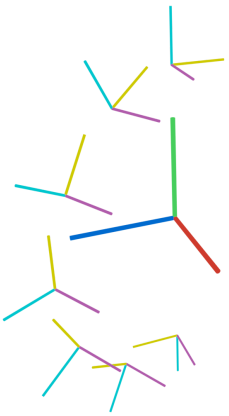
Minimum Requirements and Objects

So far we have discussed the data required to represent and manipulate compound eyes in broad strokes, namely as a set of oriented points (or objects from which oriented points can be derived, an "eye") placed in relation to a single oriented point (a "head"). What now follows is a more in-depth description of the supporting elements and ideas.

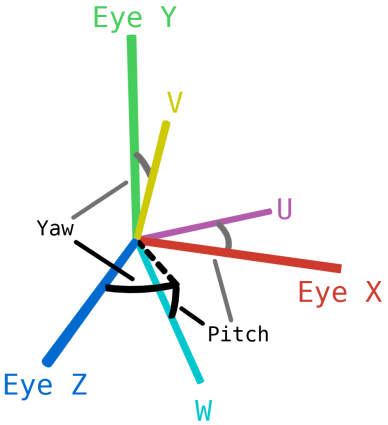
Local Ommatidial Coordinate Space

A number of ommatidial properties require a 3D coordinate space relative to each ommatidium - as any property that can be described spatially as off the ommatidial axis requires a coordinate system that can be reliably reconstructed for each ommatidium. While

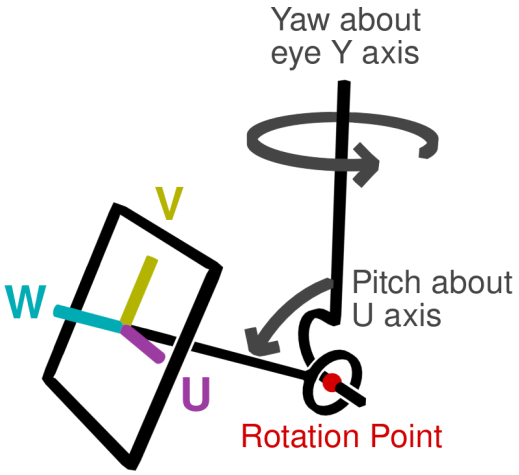
the head- and eye-level coordinate systems are implicitly stored as the relative transform from the head's coordinate space to the eye's coordinate space, each ommatidia will require its own individual coordinate space to define these relative, off-axis vectors. Further to this, the construction of this space (which we will refer to as the **Local Ommatidial Coordinate Space**, or **LOCS**) must be well-defined to ensure congruence between different systems implementing the OCES standard.



Following typical approaches to forming pitch-yaw camera systems in interactive media, we propose constructing a local U/V/W coordinate system in alignment with the the **eye's coordinate system** (**ECS**): By taking the ommatidial axis as the initial local axis (here-on W), a further axis perpendicular to it and the eye's vertical (Y) axis can be constructed (U), with a final local axis constructed perpendicular to both W and U (here called V), forming the rotated counter-part of the eye's Y axis. In the cases where the ommatidial axis (W) is parallel to the eye's Y axis, this method suffers from gimbal lock as an infinite number of coordinate spaces could be placed around the Y and W axes. In these two edge cases, we simply constrain the U axis to be parallel to the X axis.



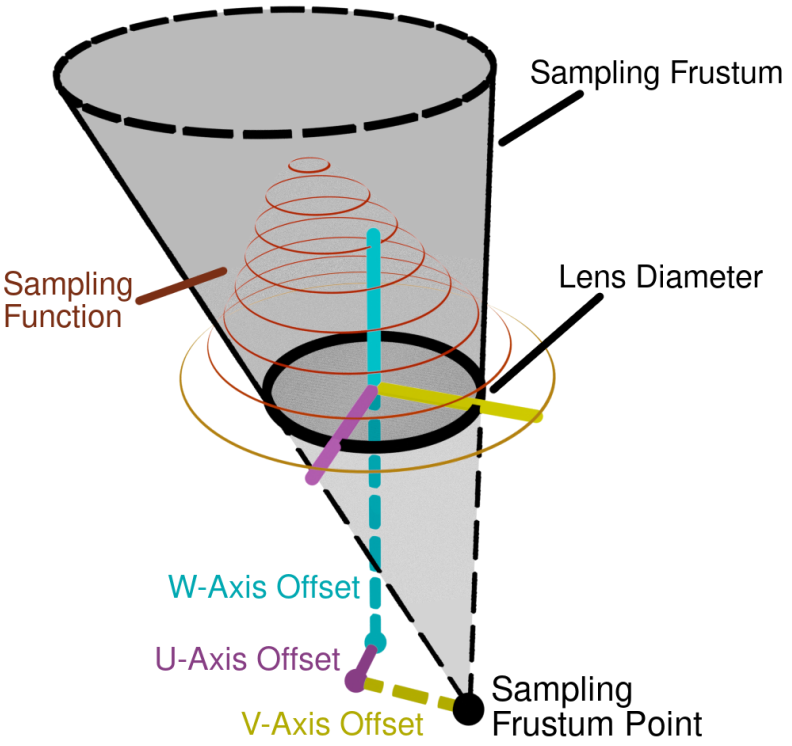
A further way of thinking about this coordinate space is to imagine a rotatable hinge joint:



Notice that the U/V/W coordinate space will never experience roll - U will always be perpendicular to the Y axis, with W and V being constructed perpendicular to that.

Point-Ommatidial Representation

As outlined in the "Compound Eye Primitives Overview" section, the point-ommatidial representation is the most direct representation of a collection of ommatidia. It consists of the minimum required constraints to define each oriented, sheared frustum in **LOC** space that represents the conical spread of the ommatidium's sampling function into the environment.



In the sample image above, a U- and V- local ommatidial coordinate space axial offset is demonstrated, with the W-axial offset (that is always negative) controlling the frustum's conical spread, and the U- and V- axis controlling the frustum's shear about the origin along each respective axis. In practice, however, U- and V- axial offset information is rarely collected. Additionally, most compound eye records do not directly store the W-axis offset, but instead store the acceptance angle and the facet/lens diameter. In this case, the W-axis offset can be calculated by dividing the lens radius by the tangent of half the acceptance angle:

