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

File Version 0.4.0 [2023.11.01]

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

1. Introduction

Many ways of measuring the structure and receptive features of compound eyes have been used, ranging from simple spherical measurement of acceptance angles or ommatidial density, all the way to complete mappings of ommatidia 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 representation standard 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 (OCES) built on top of the glTF standard for storing 3D scene-graphs. 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. Moreover, the standard allows 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. The result is a standard that allows for the description and storage of compound eye models, as well as (optionally) 3D environments that make up complete experimental configurations.

This document outlines the following:

1. The scope of the OCES standard at a conceptual level (section: “OCES Primitives”)
2. The technical information describing the GlTF-based file format (section: “GlTF 2.0 Extension Specification”)
3. 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.

2. Glossary of Technical Terms

Geometric Algebra	
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, LOCS, ECS, HCS	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.
face	2D and 3D objects can be described as being constructed from a number of faces, which for the purposes of this document will always be 3-sided. A face can be described as a counter-clockwise collection of 3 points in space.
mesh	The name for a set of *3D faces* that describes a 3D object.
frustum, Frusta	A frustum (plural “frusta”), is a prism where one cap is a different size than the other, forming a segment of a cone. This document largely refers to conical frusta.
shear	Often confused with (and sometimes known as) “skew”, a shear transform is an affine transformation that “slants” an object about the origin along an arbitrarily-defined axis.
axis	A line in 3D space around which other points can be defined.
Biological	
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.
facet, ommatidial lens	The lens at the top of the lens stack at the top of the ommatidium, commonly seen on the surface of a compound

	eye, though in this context can also encompass the lens of a single ocelli. Indicated in pink in **Figure 1**.
ommatidial axis	The axis along which an ommatidium is pointed - can be conceptualised as similar to the direction in which a telescope is pointed being referred to as a “telescopic axis”.
ommatidial sampling function	The amount of light that is accepted into an individual ommatidium follows a specific sampling function in relation to the angle between the incoming light and the ommatidial axis, resulting in less light being detected comparatively at the edge of the ommatidial viewing cone compared to at the centre. This function can be approximated in angular space as a Gaussian, with a full width at half maximum equal to the ommatidium’s acceptance angle.
ommatidial property	Some property that changes on an inter-ommatidial basis, such as the ommatidium’s facet diameter, or its receptivity to certain wavelengths of light.
Data Storage	
JSON	JSON (JavaScript Object Notation) is a standard for defining data via sets of key-value associations and basic primitives such as numbers, lists, and constant values.
JSON Object	A JSON object is a structured set of keyword properties and constraints on them, written in the JSON format. Keyword properties can store a JSON object itself, meaning that JSON Objects can contain further nested JSON objects.
“keyword Property”, “key-value”, “key”	A keyword property is a JSON property that has a defined keyword (or “key-value”, or simply “key”) and an assigned/stored value.
glTF file	A glTF (graphics library Transmission Format) file.
glTF-OCES file, OCES-augmented glTF file	A glTF file that has had additional data added to it that is compliant with the Open Compound Eye Standard.
data consistency, data inconsistency	When one piece of data is stored in two places at once, this data must remain consistent - i.e. it must be the same at both locations, otherwise ambiguity arises as to which copy is the correct version. This notion is referred to as data consistency, or data inconsistency.
artifacting	In the context of this document, an “artifact” or “artifacting” refers to digital artifacting, whereby the process of storing or manipulating data in a certain way results in an unintentional change to that data’s structure, often presenting as noise or inconsistencies in structure and/or meaning.
Computer Graphics	

“unwrapped mesh”, “UV map”	A set of 2D *faces* that correspond to the 3D faces in a given *mesh*. This can then be used to associate a 2D image with the surface on a 3D object.
texture	A 2D image that stores some property (usually colour, but other examples include glossiness, transparency or roughness) that changes over the surface of a 3D object.
vertex attribute	A value associated with a vertex of a 3D or 2D object - such as the colour or normal. Used to describe information about a 3D object beyond that implicitly described through the mesh/shape of the object. Commonly stored in a list of *vertex attributes* which can then be taken alongside a list of points on the object.
colour mapping/shading	A method for defining the surface colour of a 3D or 2D object by defining the colour at every point on the object (in the case of a *Vertex attribute defined 3D mesh*) or as a 2D texture that defines the colours at any given point across each surface (usually in combination with a UV map), and then interpolating the colour across the surface of each *face*.
bump mapping, height mapping, displacement mapping	Similar to colour mapping/shading, “bump maps” and “displacement maps” are textures defining a change in the surface of a 3D object.
environment map, reflection map	A plot of data (usually light measurements, such as those found in a 360-degree image) defined across a spherical surface, for 3D-directional lookup. These are usually used to store information about a virtual environment, such as the sky colour at any given 3D angle, or the reflections present in a given area. For this reason they are often referred to as “environment maps” and “reflection maps”.
scene graph	A scene-graph is a graph representation of a 3D or 2D scene - in particular it is composed of nodes that describe a transform relative to their parent nodes. In this way, a tree of nodes constructs a final transform for an element within the scene. They are commonly used to represent 3D and 2D environments in real-time graphics applications such as video games and interactive user menus.
point-cloud	A collection of points in 3D space.

3. Recommended Reading

While this document is intended to provide a comprehensive overview of the OCES standard, and will provide brief overviews of each subject and concept introduced, it is not designed to explain all relevant concepts and motivations. It is written assuming the reader is familiar with the anatomical structure of the compound eye, compound eye research in general, 3D geometrical concepts, and the basics of computer graphics/3D data storage technologies and representations. Good resources for these fields can be found in the following books:

- “Animal Eyes”, 3rd edition, chapter <BLANK>, by Michael F. Land and Dan-Eric Nilsson
- “Linear and Geometric Algebra”, by Alan Macdonald
- A good introductory book on computer graphics

Further, to fully understand the associated sections “glTF 2.0 OCES Extension Specification” and “Additional Implementation Reference Information” additional familiarity with standards specification formats, the glTF 2.0 standard, data structures and data storage methodologies are recommended. Good resources for these fields can be found in the following materials:

- “Introduction to Algorithms”, 3rd edition, chapters 10 and 33, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein
- The official glTF 2.0 specification (https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html)
- The glTF 2.0 tutorial series (https://github.khronos.org/glTF-Tutorials/gltfTutorial/)

Where possible, links to wikipedia and relevant authoritative body pages are provided for more specific technical information.

4. Open Compound Eye Standard Primitives

The Open Compound Eye Standard (OCES) is defined as an extension to the Graphics Library Transmission Format (glTF), which is a well-established format for storing and describing 3D scenes in a compact, semi-human-readable way using JSON objects. While the standard is built on glTF, this section aims to explain the core concepts and structures with little reference to the technical aspects of the format.

There are four primary primitives defined in the OCES:

- Three primitives which can represent the ommatidial structure of an eye itself
- A final “head” primitive that 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 technical descriptions of coordinate spaces and data concepts (subsection: “Minimum Requirements and Objects”). Finally, requirements for consistency and standard cohesiveness are discussed (subsection: “Ensuring Data Cohesiveness”).

4.1. Compound Eye Primitives Overview

4.1.1. Conceptual Overview

Figure 1. Abstracting from a real compound eye. Left: A real compound eye *(image credit Matt barber)*. Middle: A segmented diagram showing a set of ommatidia’s photoreceptor area (orange), its lensing and apparatus (pink), acceptance cone (light blue) and its acceptance function (red). Right: A high-level abstract representation of only the acceptance function cones in 3D space. The OCES standard uses these to describe the light that can be captured by any given ommatidium.

