

Open Game Engine Exchange Specification

Version 3.0

by Eric Lengyel

Terathon Software LLC
Lincoln, California

**Open Game Engine Exchange Specification
Version 3.0**

ISBN-13: 978-0-9858117-8-5

Copyright © 2021, by Eric Lengyel

Published by Terathon Software LLC
terathon.com

OpenGEX website: opengex.org



All mathematical expressions in this document
were typeset with the Radical Pie equation editor.
radicalpie.com

Contents

1 Introduction	1
2 Structure Specification	7
Animation	9
Atten	12
BoneCountArray	15
BoneIndexArray	16
BoneNode	17
BoneRefArray	18
BoneWeightArray	19
CameraNode	20
CameraObject	22
Clip	23
Color	25
GeometryNode	26
GeometryObject	28
IndexArray	30
Key	32
LightNode	34
LightObject	36
Material	39
MaterialRef	42
Mesh	43

Metric 46

Morph..... 50

MorphWeight..... 52

Name 54

Node 55

ObjectRef..... 57

Param..... 58

Rotation 59

Scale 62

Skeleton 64

Skin..... 65

Spectrum..... 68

Texture..... 73

Time..... 76

Track..... 78

Transform 81

Translation 84

Value..... 86

VertexArray 88

A OpenDDL Reference..... 91

B Revision History 115

1

Introduction

The Open Game Engine Exchange (OpenGEX) format is a text-based file format designed to facilitate the transfer of complex scene data between applications such as modeling tools and game engines. The OpenGEX format is built upon the data structure concepts defined by the Open Data Description Language (OpenDDL), a generic language for the storage of arbitrary data in human-readable format. This specification provides a description of the data structures defined by OpenGEX, and it includes an OpenDDL reference in Appendix A.

At the most basic level, an OpenGEX file consists of a node hierarchy, a set of objects, a set of materials, and some additional information about global units and axis orientation. The various node, object, and material structures contain all of the details such as geometric data and animation tracks within a hierarchy of additional types of structures defined by OpenGEX. The relationships among all of these structures are shown in Figure 1.1.

Nodes

The node hierarchy represents the overall organization of the scene. An OpenGEX file may contain any number of nodes at the root level, and each node may contain any number of child nodes. The nodes form tree structures in which each node can have at most one parent node.

Each node in the hierarchy may have data structures containing transformations and animation tracks. Nodes representing geometries, lights, and cameras in the scene have references to other structures containing the geometric data and various parameters for those objects. Multiple nodes may reference the same object to achieve instancing.

Geometry nodes may have one or more references to materials containing information about surface shading. Materials may be referenced by any number of geometry nodes.

Objects

The data belonging to geometries, lights, and cameras without regard for placement in the overall scene is stored in a flat set of object structures. An OpenDDL reference is used to make the connection between each node in the scene and the object that it instances.