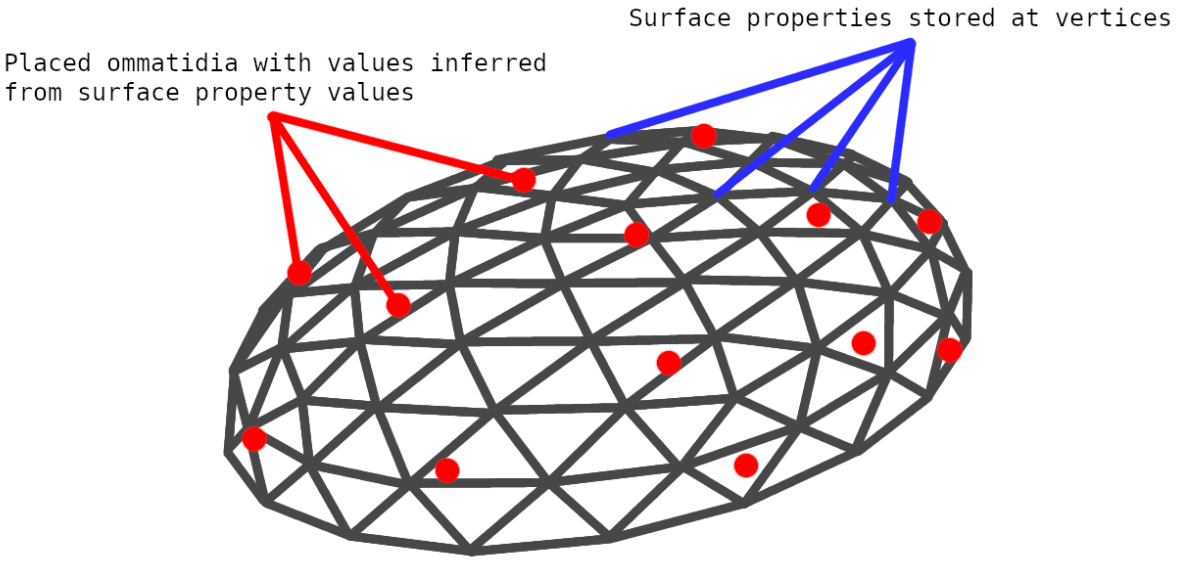
$$lensRadius \div tan(AcceptanceAngle/2)$$

Aside from the 3 3D vectors and one scalar required to represent the sampling frustum (*POSITION, ORIENTATION, FOCAL OFFSET and LENS DIAMETER*), any other properties may be appended on a per-ommatidial basis. Examples include polarisation acceptance angle, axial offset time series data/light response curves or colour sensitivity curves. While these might enrich the model of the compound eye, they should be considered **optional**, and sub-standards defined for interpreting this additional data (see subsection "Defining Additional Data Standards" under "Additional Implementation Reference Information" for more information on how to go about doing this).

Surface Properties Representation

Rather than describing the individual ommatidia themselves, a surface properties representation describes the distribution of ommatidial properties across the corneal surface

of the eye. In this case by default the *POSITION* and *ORIENTATION* of the sampling frustum is implicitly defined by the surface position and normal direction on the surface. The LENS DIAMETER and *FOCAL OFFSET* properties are then stored as “textures” across the surface, in the same way that computer graphics systems store colour and rendering data across the surface of 3D models (note that in the examples here we are showing the data stored at each vertex of the surface - known as *vertex shading* in computer graphics - but the data could also be stored in images via a *UV mapping* approach).
In this way, individual ommatidia can be reconstructed by “placing” them with regard to facet diameter across the surface, producing a point-ommatidial representation ready for rendering:



As alluded to at the beginning of this section, while *POSITION* and *ORIENTATION* are taken from the surface position and normal, it is entirely possible and desired to be able to encode additional alterations to this data - for example, if the orientation is not uniformly parallel to the surface normal, an **optional** *RELATIVE ORIENTATION* property map may be provided that represents an offset in local ommatidial coordinate space (**LOCS**). These surface property maps act just like any other property map and are still optional, however have the following significance attached to them:

- RELATIVE ORIENTATION: Describes the relative orientation (in the **LOCS**) of the ommatidia, to be summed and normalised with the surface-normal derived orientation.
- ABSOLUTE ORIENTATION: Describes a new absolute orientation (in the eye’s coordinate system - **ECS**) that completely overwrites the surface-normal derived orientation.
- DISPLACEMENT: Describes a displacement (in the **LOCS**) from the eye’s surface, similar to a displacement map seen in computer graphics.

Spherical Properties Representation

As the spherical property representation is simply values spread over the surface of a sphere, they can be thought of as akin to any spherically-mapped data such as those found in cube-maps (from the field of computer graphics), or ommatidial acceptance angle maps (from compound eye literature). There are plenty of data structures that allow for the storage

of spherical data, with the storage of spherical data of particular interest in the field of computer graphics in the form of environment and reflection maps.
The use of these maps allows for a very similar representation to the **surface properties representation**, except with the assumption that the surface is always a sphere. Instead of looking properties up directly from vertex- or UV- shading data, a single image per property is stored and referenced depending on the orientation of the ommatidium (or potential ommatidium) it is being referenced by, in the same way that environment and reflection maps are sampled in graphics engines.

Besides the additional optional properties of *RELATIVE ORIENTATION*, *ABSOLUTE ORIENTATION*, and *DISPLACEMENT* defined in the surface properties representation that all apply here, this type of eye representation also requires a "radius" value, which is a single value that describes the average radius of the eye. While this value does not affect the use of this form of representation directly, it allows this form to be compared directly to the others, and allows for the conversion from this form to the other two using the same surface-point-placement approach outlined in the previous representation.

Unfortunately, GITF currently has no set standard for representing spherical image data, but supporting updates are currently being discussed (🔗). Until a choice is made and the standard updated, the OCES specifies that the "latitude/longitude" format (where the horizontal axis represents increasing angular distance away from the eye's ZY-plane, and the vertical axis represents increasing angular distance away from the eye's XZ-plane, with the Z axis positively "extending" from the centre of the image) must be used for spherical image data. See the "Surface Properties Representation" subsection of the "Proposed Compound Eye Extension" section below for further information.

Coarse vs Fine Properties

So far, we have made the assumption that properties are defined on the per-ommatidial, or surface-level basis. However, in the case where a property is unknown or only known in estimate format (for example, when the average facet diameter of an eye is known, but not its distribution across the surface), it is inefficient to store repeat copies of the same property for every ommatidia in the eye or across an entire property map, and can also lead to difficulties in data congruence if a file is incorrectly updated and only a portion of the values change. To this end, "coarse properties" can be defined, which are single instances of a property, rather than an entire per-ommatidia or spatial map. These are to be assumed to be the "default" values of any property lookup at the spatial or ommatidial level, if no property is already defined.

Head Object and Mirror Planes

As described in the "Multi-Eyed Samples" subsection of the "Compound Eye Primitives Overview", the eye data representations alone do not account for a complete model of a head that utilises compound eyes. To this end, a **head** object needs to be defined. At its core, this is simply a named reference frame that a set of compound eyes can be described in relation to. Aside from this purpose, another common requirement of compound eyes on a head is to be mirrored across common (usually ventrodorsal) planes. To this end, we define that a set of **mirror planes** be defined in the head's coordinate space (**HCS**), which can then

be referenced from individual eyes to perform mirroring operations without having to duplicate the data whole-sale.

Ensuring Data Cohesiveness

A standard's primary job is to ensure that data can be encoded, transmitted and decoded easily. In aid of this goal, it is important to foresee and curb potential areas of confusion whereby two similar yet different OCES-supported GlTF files may lead to undefined interpretations of their data.