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 light acceptance/sampling region - i.e. the region of space that light is sample from for each ommatidium (referred to as the ommatidium’s **oriented sampling frustum**). In this document, we make reference to approximating this function as a Gaussian distribution along the axial spread of each sampling frustum, as is found in compound vision rendering literature (see Appendix D), however this function could be of any form and is not specified in the standard. Alongside these basic features, additional **ommatidial 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 omitted.

To summarise, a single **eye** can be defined as a set of points in 3D space, along with a list of corresponding **ommatidial properties** - at a minimum these properties must define the oriented sampling frusta at each point, but additional data could be included alongside them. For example, data collected about an eye with 5 ommatidial properties could look like this:

Position	Orientation	Acceptance Angle	Lens Diameter	Spectral Sensitivity
(0.7, -0.4, 0.2)	(-0.6, 0.7, -0.8)	3.23	2.349	498
(-0.9, 0.3, -0.6)	(0.4, -0.5, 0.3)	2.85	3.126	480-520
(0.1, 0.8, -0.5)	(-0.2, 0.9, -0.7)	1.38	2.874	498
(-0.3, -0.1, 0.9)	(-0.3, 0.5, 0.2)	2.90	3.215	490-505

Table 1. A set of ommatidial properties for an example eye with 4 ommatidia.

In this eye the *position*, *orientation*, *acceptance angle*, and *lens diameter* can be used to compose an oriented sampling frustum for each ommatidium. Additionally, each ommatidium has an additional “spectral sensitivity” property defined. Note that these ommatidial properties can be thought of as “images” that wrap over the surface of the eye, describing how the property changes either exactly at each ommatidium or in general over the population of ommatidia.

As an **ommatidial property** is a property (or variable) that affects individual ommatidia and changes across the surface of the eye, even the position of the ommatidia within the eye can be seen as an ommatidial property - in this way, **Table 1** can be interpreted as an eye consisting of 4 ommatidial properties. Ommatidial properties can be thought of as “layers” that build up over an eye. The “base layers” are those required to define the individual view frusta of all ommatidia (position, orientation, facet diameter etc.), while additional “layers” containing more information (polarisation sensitivity, facet shape, colour sensitivity curves etc.) that might be necessary only to one study or dataset can be added alongside these “base layers”.

4.1.2. Eye Representations

The open compound eye standard identifies and supports three separate ways that the data required to describe these property-attached view frusta is being and could be represented using three varying levels of detail, from high fidelity to low:

1. **Point-ommatidial representation (highest fidelity):** As described in **Table 1**, where the oriented view frusta and additional properties are stored directly on a per-ommatidial basis.

2. **Surface properties representation (mid-level fidelity):** Instead of directly storing the points of all ommatidia, a 3D surface is stored and used to describe the distribution of ommatidial properties in 3D space. This surface implicitly describes the position and orientation (the surface normal) of any ommatidia that could be placed onto it. Ommatidial properties are stored as textures “wrapped” around this surface, describing their distribution over the surface.
3. **Spherical properties representation (lowest fidelity):** Similar to the *surface properties representation*, only assuming a spherical surface. In this case, only the ommatidial properties are stored, to be mapped around a spherical surface, as seen predominantly in older works often in the form of, for example, acceptance angle plots.

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. **Figure 2** below shows how similar ommatidia can be represented in each representation.

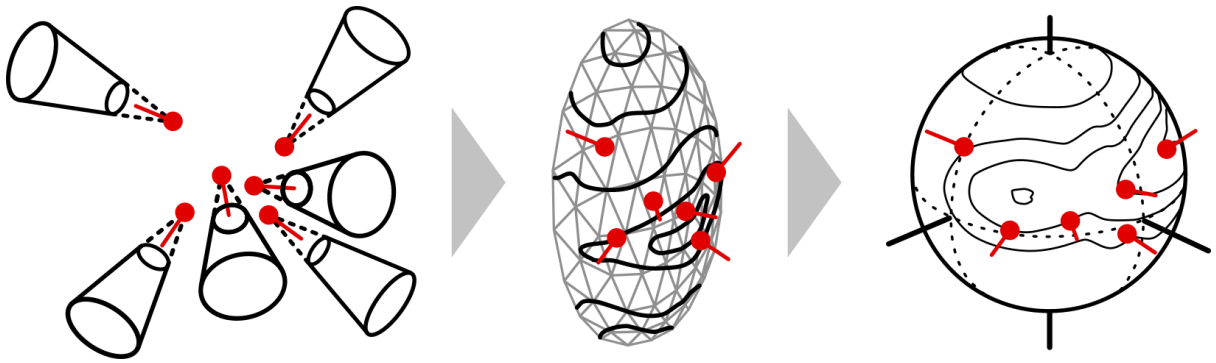


Figure 2. The same six ommatidia (red) as represented in each representation, from highest (left, **point-ommatidial**) to lowest (right, **spherical**) fidelity. Note that only the point-ommatidial representation stores the specific position and orientation of the ommatidia, while the other representations only store distributions of these values, leaving the specific ommatidia to be generated in accordance with these distributions. Conversions are possible between each representation, however it is a lossy process in the high-to-low direction.

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.

4.1.3. Multi-Eyed Samples

The above text presents compound eyes as a singular unit - i.e. defining an eye as a single collection of ommatidia. However, in practice eyes do not exist in a vacuum - often users will want to describe two or more eyes as part of an entire head-based invertebrate visual system, perhaps even including additional ocelli groups. 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 acts as a root-object for compound eye primitives to be described relative to, defining a common coordinate space (the **Head Coordinate Space**). This common coordinate space allows for common simple manipulation tasks - such as mirroring eye primitives along a given head axis to produce left/right eye pairs, and allowing for head-level transforms such as scaling, rotation and translation to be applied. By using a single shared coordinate space and defining scaling/mirroring operations within it, individual ommatidial data does not need to be duplicated or directly altered, which would otherwise be prone to data inconsistencies between the mirrored pairs, or other artifacting resulting from repeatedly manipulating the data.

4.1.4. Non-Eye Data Storage: Experimental Configurations

While this document is primarily interested in explaining the concept of storing compound eye structural data, it should be noted that this is not the extent of the OCES-glTF file standard described herein. As the OCES standard is simply an extension to the glTF 3D graphics file format, eyes can be stored alongside entire 3D environments that comply to the glTF file specification (which itself supports animation, mesh deforming, light, camera and scene information, with further data optionally storable using other extensions).

It is envisioned that there will be two primary uses for the OCES-glTF file format:

- Storing individual eyes, or collections of them, for comparative and distribution purposes
- Storing entire experimental configurations complete with 3D environments, eyes and animations

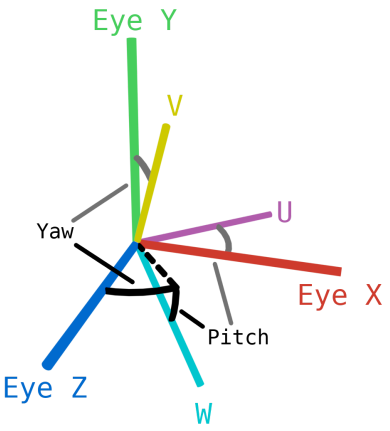
In this way, it is important to remember that a single OCES-glTF file may represent either use-case and that **eyes** and **heads** may be defined as sub-objects within a wider 3D scene. Further, it is expected that users will want to transplant eyes from eye-only files into files containing 3D environments and vice-versa.