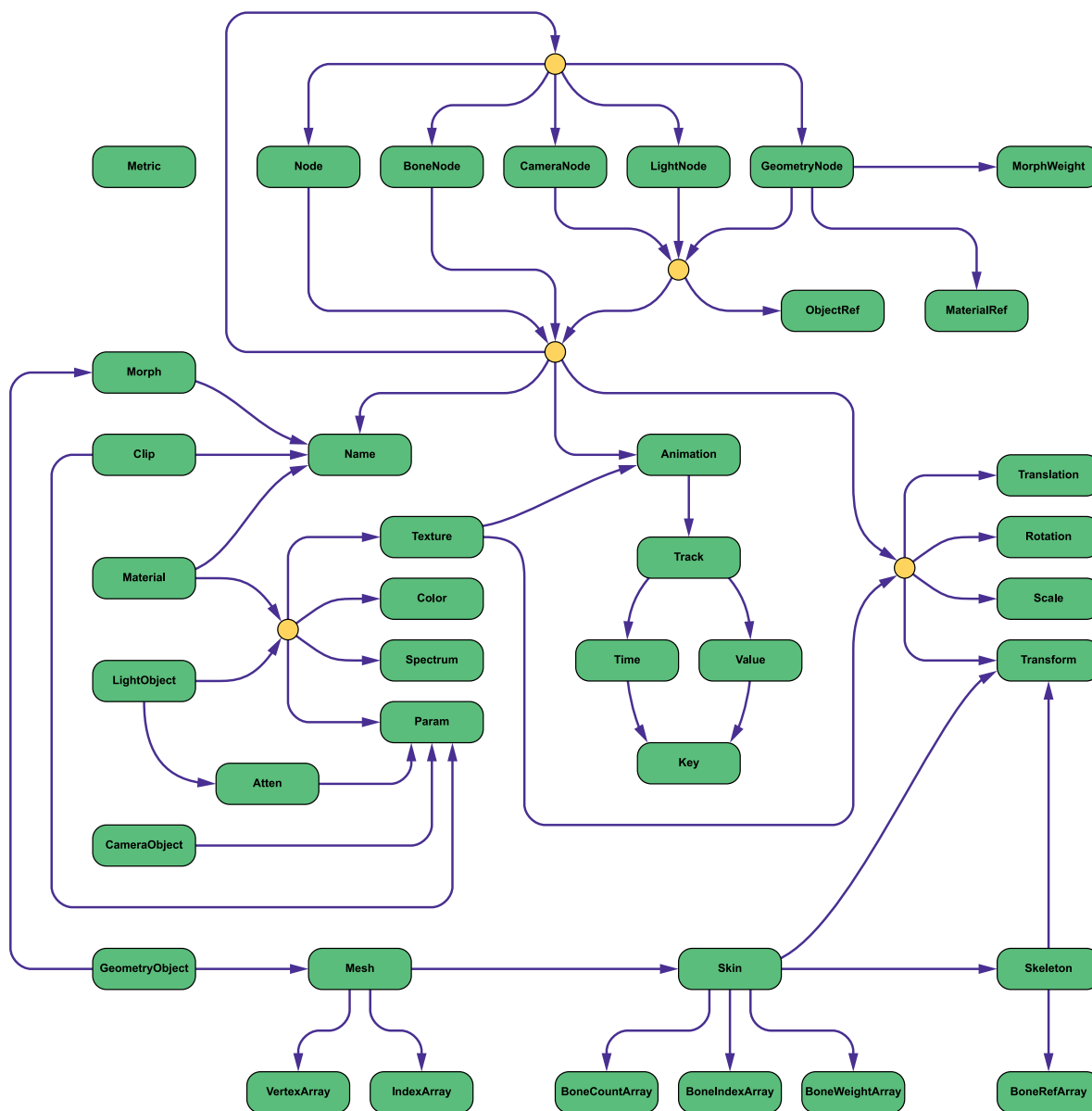


Figure 1.1. This diagram illustrates the relationships among the structures defined by the OpenGEX format. The purple arrows point from each of the structures to the specific substructures they are allowed to contain. (Substructures that are simply OpenDDL data types have been omitted.) The circular orange nodes serve only to combine paths in order to simplify the diagram where common relationships exist.

Geometry objects contain a set of one or more mesh structures that each contain vertex and primitive information as well as optional skinning data. Light objects contain information about a light's color, brightness, and attenuation functions. Camera objects contain information about a camera's field of view and clipping planes.

Materials

A material structure contains basic information about various colors and texture maps used by a surface shader. Texture maps may include texture coordinate transformations and animation tracks that affect those transformations.

Animation

Node transformations, morph weights, and texture coordinate transformations may all be animated through the inclusion of `Animation` structures inside `Node` structures and `Texture` structures. A complete transformation may be decomposed into multiple components, such as rotations about one or more axes followed by a translation, and an animation may contain several tracks that animate each component separately. An OpenGEX file may contain multiple animation clips, and each `Animation` structure identifies which clip it belongs to. Information about a complete animation clip is stored inside a `Clip` structure that can appear at the top level of the file.

Extensions

OpenDDL requires that readers ignore any structures and properties having an unrecognized identifier without producing an error. This allows extensions to be implemented through new structure types and property values without breaking readers that are not aware of them.

Example

A very simple example of a complete OpenGEX file describing a green cube is shown in Listing 1.1. It begins with a pair of `Metric` structures that define the units of distance measurement and the global up direction. Those are followed by a single `GeometryNode` structure that provides the name and transform for the cube. The geometric data for the cube is stored in the `GeometryObject` structure that is referenced by the geometry node. The geometry object structure contains a single mesh of triangle primitives that includes per-vertex positions, normals, and texture coordinates. Finally, the `Material` structure at the end of the file contains the green diffuse reflection color.

Listing 1.1. This is an example of a near-minimal OpenGEX file containing the data for a green cube. It consists of a single geometry node that references a geometry object and a material.

```

Metric (key = "distance") {float {0.01}}
Metric (key = "up") {string {"z"}}

GeometryNode $node1
{
  Name {string {"Cube"}}
  ObjectRef {ref {$geometry1}}
  MaterialRef {ref {$material1}}

  Transform
  {
    float[12]
    {
      {1.0, 0.0, 0.0,
        0.0, 1.0, 0.0,
        0.0, 0.0, 1.0,
        50.0, 50.0, 0.0}
    }
  }
}

GeometryObject $geometry1 // Cube
{
  Mesh (primitive = "triangles")
  {
    VertexArray (attrib = "position")
    {
      float[3] // 24
      {
        {-50.0, -50.0, 0.0}, {-50.0, 50.0, 0.0},
        {50.0, 50.0, 0.0}, {50.0, -50.0, 0.0},
        {-50.0, -50.0, 100.0}, {50.0, -50.0, 100.0},
        {50.0, 50.0, 100.0}, {-50.0, 50.0, 100.0},
        {-50.0, -50.0, 0.0}, {50.0, -50.0, 0.0},
        {50.0, -50.0, 100.0}, {-50.0, -50.0, 100.0},
        {50.0, -50.0, 0.0}, {50.0, 50.0, 0.0},
        {50.0, 50.0, 100.0}, {50.0, -50.0, 100.0},
        {50.0, 50.0, 0.0}, {-50.0, 50.0, 0.0},
        {-50.0, 50.0, 100.0}, {50.0, 50.0, 100.0},
        {-50.0, 50.0, 0.0}, {-50.0, -50.0, 0.0},
        {-50.0, -50.0, 100.0}, {-50.0, 50.0, 100.0}
      }
    }
  }
}