Sub-Extensions

Despite our best efforts, it is not possible to fully account for every possible piece of data that an eye renderer might require - even the provisioning of optional eye properties can lead to confusion as to their context and uses. To this end, we have limited the base OCES definition to the absolute minimum required to describe the acceptance fields of compound eyes.

Taking from the GlTF approach to standard extension, provisions have been made for specifying extensions to the OCES standard, allowing people to add the context required to understand the contents of the file where it is supplying non-OCES-required data (either additional optional eye properties or full extensions). These extensions will be stored alongside the official OCES definition after a simple open submission vetting process, ensuring that standards are always unified and centrally located, while still being collaboratively supportable.

Scale

The varieties of scale involved with insect research are often of vastly different orders of magnitude - an environment can span several kilometres while an eye can be only a handful of micrometres across. To this end, we explicitly follow the GlTF standard's scaling system, whereby all units are in metres, leaving the option open to structure an eye in such a way that scaling can be performed uniformly to the entire eye, by leveraging GlTF's scene graph structure, allowing for the eye to be specified in one metric and then "wrapped" in a scaling node to make it compliant with the metre-wise scale of the rest of the file.

Versioning

As standards are subject to change, it is imperative that versioning is tracked. To this end, the OCES standard itself will always contain a version identifier, and all official extensions will be required to do the same.

glTF 2.0 OCES Extension Specification

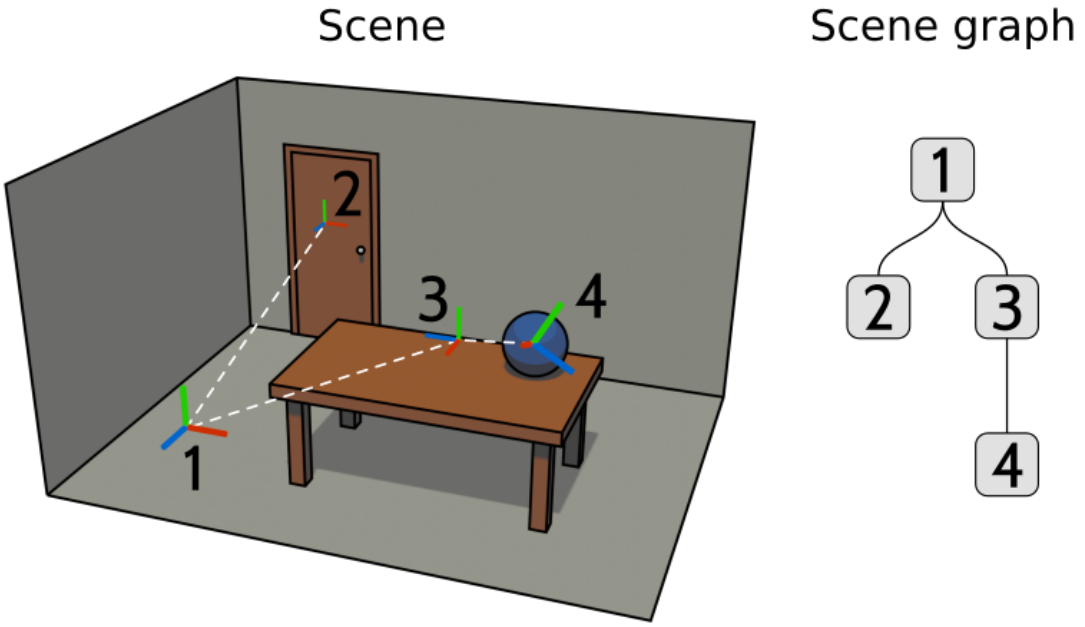
Introduction

This section of the document gives a brief overview of the glTF 2.0 file format for scene representation and outlines the OCES glTF 2.0 extension that is capable of storing any number of the four primary primitives described in the *OCES Primitives* section.

The glTF 2.0 Format Recap

The glTF file format is an extensible, well-understood and widely-supported file format for describing and transmitting 3D scenes including 3D geometry (meshes), animation keyframes/motion targets, textures, lights (although implementation differences in renderers make this less useful) and cameras.

GlTF files consist of a structured JSON file (either in human-readable utf-8 format [.gltf], or as a condensed binary format [.glb]). In practical terms, the JSON file consists of a **top-level object** that contains **scenes**, **nodes**, **meshes**, **cameras**, **materials**, **samplers**, **accessors**, **bufferViews** and **buffers**. Multiple **scene** graphs are stored as a list of trees of **node** objects, containing relative transform and scene structure data, making references to **meshes** (which in turn can reference a list of **materials** either directly or via a **sampler**):



In the example above, a simple scene graph is composed of 4 objects - the room (*1*), a door (*2*), a table (*3*), and a ball (*4*). Each object has its own local coordinate space as indicated by the axes visible at each object, and each coordinate space is transformed via some *rotation*, *translation*, or *scale*. Each object in the scene has a "parent" object (for instance, the ball's parent is the table, and the table's parent is the room) that "stores" it (in practice, this means that each child object is described spatially relative to its parent, using its parent's coordinate system). On the right of the image above you can see the "scene graph" of the scene - that is, the parent-child relationships between each object in the scene. In this scene the room is the "root object" as it is the single object that all scene objects are descendants of. In a GlTF file, a scene graph is constructed out of **nodes**, which may or may not have attached **mesh** objects - in this way, a mesh can be arbitrarily placed within a scene via any degree of separation from the root node, with a hierarchy suited to the spatial structure of the scene. For instance, in the above example, when moving the table it makes sense for the ball to move with it. Similarly, it follows that moving the room itself would move the contents of that room.

Rather than storing data directly within mesh and material objects, data is - in general - stored in **buffers**. Buffers are simply long streams of data, typically stored as a byte array. They can either be included explicitly in the file as a raw byte array or externally as a separate binary file that is pointed to in the glTF file proper. Most glTF files will hold a single buffer that stores *all* data within the file.

As mentioned, this data is referenced primarily by mesh and material nodes. This is done using **buffer views** and **accessors**, which work together to act as “windows” into the buffer primitives. Buffer views specify a section of a buffer (it’s length in bytes and position) and explain how its bytes are arranged, while accessors specify how to interpret a buffer view - what type it is a list of (2,3,4-vector, scalar, short, int, float etc). These buffer views are what are referenced by meshes and materials (via samplers). Appendix A shows a sample glTF file that makes use of most of the elements discussed above, and Appendix B shows the complete glTF Object Hierarchy diagram taken from the glTF 2.0 specification document.

A more in-depth description of the glTF file format can be found in the glTF 2.0 file format specification document, or by following the community-documented glTF Tutorial.

Extensions

The glTF file format is extensible - it supports two types of extension mechanisms: **extras** and **extensions**. Extras are a simple method for adding additional data to a scene graph - all objects can have an “extras” key that can be used to store arbitrary data on any node. Extras are not strictly specified: they are free-form, non-namespaced data that can be added to any node within the glTF file. For this reason it can be difficult to standardise on this type of extension mechanic beyond requiring simple data or options, so their use is avoided within the OCES standard. However, many 3D modelling software packages (such as Blender) allow you to manually append these extras within their UI, making it very easy to manipulate these values without the requirement of specialised export plugins, making them potentially a useful vessel for inter-program support (see “Developer Notes” under “Schema and Reference Implementations” for further information on how they might be useful).