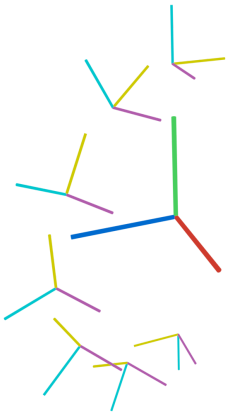
4.2. Minimum Requirements and Specific Primitive Definitions

The previous sections have discussed the data required to represent and manipulate compound eyes in broad conceptual strokes, in particular 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”). This section provides a more in-depth description of the supporting concepts and ideas.

4.2.1. Coordinate Spaces

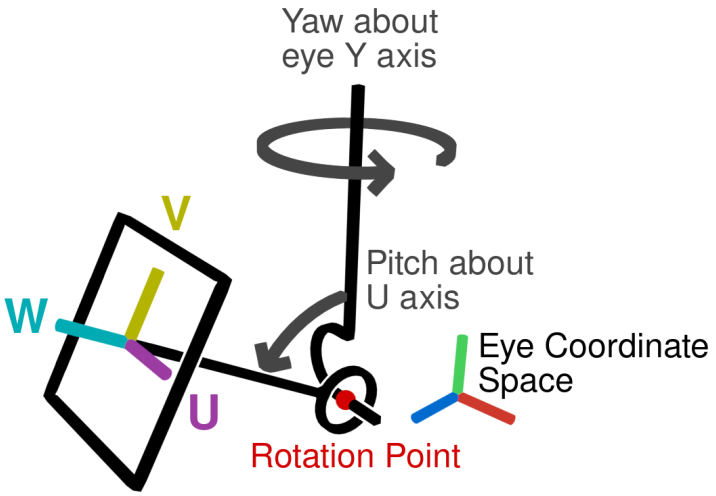
A number of ommatidial properties require a 3D coordinate space relative to each ommatidium - as any property that can be described spatially as not directly on the ommatidial axis requires a coordinate system that can be reliably reconstructed for each ommatidium. While the head- and eye-level coordinate 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 form 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 a yaw singularity 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.

In the completely upwards and downwards direction, the coordinate system becomes unconstrained, producing a singularity about the yaw axis where the same heading direction (directly up or down, when the W component is either *[0,1,0]* or *[0,-1,0]*) can be described by any number of yaw values. As stated, in these cases, the U axis should be considered parallel to the X axis (in the eye's coordinate space), resulting in two "polar" coordinate systems at the upper-most and lower-most decrees of rotation:

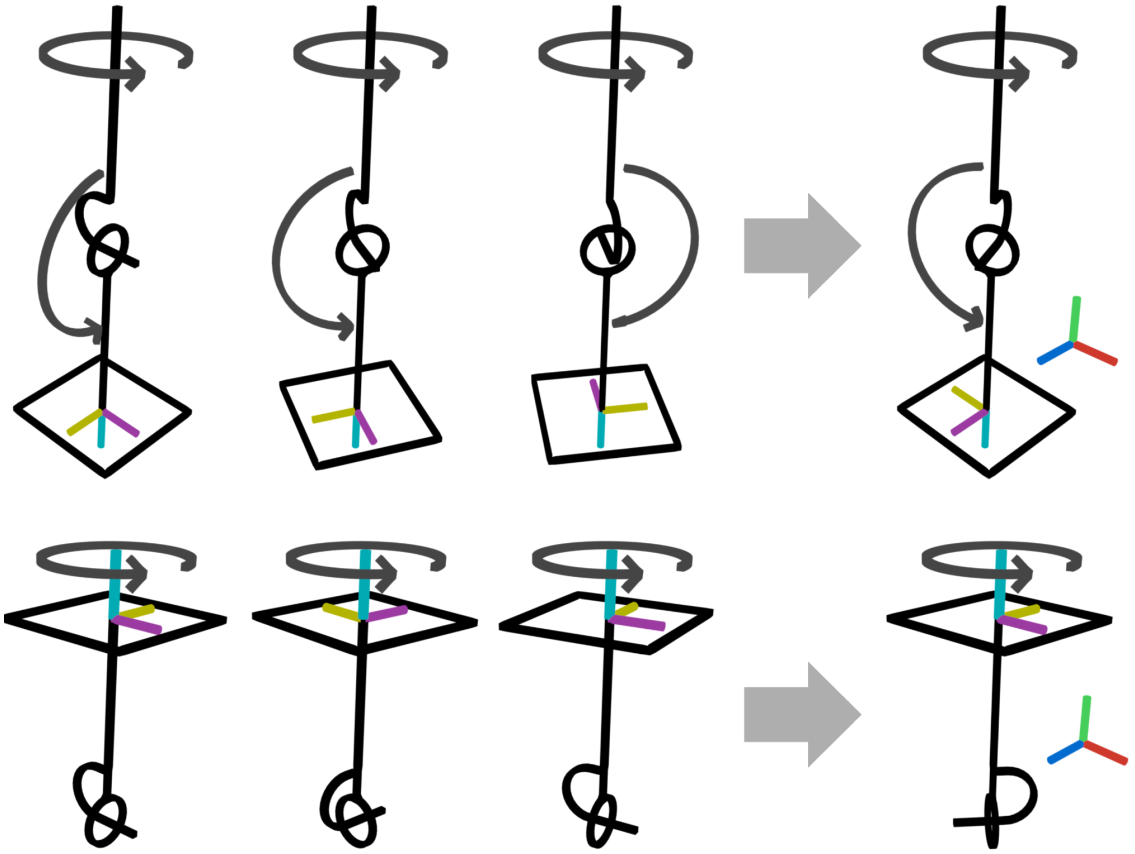


Figure N. When an ommatidium is oriented either fully up or fully down, any number of yaw values will describe the same heading (left). To ensure that all implementations are consistent with each other, in these situations the yaw component of an orientation should be ignored, and the U axis aligned with the eye's X axis (right).

4.2.2. Eye Data Representations

Representation 1: Point-Ommatidial Representation

The point-ommatidial representation consists of the minimum required constraints to define each oriented, sheared frustum in **LOC** space that represents the conical spread of each ommatidium's sampling function into the environment.

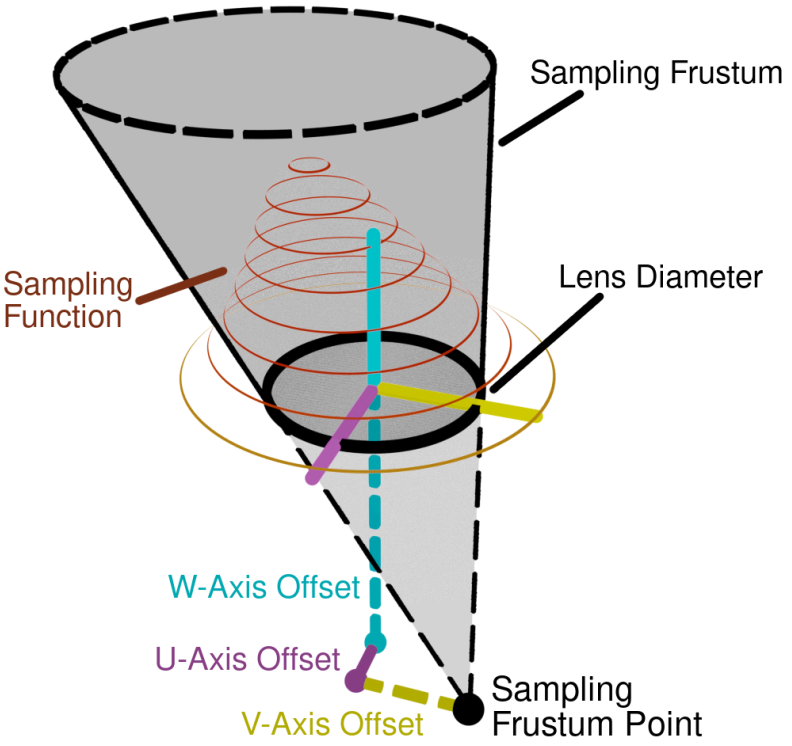


Figure N: The sampling function of an ommatidium described in relation to an oriented, sheared frustum. Note that here the sampling function used is a Gaussian distribution with respect to its polar position within the frustum (as per rendering literature, see appendix D), however in practice the sampling function can be any that is derived from the sampling frustum.

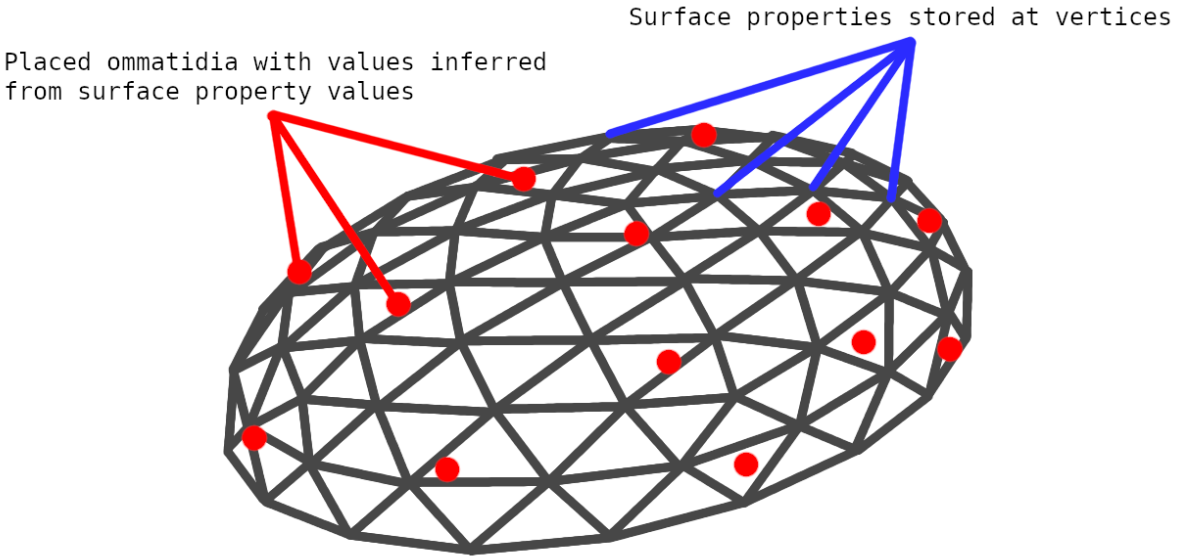
In the sample image above, a U- and V- local ommatidial coordinate space axial offset is demonstrated, with the W-axial offset (that is always negative) controlling the frustum's conical spread, and the U- and V- axis controlling the frustum's shear about the origin along each respective axis. In practice, 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).

Representation 2: 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. This is consistent with 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:



While *POSITION* and *ORIENTATION* are taken from the surface position and normal, it is entirely possible 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.

Representation 3: Spherical Properties Representation

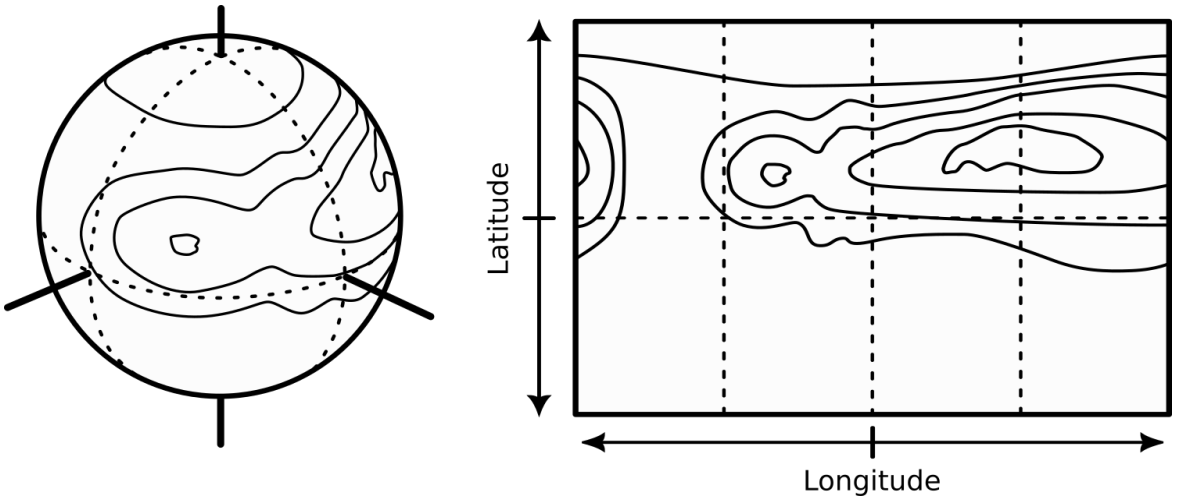


Figure N. A spherical model of the compound eye. Left: An arbitrary property plot mapped onto a sphere. Right: The “unwrapped” property plot in 2D space.

The **spherical properties representation** more closely aligns to eye depictions and data collected in older works of research, where ommatidial properties were recorded against their orientation, forming a spherical plot of the values, entirely excluding the surface shape. Due to the simplicity in this measurement methodology, it has been in use for many decades despite its comparative lack of information.

As the representation is simply values spread over the surface of a sphere, they naturally align to spherically-mapped media within computer graphics, namely 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” to be defined, which is a single value that describes the average radius of the eye. This allows these eye types to be compared directly to the others, and allows for the conversion from this form to the other two (**point-ommatidial** and **surface**) using the same surface-point-placement approach outlined in the **surface** representation.

At the time of writing, GITF has no defined 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, as seen in **Figure N**) must be used for spherical image data. See the “Surface Properties Representation” subsection of the “Proposed Compound Eye Extension” section below for further information.

4.2.3. 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.

4.2.4. Ommatidial Property Management

As described in section “3.1. Compound Eye Primitives Overview”, **ommatidial properties** are datasets that define arbitrary variables that change on a per-ommatidial basis. They can describe properties that are required to specify the view frusta of each ommatidium (position, orientation, facet diameter etc.) or extraneous properties (such as polarisation sensitivity, light response curves, facet shape etc.).

As an eye itself can be represented in a number of different ways (described above as **point-ommatidial**, **surface properties** or **spherical**), and the kind of data that can be classified as an “ommatidial property” is vast (facet diameters are simple numbers, while orientations can be stored as 3D vectors, and yet still time series data can be in the form of a large array of primitives). As such, an individual **ommatidial property** can be defined in many ways:

1. A set of property values, individually given for each ommatidium. As in **Table 1**, in the case where a full list of ommatidia is considered (a.k.a. In the **point-ommatidial** representation), each **ommatidial property** is defined as a list of values, one for each ommatidium.
2. An image or array of property values, showing the distribution of the property (a *property map*). In **point-ommatidial** and **surface properties** representations, individual ommatidia are not directly specified, requiring **ommatidial properties** to describe their changing property values over the eye surfaces (mesh and spherical respectively) they are attached to. The specifics of the mapping from the 2D image or array of the property map to the 3D surface of the eye depends on the specifics of the eye defined (for instance, spherical eyes will use a spherical look-up method, mesh surfaces may use UV or vertex mapping).
3. A singular property value. In the case that a property is not fully known (such as when an average is taken of all values for any given property), it is suitable to define a single value to be used as the property value at all ommatidia.

While in all cases the underlying data will be trivial (an N-dimensional list of 1 or more values), supporting information describing how to interpret the data is required. In this way, an **ommatidial property** primitive will consist of not only the data stored in each ommatidial property, but also interpretation data. In this way eyes can be associated with **ommatidial properties** without having to explicitly describe the structure of each property.