The second type of extension mechanism, **extensions**, are more well-formed extensions to the glTF file format that specify new primitive object types - for instance, one might specify a new object primitive of type “curve” if you wanted to extend the glTF format to support Bezier or NURBS curves (someone has conveniently implemented this here for this example). They are much more powerful in terms of being able to specify a full complex object structure, and allow a more formal procedure to describing and sharing extensions. Many companies and groups have created extensions that are now listed on the official glTF extension registry.

All extensions have a unique *namespace* - a name that uniquely identifies all additions provided by the namespace. In this way, extensions can be thought of as namespaced extras. All objects within the GlTF file (including the top-level object) can have an “extensions” key that contains a sub-object with namespace-separated extension-specific information. For example, the top-level object using the OCES eye extension may look like this:

```
Unset
...
"extensionsUsed" : [
	"OCES_eyes"
],
"extensions" : {
	"OCES_eyes" : {
		"version" : "1.0",
		"unitScal" : 0.01
	}
}
...
```

Here, only one extension is specified as in-use: the “OCES_eyes” extension. It is added to the list of used extensions by including its namespace (“OCES_eyes”) in the “extensionsUsed” key of the top-level object (which is how all extensions must register themselves as being used in a file), and then extension-related data is listed under the “OCES_eyes” sub-object of the “extensions” object.

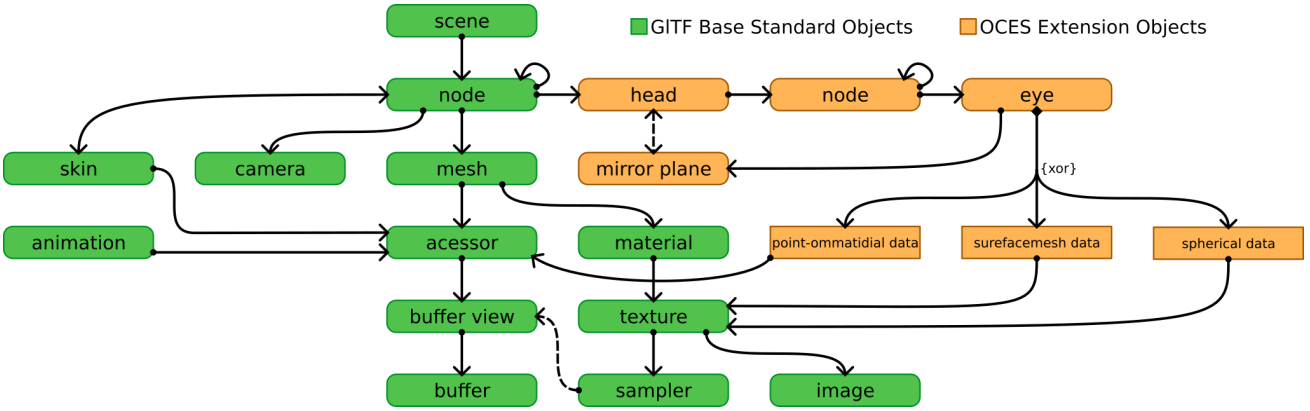
As the “extensions” key exists on any GITF object, similar to the “extras” key, it is then used to attach extension-specific values to any object, for instance, here we see a node with an OCES eye and mirror planes attached:

```
Unset
...
"nodes" : [
	{
		"name" : "example-eye-node",
		"extensions" : {
			"OCES_eyes" : {
				"eye" : 0,
				"mirrorPlanes" : [0,2]
			}
		}
	}
]
...
```

Here a simple **node** has an **eye** and two **mirror plane** references attached to it. For further information on how extensions work in the GITF file format, please see the [extensions section of the GITF standard].

Proposed Compound Eye Extension

We propose a set of additional glTF primitives, implemented as extensions, that mirror the four primitives defined in the “OCES Primitives” section.



The above diagram shows an adapted copy of the GlTF objects diagram (provided in Appendix B), with a number of additional objects defined in orange that make up the OCES extension. Each rounded-edge box indicates a full object, while the square-edged “-data” objects are data specifications, one of each can be used to define an “eye” object. Below, each of these objects are examined and their properties listed.

Appendix C contains an annotated example OCES-enabled GlTF file, please refer to it if you are unsure where definitions lie within an entire file.

Head

As a **head** object is a simple grouping indicator, it is essentially just a specialised node, so all that is needed to define a head object is a “head” key holding the value of *true* in the node’s OCES extension definition:

```
Unset
...
"nodes" : [
	{
		"name" : "head-example",
		...
		"extensions" : {
			"OCES_eyes" : {
				"head" : true,
				"enabled" : true,
			}
		}
	}
],
...
```

This allows any arbitrary node to be defined as a **head** object. It is expected that no head objects will be defined as descendants of any **head** object, and that all **eye** objects will be attached as descendants of a **head** object, so that any **eye** can simply search upward through the scene graph to find the **head** associated with them.

Additionally, a head may be “enabled” or “disabled” by switching the “enabled” key to *true* or *false*. This can be used as an indicator to programs processing OCES-enabled GITF files as to whether or not to, for example, render from this **head**, and should override any true “enabled” values of any **eyes** below it in the scene hierarchy.

Complete Key-value Options

Key	Type	Default	Comment
head	boolean	n/a	Implicitly only ever used in the “true” state.
enabled	boolean	true	If set to “false”, is intended to act to disable rendering or functionality of any associated **eyes**.

Mirror Plane

Mirror planes define 3D planes in the **head coordinate space** (**HCS**) that can be specified to mirror an **eye** object across. They are composed of a position and orientation, additionally, rather than defining the position and orientation of the plane, a number of pre-defined common mirror planes (such as the YZ or dorsoventral plane) may be referred to by name.

While **mirror planes** are used in relationship to the coordinate system of a **head** object, they may be re-used by many different **eye** objects (in much the same way that a glTF **material** may be re-used by many **meshes**) - in this way, multiple **eyes** can refer to the same mirror plane in relation to their associated **head** object.

Mirror planes are defined in a list stored in the top-level OCES extension object in a list called “mirrorPlanes”, and then referenced by **eye** objects. Here is an example mirror list that defines four mirror planes:

```
Unset
...
"mirrorPlanes" : [
       { "position": [0,0,0], "normal": [0.747, 0.747, 0] },
       { "position": [2,1,0], "normal": [0, 0.747, 0.747] },
       { "type": "DORSOVENTRAL" },
       { "type": "XY" }
]
...
```

Complete Key-value Options

Key	Type	Default	Comment
position	3D vector	[0,0,0]	The position of a point on the mirror plane
normal	3D vector, String	[1,0,0]	The normal direction of the mirror plane. **Must** be normalised if provided as a 3D vector. If provided as a string, must be one of the following values: • X • FRONTAL • LEFT • RIGHT — The X axis in head coordinate space. • Y • TRANSVERSE • DORSOVENTRAL • UP • DOWN — The Y axis in head coordinate space. • Z • SAGITTAL • ANTEROPOSTERIOR • FORWARD • BACK — The Z axis in head coordinate space.