4.3. 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 GITF files may lead to undefined interpretations of their data.

4.3.1. Sub-Extensions

t 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, the base OCES definition has been limited to the absolute minimum required to describe the acceptance fields of compound eyes.

Taking from the GITF 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.

4.3.2. 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 GITF 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 GITF'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.

4.3.3. 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

5. Introduction

This section of the document gives a brief overview of the glTF 2.0 file format for scene representation and outlines at the conceptual level the OCES glTF 2.0 extension that is capable of storing any number of the primitives described in the *Open Compound Eye Standard Primitives* section.

The full schema, written in JSON Schema, can be found at the OCES github repository: https://github.com/Blayzeing/open-compound-eye-standard/tree/main/schema. This schema can be used to validate OCES-compatible glTF files and as a guide for constructing and structuring OCES data within a glTF file.

6. 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 (a file that defines structured objects as a set of pairs of *keyword properties* - namely, key-value pairs consisting of a property name; here referred to as its *keyword*, and a value, which can be numeric, textual or a further JSON object) of 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**):

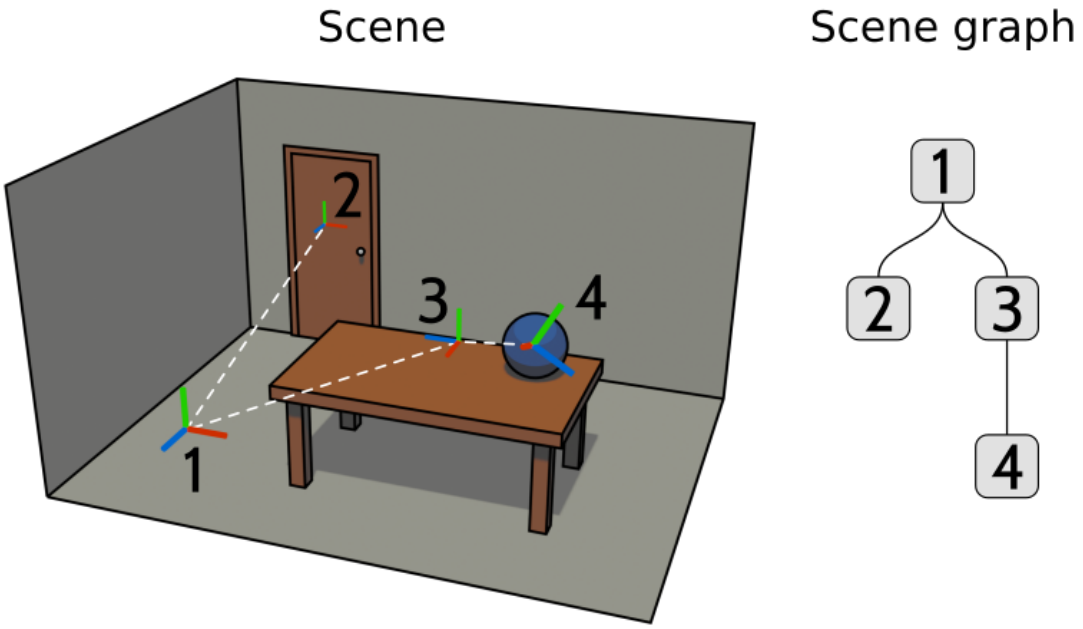


Figure 1. An example scenegraph showing 4 objects: A door, table, ball and the room itself. The door and table are *children* of the room, with the ball being a child of the table.

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.

6.1. 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 GlTF 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.

7. 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.

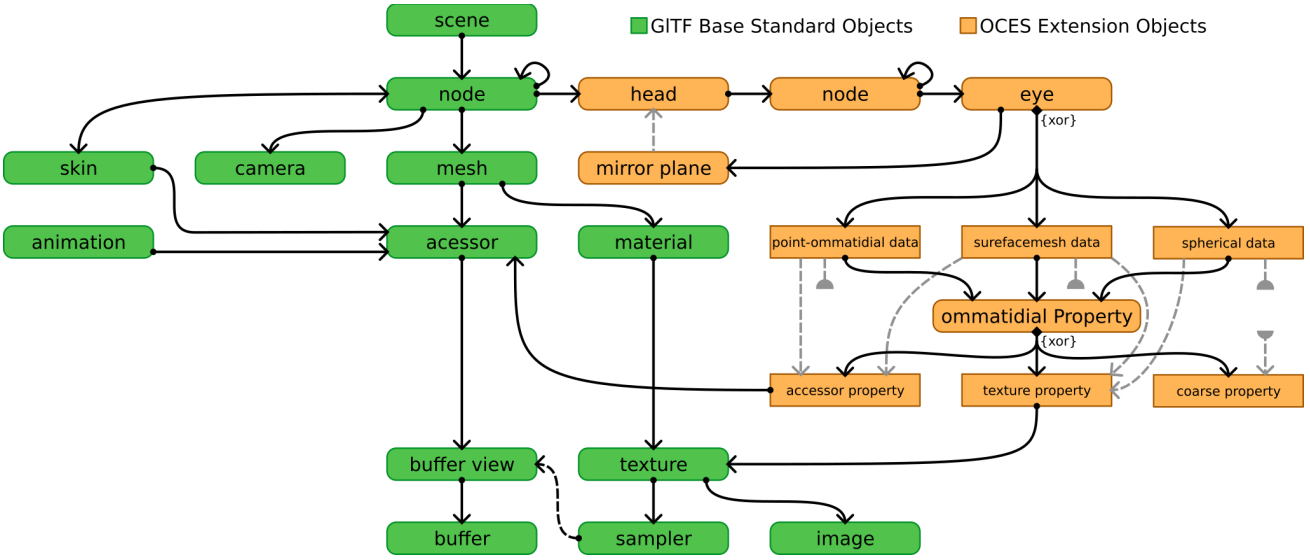


Figure 2. An extended OCES-glTF object map. In green is listed the standard glTF hierarchy, in orange the additional objects outlined within this extension. Rounded rectangles indicate objects, squared rectangles indicate types of an object (for instance eye objects can be one of three types: point-ommatidial, surfacemesh, or spherical). Black solid arrows indicate when one object references one or more of the pointed-to objects. Grey dashed lines indicate when an object depends on the data of another (mirror planes use their associated head object’s coordinate system, and each eye type determines which ommatidial property types they can use).

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. 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.

7.1. 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” keyword property 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 **will** override any true “enabled” values of any **eyes** below it in the scene hierarchy, but **will not** override false “enabled” values of any **eyes** below it in the scene hierarchy. In other words, an individual **eye**’s enabled-ness will be the binary *inclusive or* of the **eye**’s “enabled” value and its associated **head**’s “enabled” value.

7.1.1. Keyword Properties

These are the keyword properties for head objects:

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**.

7.2. 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 “X” or “dorsoventral” normal heading/mirror direction) may be referred to by name when configuring their normal direction.

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] },
       { normal: "DORSOVENTRAL" },
       { normal: "Z" }
]
...
```

7.2.1. Keyword Options

These are the keyword properties for mirrorPlane objects:

Key	Type	Default	Comment
name	String	“”	**Optional**. May provide the name of the mirror plane.
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.

7.3. Ommatidial Properties

As described in the OCES Primitives section, an **Ommatidial Property** is a set of data that describes the per-ommatidial properties of the eye. This can be defined as either point-data (one value from each property set for each ommatidium) as in the case of the **point-ommatidial** representation, or as an image map (a plot - or texture - over the eye surface that describes how the property changes) as is the case of the **surface properties** and **spherical properties** representations.

GlTF has built-in data representations for storing textures and arrays of data in the form of **textures** and **accessors**. Additionally, instead of defining an entire set of data, sometimes the real-world information collected may not be available to the level of every individual eye - for instance, oftentimes an average might be measured (such as the average facet diameter, or average spectral response curve). In this case only a single value is required. We refer to these values as **coarse values**.

To support these three types of data, an **OmmatidialProperty** object has been defined that acts as a wrapper around one of, optionally, an **accessor**, **texture**, or directly defined **coarse value**. OmmatidialProperty objects define a *type* keyword that specifies which type of data they represent, and a *value* keyword, which will either (in the case of a **coarse value**) be the data itself, or (in the other cases) be a pointer to an accessor or texture. These OmmatidialProperty objects are stored in a list at the root level of the document, in the same way that **mirrorPlanes** and **eyes** are, ready to be index-referenced from individual **eye** objects.

The below example shows three **OmmatidialProperty** objects, each wrapping one of the three types of properties:

```
Unset
...
ommatidialProperties: [
	{
		type: "TEXTURE",
		value: 0,
		textureScale: 2,
		textureCenter: 0.5
	},
	{
		type: "ACCESSOR",
		value: 0
		dataStride: 1
	},
	{
		type: "COARSE",
		value: [0.2, 0.5, 0.1]
	},
]
```