Eye

Eye objects define the actual eye data structure. They are defined in a list within the top-level OCES extension object. All eyes contain the key **values** of *name*, *type* and *enabled*, as well as a list of associated mirror plane indices - *mirrorPlanes*. All eyes also contain a list of **properties** and/or **coarseProperties** which refer to **buffer accessors** that store the relevant property in the relevant format, however the requirements on each change depending on the *type* of eye that is being represented. For this reason, the requirements have been listed out below under three separate categories.

Shared Key-value Options

The following key-value options are defined for all eyes, no matter the data type:

Key	Type	Default	Comment
name	String	“”	Defines the name of this eye dataset.
type	String	n/a	**Must** exist. Defines the type of this eye dataset, and therefore what other data is present. Must be one of: • POINT_OMMATIDIAL • SURFACE

			• SPHERICAL
enabled	Boolean	true	Used as a marker for whether or not to include this in program-specific rendering/visualisation/processing applications.
mirrorPlanes	Integer List	[]	An integer list of (0-indexed) indices of **mirrorPlanes** to apply to this eye dataset. When specified, the data in this dataset will be mirrored along the specified plane, in the attached eye's **head** coordinate space.

Coarse and *Fine* Properties

As explained in "OCES Primitives", sometimes data will not be available to map how a **property** changes for every ommatidium across an eye surface, but it may be available in the form of a single average - for example, the *average* facet diameter may be known, but not *every* facet diameter. In that case, the assumption can be made that the single value be projected to *all* ommatidia. This will mean that the list of values for each ommatidium will all be the same, or that an image for a property distribution will be of one singular value. To save file space and reduce complexity, eyes have two properties list - "properties" whereby varying ommatidial values are stored, and "coarseProperties" which stores only one of each property value, to be broadcast to every ommatidial position. In the case of facet diameter, the two properties definitions are equivalent:

```
Unset
"coarseProperties" : {
      "DIAMETER" : 0.2
},
"properties" : {
      "DIAMETER" : 0
}
```

Where the "DIAMETER" key-value in the "properties" object points to a buffer accessor (with one entry per ommatidium) of scalar values all of 0.2. In this case, the second "DIAMETER" can be removed, eliminating the need for the extra buffer. In the case that both a coarse and fine property of the same name are defined, the coarse duplicate **must** be ignored. In the property lists below, "Type" is taken to mean either a list(in the point-ommatidial case)/N-channel image (in the other two cases) or a single value for the fine and coarse instances of the property. All defaults are coarse.

Point-Ommatidial Data

When the *type* key is "POINT_OMMATIDIAL", the following requirements for *properties* and *coarseProperties* apply:

Property	**Type**	**Default**	**Comment**

POSITION	3D Vector	[0,0,0]	The position of the centre of each ommatidium's lens, in the eye's coordinate space
ORIENTATION	3D Vector	[0,0,1]	The axial heading direction of each ommatidium - i.e., the sampling cone's direction.
DIAMETER	Scalar	1	The diameter of each ommatidium's lens - i.e., the diameter of the sampling cone's ommatidium-side circular cap.
FOCAL_OFFSET	3D Vector, Scalar	-1	The focal offset of each ommatidium in **LOCS** - i.e., the point at which the sampling cone comes to a point behind each ommatidium's lens. If given as a 3D vector, encodes shear about the lens in the order [U,V,W]. If given as a 1D scalar, only the W-axis offset is stored (i.e. the resulting vector is of the form [0,0,W]). In both cases, the W-axis offset **should** be negative.

Surface Properties Data

Unlike the **point-ommatidial** and **spherical** data formats, the **surface properties** format defines a *surface mesh*. The definition of this surface mesh is purposefully similar to the GITF geometry standard, however as only one mesh will ever be defined, it is reduced in its complexity and more specific nomenclature is used. Surface geometry is stored in the "surface" property as a collection of vertex attributes:

```
Unset
"surface" : {
       "POSITION" : I,
       "NORMAL" : J,
       "INDICES" : K,
}
```

Here, as in the POSITION and ORIENTATION properties from the point-ommatidial representation, a set of oriented vertices is defined in 3D space. INDICES then indicates, in triplets of counter-clockwise indices referring to the POSITION and ORIENTATION lists, forming a set of **faces**. If only POSITION, NORMAL and INDICES is defined, then the eye surface properties (the counter-parts to other eye data types, listed below) are assumed to be lists (**buffer accessors**) of length equal to the number of vertices (i.e. the length of POSITION and NORMAL lists).

Instead of using a vertex-attribute approach, properties can be mapped to the surface using a form of UV-mapping. This requires the additional definition of the TEXTURE_COORD and TEXTURE_INDICES:

```
Unset
"surface" : {
        ...
        "TEXTURE_COORD" : N,
        "TEXTURE_INDICES" : M
}
```

TEXTURE_COORD references a buffer accessor that describes an arbitrarily-long set of 2D points to perform UV mapping with (the *unwrapped surface mesh*), and TEXTURE_INDICES references a buffer accessor that is the same length as INDICES specifying, for each of the counter-clockwise triangles in the surface mesh, an index within the TEXTURE_COORD, mapping the association between 3D **face** vertices and their corresponding 2D texture vertices. With both of these values defined, fine property data must be defined as an N-channel Image which utilises the UV-map to place data. Both of these values **must** be defined to use UV-mapped property data.

When the *type* key is SURFACE, the following requirements for *properties* and *coarseProperties* apply:

Property	**Type**	**Default**	**Comment**
DIAMETER	Scalar, 1-channel Image	1	The diameter of each ommatidium's lens - i.e., the diameter of the sampling cone's ommatidium-side circular cap.
FOCAL_OFFSET	3D Vector, Scalar, 3-channel Image, 1-channel Image	-1	The focal offset of each ommatidium in **LOCS** - i.e., the point at which the sampling cone comes to a point behind each ommatidium's lens. If given as a 3D vector, encodes shear about the lens in the order [U,V,W]. If given as a 1D scalar, only the W-axis offset is stored (i.e. the resulting vector is of the form [0,0,W]). In both cases, the W-axis offset **should** be negative.
DISPLACEMENT	Scalar, 3D Vector, 1-channel Image, 3-channel Image	0	Defines a 1- or 3-channel displacement (in **LOCS**) map over the mesh surface of the compound eye, functioning much as regular displacement maps in computer graphics are used. If only a 1-channel map is provided, then it is interpreted as only a W offset (as a height map).
RELATIVE_ORIENTATION	3D Vector, 3-channel Image	n/a	Augments the orientation of the ommatidia from their default of being normal to the surface they are on. RELATIVE_ORIENTATION defines 3D vectors for each ommatidium that are then added to the default orientation, with the results normalised to obtain the final ommatidial orientation.
ABSOLUTE_ORIENTATION			

			ABSOLUTE_ORIENTATION defines a new set of ommatidial directions that completely overwrites the default orientations. Only one **should** be defined, but if both are defined, then consideration **must** be limited to only the ABSOLUTE_ORIENTATION. Changing the “NORMAL” value of the “surface” object should be preferred over using these properties.