```
...
```

The first wraps a texture (in the 0th position of the glTF file's list of **textures**). The second wraps an array (in the 0th position of the glTF file's list of **data accessors**). Finally, the last ommatidial property directly stores a coarse value - in this case the 3D vector "[0.2, 0.5, 0.1]". When this ommatidial property is referenced, this value will always be used, no matter which ommatidium or where on the eye surface it is being referenced from.

The second ommatidial property also has an additional keyword property in the form of *dataStride* that lists how many of the array's objects should be used for each data point. For instance, if an ommatidial property required a list of 7 pairs of numbers for each ommatidium, these pairs would be stored in an array of 2D vectors with a length 7 times the number of ommatidia - then *dataStride* configured as the value 7 indicates to the program reading the file that each ommatidium needs to reference 7 values from the array. In other words, index from the first of *dataStride* elements within the array can be found by taking the index of the ommatidium, *ommatidialIndex*, and multiplying it by the data stride:

$$Ommatidial\ property\ first\ element\ index\ =\ ommatidialIndex\ *\ dataStride$$

The first ommatidial property also has additional keyword properties in the form of *textureScale* and *textureCenter*. Within real-time graphics, textures are often considered as sets of N-dimensional vectors where each component can span from 0 to 1 inclusive. This works well for colours (which can be described as a ratio of red, green and blue), however for spatial values may not work so well. To avoid having to break away from standard definitions, all **glTF textures** within the document are assumed to also range from 0 to 1 - including those referenced by TEXTURE-type **ommatidialProperty** objects. However, in practice ommatidial properties may need to vary outside of this bounded range. For this purpose, the *textureSCale* and *textureCenter* keyword options have been specified for TEXTURE-type **ommatidialProperty** objects. This **must** be taken as modifiers to data retrieved from the texture by any program parsing the file following this simple equation:

$$True\ ommatidial\ property\ value\ =\ (textureValue\ -\ textureCenter)\ *\ textureScale$$

Where *textureValue* is the value read from the texture. In this way, the coordinate space of a given TEXTURE-type **ommatidialProperty** object can be transformed to match a desired range.

7.3.1. Keyword Options

These are the keyword properties for ommatidialProperty objects:

Key	Type	Comment
type	String	**Required.** Describes the type of data that this ommatidialProperty wraps. Must be one of: • ACCESSOR

		• TEXTURE • COARSE
value	Integer or direct value	**Required.** If *type* is ACCESSOR or TEXTURE, refers to an index in the glTF accessors or textures list, depending on *type*'s value. If *type* is COARSE, can be any single numeric value or an array of any length of numeric values, or a 2D array of any primary length and then up to 16 dimensions of width. Assumed to be the same as an accessor (elements the max size of which can be 4x4 matrices), but with only one element.
dataStride	Integer	**Optional, defaults to 1.** Only considered if *type* is ASSESSOR, specifies the data stride of the array this ommatidialProperty is wrapping. For instance, if the data is a set of 3D points per ommatidium (such as a position map), then *dataStride* can be 1, with the accessor being an array of 3D vectors. If the data is a set of 5 3D points per ommatidium (such as a movement map over 5 points in time), then the *dataStride* can be 5, with the accessor being an array of 3D vectors.
textureScale	Scalar, 2D vector, 4D vector	**Optional, defaults to 1.** Only considered if *type* is TEXTURE. As **glTF texture** objects are usually defined such that each point in the textures define values between [0,0,0] and [1,1,1], this value is applied to any retrieved texture values by way of multiplication. For instance, a 1D displacement map texture will store values between 0 and 1, and setting *textureScale* to 10 will result in displacements between 0 and 10. **Should** be scalar, so all texture values are within the same scale on each axis, however sometimes this may be undesired.
textureCenter	Scalar, 2D Vector, 4D vector between the origin and the unit value (*1*, *[1,1]*, *[1,1,1]*, *[1,1,1,1]*)	**Optional, defaults to 0.** As per *textureScale*, affects the way texture values are processed. Before being scaled by *textureScale*, this value is subtracted from retrieved texture values. For instance, a 1D displacement map texture will store values between 0 and 1. Setting *textureCenter* to 0.5 will result in displacements between -0.5 and 0.5. Additionally, setting *textureCenter* to 0.5 and *textureScale* to 2 will result in displacements between -1.0 and 1.0. **Should** be scalar, so all texture values are within the same scale on each axis, however sometimes this may be undesired.

7.4. Eye

Eye objects define the actual eye data structure. They, like **mirrorPlanes** and **ommatidialProperties**, are defined in a list within the top-level OCES extension object. All eyes contain at least the keyword properties of *name*, *type* and *enabled*, as well as a list of

associated **mirror plane** indices (under the keyword *mirrorPlanes*) and a list of **ommatidial property** indices (under the keyword *ommatidialProperty*).
Required keyword properties, as well as minimum required ommatidial properties 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 (one for each type of eye), with the shared keyword options listed below.

Shared Keyword Properties

The following keyword options are defined for all eyes, no matter the data type:

Key	**Type**	**Default**	**Comment**
name	String	“”	**Optional**. May 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.
ommatidialProperties	JSON object consisting of “Keyword: Integer” pairs	[]	A JSON object acting as a dictionary referencing ommatidial property names to **ommatidialProperties** by their index within the ommatidial properties list that describe this eye dataset. The minimum required properties change depending on the *type* of eye defined, however an example may look as below: Unset `ommatidialProperties: { "POSITION": 0, "ORIENTATION": 1, "DIAMETER": 2, "FOCAL_OFFSET": 3 }` Where each named ommatidial property is referring to an **ommatidialProperty** object within the root *ommatidialProperties* list.

7.4.1. Point-Ommatidial Data

When the *type* key is “POINT_OMMATIDIAL”, no additional keyword properties are required, however following **ommatidialProperties** *are* required (if not given, parsers will populate the property as a coarse property containing the default value).

Ommatidial Properties

Since point-ommatidial data is defined on a per-ommatidial basis, with each ommatidium defined as a single data-rich point, point-ommatidial eyes can only use ACCESSOR or COARSE **ommatidialProperties**, defining the properties of each ommatidium in a large array or as a single value to be used at all points.

Point-ommatidial data also stores only this data, so it is necessary to fully define all spatial properties of the ommatidial data coarsely or via an array accessor:

Ommatidial Property	**Type (coarse or array)**	**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 virtual 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 per-ommatidium vector is of the form [0,0,W]). In both cases, the W-axis offset **should** be negative and **must not** be zero.

7.4.2. Surface Properties Data

Unlike the **point-ommatidial** and **spherical** data formats, the **surface properties** format defines a *surface mesh* that most of its spatial ommatidial properties are defined relative to. The definition of this surface mesh is purposefully similar to the GITF geometry standard, however as only one mesh will ever be defined per eye, it is reduced in its complexity and more specific nomenclature is used. Surface geometry is stored in the “surface” keyword as

a collection of *vertex attributes* encoded as a simple JSON object that makes named index references to **glTF accessors**:

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

Here (just as in the POSITION and ORIENTATION properties from the point-ommatidial representation), a set of oriented vertices is defined in 3D space via references to two array of 3D vectors of vertex positions and vertex normals at the Ith and Jth positions in the glTF list of **accessors**.

The INDICES keyword property references an **accessor** (of size 3n) that stores, in triplets of counter-clockwise indices referring to locations in the POSITION and ORIENTATION arrays, a set of faces. If only POSITION, NORMAL and INDICES is defined, then **ommatidial properties** are assumed to be of type ACCESSOR and of length equal to the number of vertices (i.e. the length of POSITION and NORMAL lists). This way additional **ommatidialProperty** objects are associated with this eye as values at each vertex of the mesh - they are *vertex attributes* similar to vertex colour mapping/shading approaches found within real-time graphics applications.

Instead of using a vertex-attribute approach, **ommatidial properties** can be mapped to the surface using a form of UV-mapping, allowing property maps of ommatidial properties to be stored in 2D images by associating an **ommatidialProperty** object of type TEXTURE. This requires the additional definition of the TEXTURE_COORD and TEXTURE_INDICES within the surface keyword property:

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

TEXTURE_COORD references a **glTF buffer accessor** that describes an arbitrarily-long set of 2D points to perform UV mapping with (the *unwrapped mesh*), and TEXTURE_INDICES references a buffer accessor that is the same length as INDICES specifying, for each of the points in each of INDICE's counter-clockwise triangles, an index within the TEXTURE_COORD array, mapping the association between 3D face vertices and their corresponding 2D texture vertices. With both of these values defined, **ommatidialProperty** objects must be of type TEXTURE (if not COARSE) and reference 2D **glTF texture** objects that store the property's distribution in an N-channel Image which utilises the UV-map

described by the TEXTURE_COORD and TEXTURE_INDICES **accessor**. Both TEXTURE_COORD and TEXTURE_INDICES **must** be defined to use UV-mapped property data.

Ommatidial Properties

Since *Surface Properties Data* eyes have a defined eye surface mesh in the form of the object's *surface* keyword property, many of the spatial ommatidial properties are derived from or directly defined by, the surface mesh. As an example, the POSITION ommatidial property is no longer used, as this information is directly inferred from the surface mesh. However, DISPLACEMENT is defined, allowing for finer control over the perceived shape of the surface mesh, without having to define an incredibly high-resolution mesh, much in the same way that bump or displacement mapping is used in real-time graphics.

Property	**Type (coarse, array or texture)**	**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 and **must not** be zero.
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 bump/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 defines a new set of ommatidial directions that completely overwrites the default orientations. Only one **should** be defined, but if both are
ABSOLUTE_ORIENTATION			

			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.

Keyword Properties

Additionally, surface properties data allows for these keyword properties:

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

7.4.3. Spherical Properties Data

The most abstracted form of eye representation covered by the OCES standard, **spherical properties data** eyes make the assumption that the eye surface is spherical, and that all data is stored as a spherical map. As previously noted, GITF has no defined 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:

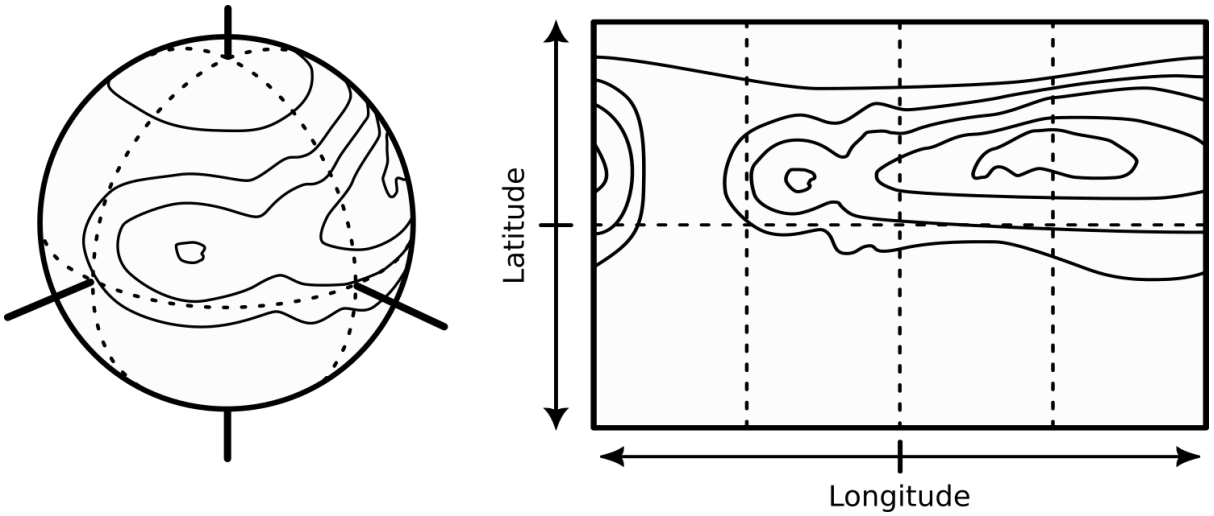


Figure 3. A spherically-mapped image. **Right:** The image as it is stored in the computer in 2D space. **Left:** The image as it is wrapped around a sphere.

Ommatidial Properties

Property	Type	Default	Comment
OMMATIDIAL_BOUNDS	1-channel Image	1	Specifies the extent to which ommatidia extend to on the sphere. Essentially a mask showing the maximum bounds of

			the eye, in spherical-space, where a value of 1 corresponds to an ommatidial area, and a value of 0 corresponds to an area where no ommatidia are present. Using a COARSE ommatidial property here is reductive, as it essentially amounts to either disallowing entirely ommatidial placement on the surface, or allowing them to be placed anywhere.
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 ABSOLUTE_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 limited to only the ABSOLUTE_ORIENTATION

Keyword Properties

Additionally, spherical properties data allows for these keyword 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

7.4.4. Example Eyes

Here are some example eyes:
TODO

7.5. 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
mirrorPlanes	[mirrorPlane]	An array of **mirrorPlane** objects, later to be referenced by index.
eyes	[eye]	An array of **eye** objects, later to be referenced by index.
version	String	**Must** be provided. A string representing what version of the OCES standard the OCES-specific data is following. **Must** be in the format of “major.minor.fix” that this file follows. *The latest version is currently 0.4.0.*
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, as implemented through RFC 3339’s “date-time” ABNF rule (e.g. `YYYY-MM-DDThh:mm:ss.ms+00:00`), full specification recommended, **should** extend to at least minute resolution). When parsing this it is strongly recommended to use an external trusted library.
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

8. General Guidance to File Composition

- Point about the expected dual use-cases of OCES-glTF files
    - We expect OCES-glTF files to be used in one of two ways:
        - Just to store an eye/head object, to then be integrated into other tools (renderers, comparison systems, environments etc)
        - As a complete experimental set-up, with multiple eye/head objects, **as well as** a 3D environment (with potential other experiment-specific extras and extensions in-use) stored within the rest of the glTF file.
    - Because of this,
- Point about how to encourage reproducible experimentation/science, we expect people to store the same eye in many representations at the same time - this is expected to be a single head object, with many eye objects, for instance - one storing the original spherically-represented data collected by the experimenter, and alongside it in the same place, the point-ommatidial representation.
    - This allows other researchers to have access to the actual collected data (the spherical map) as well as the exact permutation of the point-ommatidial representation used to perform experiments.
    - In other words, if data is collected in a low-fidelity way such that assumptions (such as generating positions and orientations based off of surface or spherical properties) need to be made to generate specific point-ommatidial representations required for experimentation, it should be expected that a head object will store both the originally collected data (to allow for peers to examine the original data) *as well as* the point-ommatidial data (to allow for peers to perfectly reproduce the experimental configuration).
    - As such tools that parse OCES-glTF files should be able to accommodate both situations (e.g. have an explicit “import eyes from OCES-glTF file” option as well as an “import whole OCES-glTF file” option, OR simply be aware that they might have to import OCES-glTF files as “additional objects” into an already-defined scene-graph, essentially merging them.

9. 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.

9.1. 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.

10. Inter-primitive Conversion Methods

As mentioned in the standard, an eye can be represented in one of three ways, and each of these can be converted (in a lossy fashion) from one to the other:

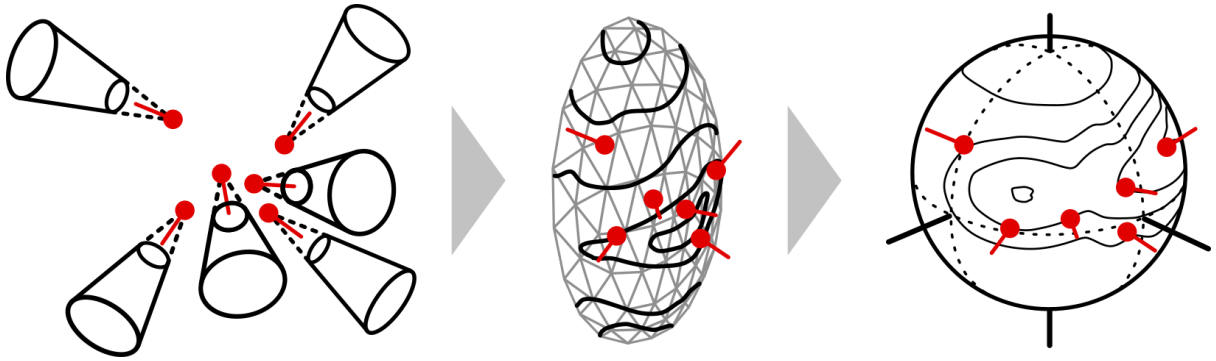


Figure N. The same six ommatidia (red) as represented in each representation, from highest (left, **point-ommatidial**) to lowest (right, **spherical**) fidelity. Note that only the point-ommatidial representation stores the specific position and orientation of the ommatidia, while the other representations only store distributions of these values, leaving the specific ommatidia to be generated in accordance with these distributions. Conversions are possible between each representation, however it is a lossy process in the high-to-low direction.

This section outlines methods to convert between each primitive, should a situation arise.

11. Schema and Reference Implementations

11.1. 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>.

11.2. 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 an ommatidial property is not defined, but the property has a “default” value described in the above schema, Parsers **must** act as if a coarse ommatidial property has been defined with this value. For example, if a **point-ommatidial** eye has been defined with no POSITION ommatidial property, then when referenced, position data should always return [0,0,0] - as if there were a coarse ommatidial property defined with that value, matching the point-ommatidial POSITION’s default value.

11.3. 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)

11.4. Reference Implementation Notes

11.5. 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

12. 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.

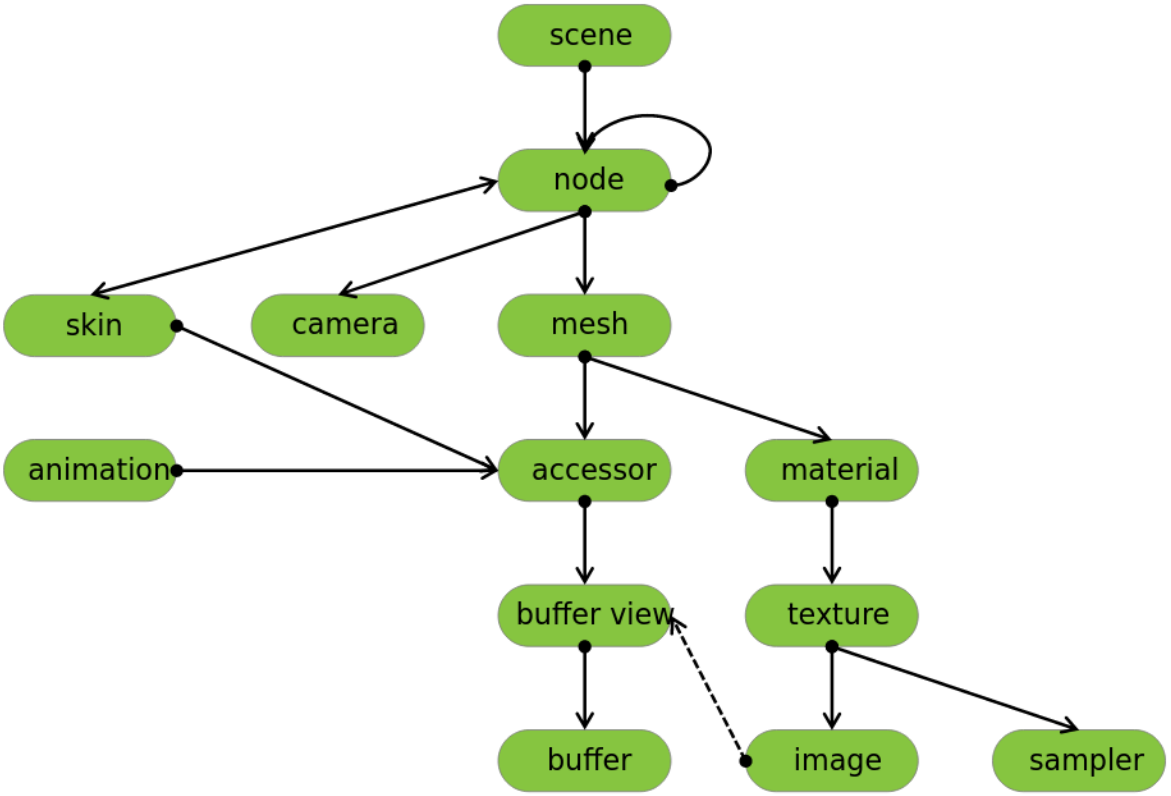
```
Unset
{
        "scene" : 0,
        "scenes" : [
        {
                "name" : "Scene",
                "nodes" : [
                0
                ]
        }
        ],
        "nodes" : [
        {
                "mesh" : 0,
                "name" : "triangle",
                "translation" : [
                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="
        }
        ]
}
```


13. 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.



14. C.

As of version 0.4.0, this example is out of date

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" }
        }
        }
        }
}
```

15. D.

TODO: Flesh out this section

While in reality, the per-ommatidial acceptance region is a complex function of describing the probability of light entering the ommatidium across the angular region of the cone of acceptance, much rendering and analysis literature surrounding the nature of this sampling function works on the assumption that this sampling function can be described as a Gaussian distribution with respect to the angular spread from the ommatidial axis:

- CompoundRay, 2022
- Polster et. al.
- The optics of the compound eye of the honeybee (Apis mellifera), Varela 1970