Additionally, spherical properties data allows for these additional key-value properties:

Key	**Type**	**Default**	**Comment**
ommatidialCount	Integer	1	The number of ommatidia on this eye

Spherical Properties Data

When the *type* key is “SPHERICAL”, the following requirements for *properties* and *coarseProperties* apply:

Property	**Type**	**Default**	**Comment**
OMMATIDIAL_BOUNDS	1-channel Image	1	Specifies the extent to which ommatidia can be placed. Essentially a mask showing the maximum bounds of the eye, in spherical structure.
DISPLACEMENT	1-channel Image, 3-channel Image	0	Defines a 1- or 3-channel displacement (in **LOCS**) map over the surface of the spherical compound eye, functioning much as regular displacement maps in computer graphics are used. If only a 1-channel map is provided, then it is interpreted as only a W offset (as a height map).
RELATIVE_ORIENTATION	3-channel Image	n/a	Augments the orientation of the ommatidia from their default of being normal to the sphere they are on. RELATIVE_ORIENTATION defines 3D vectors for each ommatidium that are then added to the default orientation, with the results normalised to obtain the final ommatidial orientation. ABSOLUTE_ORIENTATION defines a new set of ommatidial directions that completely overwrites the default orientations. Only one **should** be defined, but if both are defined, then consideration **must** be
ABSOLUTE_ORIENTATION			

			limited to only the ABSOLUTE_ORIENTATION

Additionally, spherical properties data allows for these additional key-value properties:

Key	Type	Default	Comment
radius	scalar	1.0	The radius of the “sphere” that the data sits on.
ommatidialCount	Integer	1	The number of ommatidia on this eye

Example Eyes

Here are some example eyes:
TODO

File Level

At the file level (i.e. within the “OCES_eyes” object within the file’s root “extensions” object), there are a number of key-value pairs that can be defined:

Key	Type	Comment
generator	Object: {“program”: String, “Version”: String}	**Should** be provided. A simple object consisting of “program” and “version” keys, which **must** as their values specify a program name and version. E.g: Unset `"generator" : {` `  "program" : "Eye Editor",` `  "version" : "1.0.0"` `}`
creationDatetime	String	**Should** be provided. A single string showing date-time creation of the file. **Must** follow ISO 8601 (YYYY-MM-DD hh:mm:ss.ms, full specification recommended, **should** extend to at least minute resolution).
maximumRenderDistance	Scalar	**Should** be provided. By describing the per-ommatidial sampling frustum and close (to the ommatidial side) cap, the near rendering plane is implicitly defined. However, the far rendering plane is undefined. This property defines the maximum render distance. Without it, rendering engines are free to choose their own maximum rendering distance.
extensionsUsed	[String]	Mirrors the GITF extension system, this and the next

		two keys define in-file-use extensions of this extension. A list of strings identifying sub-extensions that this OCES file uses. A list of sub-extension standards and their implementation details can be found on the OCES github page. By listing them here, renderers and parsers can expect and look for sub-extension specific data. See “Sub-Extensions: Defining Additional Data Standards” under “Schema and Reference Implementations” below for more information.
extensionsRequired	[String]	See “extensionsUsed” key. Defines by name any extensions that are *required* in order for the file to be used successfully.
extensions	Custom Object	See “extensionsUsed” key. Defines all custom sub-extension data, in much the same way all root-level OCES data is stored in an OCES_eyes object within the root GlTF file’s “extensions” object.

Additional Implementation Reference Information

This section outlines the information required to

Calculating DIAMETER and FOCAL_OFFSET

The DIAMETER and FOCAL_OFFSET properties were chosen to give the most accurate and generic description of the visual receptive field of an ommatidium, as they are the minimum required values to reconstruct the sewed cone. However, they can be unintuitive to calculate. Below are instructions on converting commonly recorded ommatidial properties (such as acceptance angle and facet diameter) into the DIAMETER, FOCAL_OFFSET model.

From (Acceptance) Angle and (Facet) Diameter

// Instead of "DIAMETER", "ANGLE" can be specified, which assumes that the focal offset is on the ommatidial axis. ("FOCAL_OFFSET" then has a z component configured such th at the diameter matches recorded data
165 // Equally, instead of "FOCAL_OFFSET", just "DIAMETER" and "ANGLE" can be specified to calculate the FOCAL_OFFSET, again assuming an xy skew of 0.
166 /// TODO: This shouldn't be in here. It should be under the Add. Implementation References and should be described there how to get to this format.

Inter-primitive Conversion Methods

As mentioned above, in order for all

Point-Ommatidial to Surface Properties Model

Point-Ommatidial to Spherical Property Field Collection

Surface Properties Model to Spherical Property Field Collection

Surface Properties Model to Point-Ommatidial

Spherical Property Field Collection to Point-Ommatidial

Spherical Property Field Collection to Surface Properties Model

Schema and Reference Implementations

Access

The JSON schema describing the Open Compound Eye Standard as a glTF extension can be found at https://github.com/Blayzeing/open-compound-eye-standard. A reference Python/JS/C++ library can be found here<NOT HERE YET>. We also provide a tool for viewing, converting and modifying OCES-augmented glTF files here<ALSO NOT HERE YET>.

Parser Requirements

Here we list rules that compliant parsers **must** adhere to.

- Parsers **should** verbosely check that the incoming file contains all the required information the parser wants. For instance, if a parser is written for a system that requires polarisation angle as an ommatidial property, it should first check that the file contains this and report back to the user if this information is missing.
- Parsers **must** parse mandatory minimum ommatidial properties.
    - Parsers **must** emit a warning if an eye data object lacks one or more mandatory minimum ommatidial property.
        - Renderers **must** not load that eye into the scene graph/scene representation being used.
- Parsers **must not** manipulate additional ommatidial properties. For instance, if a parser is written for a system that does not support axial U/V offsets, and they are specified in an OCES-glTF file then the parser should not remove them or set them to some other value.
- When presented with a choice between a coarse *property* and a *property* (i.e. a coarse property is defined, but so is a property of the same name), renderers **must** ignore the coarse property, choosing the more detailed regular property.

Sub-Extensions: Defining Additional Data Standards

TODO: Extension Examples:

- Ommatidial shape
    - Polygon in LOCS-UV defining each ommatidium's shape
- Knockout experimental control
    - Additional enabled property allowing for control over which ommatidia are enabled
- Property colourmaps
    - For visualisers, an additional list that defines arbitrary colour maps for both single-channel data (in the form of a list of *{scalar value, colour}* pairs) or 3-channel data (in the form of 3 lists of {channel-specific scalar value, channel-specific color value} pairs) (might need to iron out the details of this one).
- Sky rendering
    - Could be an image

    - Could be a shader
    - Could be a vertical colour gradient (with interpolation settings)

Reference Implementation Notes

Developer Notes

- Note about leveraging 3D software (say, Blender)'s ability to store "extras" data in GlTF files to allow for input configuration of a GlTF file to "pre-process" it before being passed through some third-party program (for instance, setting an "empty" node to have an extras property of "OCES: head" before uploading to the web editor.

Appendix

A.

A simple glTF file that stores a single triangle in 3D space, with vertex-colouring attributes. Note that the single “triangle” mesh references 4 accessors to store POSITION, NORMAL, TEXCOORD_0 and COLOR_0 attributes. Also note that rendering choices are handled by the renderer - here while TEXXOORD_0 is specified, we actually render using COLOR_0.

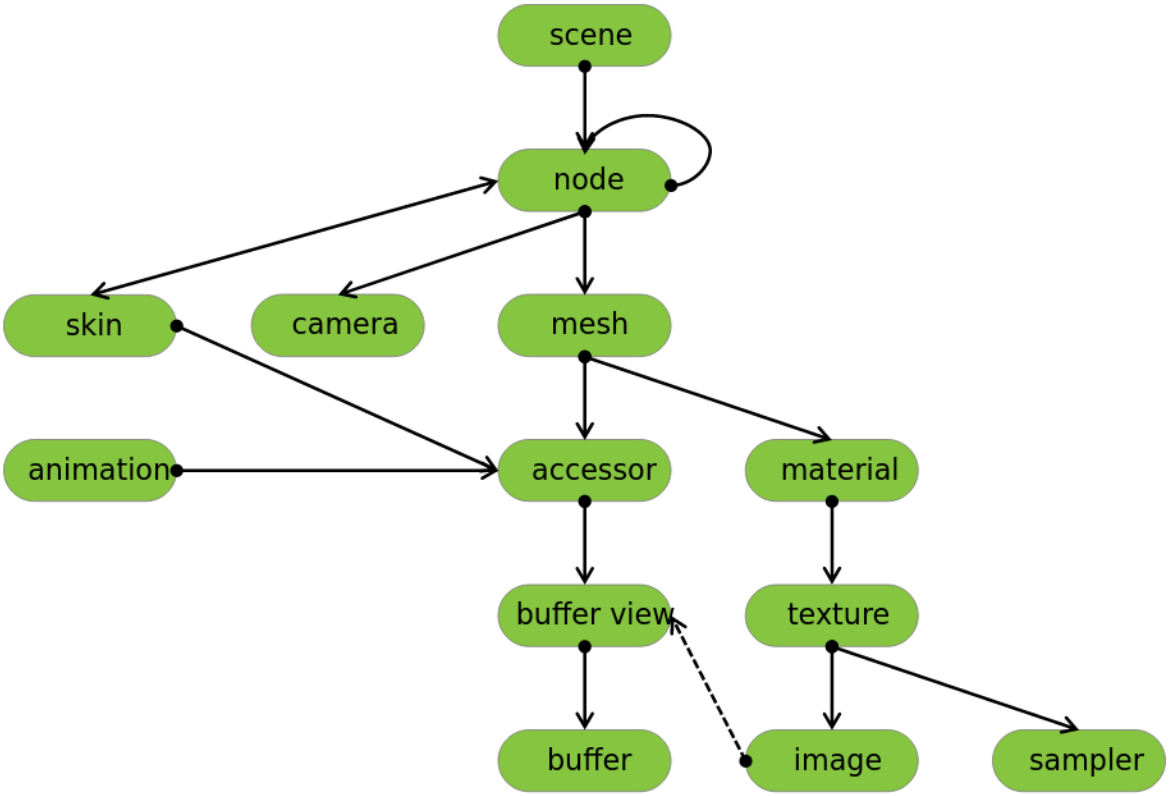
```
Unset
{
        "scene" : 0,
        "scenes" : [
        {
                "name" : "Scene",
                "nodes" : [
                0
                ]
        }
        ],
        "nodes" : [
        {
                "mesh" : 0,
                "name" : "triangle",
                "translation" : [
                0,
                0,
                1
                ]
        }
        ],
        "materials" : [
        {
                "doubleSided" : true,
                "name" : "Simple Material",
                "pbrMetallicRoughness" : {
                "metallicFactor" : 0,
                "roughnessFactor" : 0.5
                }
        }
        ],
        "meshes" : [
        {
                "name" : "triangle",
                "primitives" : [
                {
                        "attributes" : {
                        "POSITION" : 0,
                        "NORMAL" : 1,
                        "TEXCOORD_0" : 2,
                        "COLOR_0" : 3
```

```
                },
                "indices" : 4,
                "material" : 0
        }
        ]
}
],
"accessors" : [
{
        "bufferView" : 0,
        "componentType" : 5126,
        "count" : 3,
        "max" : [
        1,
        0,
        1
        ],
        "min" : [
        -1,
        0,
        -1
        ],
        "type" : "VEC3"
},
{
        "bufferView" : 1,
        "componentType" : 5126,
        "count" : 3,
        "type" : "VEC3"
},
{
        "bufferView" : 2,
        "componentType" : 5126,
        "count" : 3,
        "type" : "VEC2"
},
{
        "bufferView" : 3,
        "componentType" : 5123,
        "count" : 3,
        "normalized" : true,
        "type" : "VEC4"
},
{
        "bufferView" : 4,
        "componentType" : 5123,
        "count" : 3,
        "type" : "SCALAR"
}
],
"bufferViews" : [
{
        "buffer" : 0,
        "byteLength" : 36,
```

```
                "byteOffset" : 0,
                "target" : 34962
        },
        {
                "buffer" : 0,
                "byteLength" : 36,
                "byteOffset" : 36,
                "target" : 34962
        },
        {
                "buffer" : 0,
                "byteLength" : 24,
                "byteOffset" : 72,
                "target" : 34962
        },
        {
                "buffer" : 0,
                "byteLength" : 24,
                "byteOffset" : 96,
                "target" : 34962
        },
        {
                "buffer" : 0,
                "byteLength" : 6,
                "byteOffset" : 120,
                "target" : 34963
        }
        ],
        "buffers" : [
        {
                "byteLength" : 128,
                "uri" :
"data:application/octet-stream;base64,AACAvwAAAAAAIA/AACAPwAAAAAAIC/AACAvwAAAAAAIC/
AAAAAAAAgD8AAACAAAAAAAAAgD8AAACAAAAAAAAAgD8AAACAAAAAAAAAgD8AAIA/AAAAAAAAAAAAAAAA//8AAA
AA//8AAAAA/////wAA//8AAP//AAABAAIAAAA="
        }
        ]
}
```


B.

The glTF Object Hierarchy, taken from the [glTF 2.0 file format specification document](). Note that the scene graph of each individual scene (of which a single glTF file can hold many) is represented by the one-to-many self-directed connection on the **node** object.



C.

An example OCES-enabled GITF file. Note that this file is not valid JSON as JSON does not support comments using the double-slash and triple-dot notation seen here. Also note that the file omits non-OCES data types:

```
Unset
{
        "asset" : {
        "generator" : "Khronos glTF Blender I/O v3.3.27",
        "version" : "2.0"
        },
        "extensionsRequired":[
        ],
        "extensionsUsed": [
        "OCES_eyes"
        ],
        "scene" : 0,
        "scenes" : [
        {
        "name" : "Scene",
        "nodes" : [
        0
        ]
```

```
        }
        ],
        "nodes" : [
        {
        "name" : "example-head-1",
        "translation" : [
        5,
        5,
        5
        ]
        "extensions" : {
        "OCES_eyes" : {
               "head" : true // Indicates that this is a head object
        }
        },
        "children" : [2,3] // Has two children, one is an eye, one is an eye wrapped
in a node
        },
        {
        "name" : "single-omm-eye",
        "extensions" : {
        "OCES_eyes" : {
        "eye" : 0, // Refers to an eye object from the OCES_eyes eyes list
        "mirrorPlanes" : [0,2] // Refers to mirror planes in the OCES_eyes mirro
planes list, which are assumed to be in the coordinate space of the head this eye is a
part of
        }
        }
        },
        {
        "name" : "arbitrary-transform",
        "rotation" : [ 0.5, 0.2, 1.0, 0.4],
        "translation" : [0.3, 0.1, 0.0],
        "scale" : [0.1,0.1,0.1],
        "children" : [2]
        }
        ],
        "meshes" : [...]
        }
        ],
        "accessors" : [...],
        "bufferViews" : [...],
        "buffers" : [...],
        "extensions": {
        "OCES_eyes": {
        "version" : "1.0", // Which verision of the OCES standard is being adhered to
```

```
	//"unitScale" : 0.001, // [Optional, defaults to 1] Either defines the unit scale of all eyes in this file, relative to 1 metre. Or takes a metric/imperial named identifier, which is then interpreted as a scaling factor
	//		//		e.g. "metre", "meter", "millimetre", "micrometre", 0.01 (cms), "inch", "foot", "yard", "thou"/"mil", "m", "dm", "cm", "mm", "um", "pm"
	// Heads are just on nodes
	"eyes": [
	// Eyes have a type that specifies which of the subtypes of eyes it is ("pointOmmatidial", "surface", "spherical")

	// Here's a point-ommatidial eye data
	{
	// On every eye data
	"name" : "example-point-ommatidial-data",
	"type" : "pointOmmatidial",
	"enabled" : true/false, // Defines whether or not this eye is active (if it is not active, it is skipped. This is useful when you have a renderer that only cares about one type of eye

	// Minimum required for full eye:
	// POSITION, ORIENTATION, DIAMETER, FOCAL_OFFSET

	// Stores properties that are singularly defined - i.e., they're averages. These *must* be over-written if the same propertiy is found in "properties". If that happens a viewer *should* warn the user.
	// "ORIENTATION" could be stored here, but that'd kind of be stupid.
	"coarseProperties" : {
		"DIAMETER" : f, // floating-point number describing the average facet diameter
		"FOCAL_OFFSET" : // Either a 3-vec representing all ommatidial focal offsets (a bit pointless), or a singular floating-point number (<=0) showing the inset on the z-axis
	},

	// Stores properties that change on a per-ommatidial basis
	"properties" : {
		"POSITION" : N,	// 3-vec, describes the location of the center of the near-plane/ommatidial lens
		"ORIENTATION" : N, // 3-vec, descirbes the orientation of that plane, as a unit vector in the direction of the ommatidial axis
		"DIAMETER" : N,	// 1-vec, describes the diameter of the oriented view frustum at the near-plane/ommatidial lens (facet diameter)
		"FOCAL_OFFSET" : N // 3-vec, describes the tip of the oriented view frustum below its lens (Z component should always be <=0), in Local Ommatidial Coordinate Space (LOCS)
			// 1-vec, can be a 1-vec to assume that the x and y coordinates are 0, and we only care about the LOCS z-axis
```

```
        }
        }

        // Here's a spherical eye data
        {
        // On every eye data
        "name" : "example-spherical-eye-data",
        "type" : "spherical",
        "active" : true/false",

        // Specific to this eye
        "radius" : 1.0, // The base-line radius of this eye by which every property
is relative to. Defaults to "1.0".

        // Minimum required for full eye:
        // DIAMETER, FOCAL_OFFSET
        // [Position is generated from the sphere itself, and Orientation is formed
as it's normal by default (but this can be augmented or over-written)]

        "coarseProperties" : {
        },
        "properties" : {
                "DIAMETER" : N,       // 1-c image, describes the diameter
distribution of the ommatidia
                "FOCAL_OFFSET" : N,          // 1- or 3-c image, describes, in the
LOCS, the displacement of the top of the oriented view frustum below its lens (Z
component <= 0). If 1-c, only z-axis.
                "DISPLACEMENT" : N,          // 1- or 3-c image, describes, in the
LOCS, the displacement of the oriented view frustum from the surface of the sphere.
If 1-c, only describes z-axis.
                "ABSOLUTE_ORIENTATION" : N, // 3-c image describing the absolute (in
Head Coordinate Space (HCS)) orientation of the ommatidia, as per "ORIENTATION" in
the point-ommatidial desc. Use with care, as you can end up specifying eyes that are
inside-out. Unit vector.
                "RELATIVE_ORIENTATION" : N, // 3-c image describing the relative
orientation (in LOCS) of the ommatidia. Summed (and normalized) with the spherical
normal to form the final orientation
        }
        }
        ],
        "mirrorPlanes": [
        // MirrorPlanes have a type - "generic" (default) or a specific indicator
(DOROSVENTRAL, XY, YZ etc).
        // If the type is "generic", then "position" and "normal" vectors must be
supplied.
        // specified indicators specify pre-defined planes (XY, YZ etc are all in
local coords, DORSOVENTRAL etc are all just aliases to their respective XY, YZ etc
planes - as such, they make assumptions about how the coordinate system is used.)
```

```
        { "position" : [0,0,0], "normal" : [0.747, 0.747, 0] },
        { "position" : [2,1,0], "normal" : [0, 0.747, 0.747] },
        { "type" : "DORSOVENTRAL" } //
        ],

        // As with this OCES_eyes extension itself, lists extensions that are
explicitly designed to work with the OCES_eyes ecosystem
        // Lists which extensions are provided by this file - should align with
official extensions list
        //      Note that an extension can be as simple as a requirement for a
particular property on used eyes. The statement of a particular extension here is
more like a promise that the file is adhering to that standard and providing the
data it'll want
        "extensionsRequired" : [
        ],
        "extensionsUsed" : [
        "someExtension"
        ],
        "extensions" : {
        "someExtension" : { "version" : "0.1" }
        }
        }
        }
}
```