```

```
VertexArray (attrib = "normal")
{
    float[3]      // 24
    {
        {0.0, 0.0, -1.0}, {0.0, 0.0, -1.0}, {0.0, 0.0, -1.0},
        {0.0, 0.0, -1.0}, {0.0, 0.0, 1.0}, {0.0, 0.0, 1.0},
        {0.0, 0.0, 1.0}, {0.0, 0.0, 1.0}, {0.0, -1.0, 0.0},
        {0.0, -1.0, 0.0}, {0.0, -1.0, 0.0}, {0.0, -1.0, 0.0},
        {1.0, 0.0, 0.0}, {1.0, 0.0, 0.0}, {1.0, 0.0, 0.0},
        {1.0, 0.0, 0.0}, {0.0, 1.0, 0.0}, {0.0, 1.0, 0.0},
        {0.0, 1.0, 0.0}, {0.0, 1.0, 0.0}, {-1.0, 0.0, 0.0},
        {-1.0, 0.0, 0.0}, {-1.0, 0.0, 0.0}, {-1.0, 0.0, 0.0}
    }
}

VertexArray (attrib = "texcoord")
{
    float[2]      // 24
    {
        {1.0, 0.0}, {1.0, 1.0}, {0.0, 1.0}, {0.0, 0.0},
        {0.0, 0.0}, {1.0, 0.0}, {1.0, 1.0}, {0.0, 1.0},
        {0.0, 0.0}, {1.0, 0.0}, {1.0, 1.0}, {0.0, 1.0},
        {0.0, 0.0}, {1.0, 0.0}, {1.0, 1.0}, {0.0, 1.0},
        {0.0, 0.0}, {1.0, 0.0}, {1.0, 1.0}, {0.0, 1.0},
        {0.0, 0.0}, {1.0, 0.0}, {1.0, 1.0}, {0.0, 1.0}
    }
}

IndexArray
{
    uint32[3]     // 12
    {
        {0, 1, 2}, {2, 3, 0}, {4, 5, 6}, {6, 7, 4}, {8, 9, 10},
        {10, 11, 8}, {12, 13, 14}, {14, 15, 12}, {16, 17, 18},
        {18, 19, 16}, {20, 21, 22}, {22, 23, 20}
    }
}

}

Material $material1
{
    Name {string {"Green"}}

    Color (attrib = "diffuse") {float[3] {{0, 1, 0}}}
}
}
```


2

Structure Specification

This section provides a detailed specification for each of the 39 data structures defined by OpenGEX. The structures appear in alphabetical order without regard for the possible hierarchical relationships or the order in which they are likely to appear in an OpenGEX file.

The description of each data structure includes tables that list the properties that are accepted, the types of data structures that may appear as substructures, and the possible containing structures. The meaning of the information provided in these tables is discussed below.

Properties

If a structure accepts any property values, then they are listed in a table with a blue heading, as shown in the example below. The name of each property is listed in the first column, and its OpenDDL data type is listed in the second column. The default value used when a particular property is not specified is listed in the third column. Some properties do not have default values, and this means that either the property is required or there is some specified behavior that should be followed when the property is not present. Many properties have a small set of possible values, and in such a case, a description of each particular value is provided where necessary.

Property	Type	Default	Description
example	int32	0	A description of the property value.

Structure Data

The types of substructures composing the data payload for a particular OpenGEX structure are listed in a table with a green heading, as shown in the example below. The allowed substructures may include other OpenGEX structures, native OpenDDL data types, or both. Except where noted otherwise, the order in which substructures appear is not significant. The name of each possible substructure is listed in the first column. The second and third columns indicate the minimum and maximum number of times that each substructure may occur within the structure. A dash in the third column indicates that there is no maximum number.

Substructure	Min	Max	Description
example	0	1	A description of the relationship to the substructure.

Hierarchy

The types of structures that may hierarchically contain a particular OpenGEX structure are listed in a table with an orange heading, as shown in the example below. The allowed containing structures listed in the first column are always OpenGEX structures. For structures that are allowed to exist at the root level of the file, there is an entry for which a dash is listed in the first column.

Containing Structure	Description
example	A description of the relationship to the containing structure.

File Names

Where file names are specified inside a structure's data (currently possible only inside the Texture structure), they shall be formatted according to the following rules.

- A file name consists of a sequence of zero or more directory names followed by the name of the file and an optional extension.
- The delimiter between directory names and between the last directory name and the file name must always be a single forward slash character having ASCII value 47.
- A file name representing an absolute path begins with a forward slash character, and a file name representing a relative path does not. In the case of a relative path, a file name specifies a path relative to the directory containing the OpenGEX file being processed.
- If a file name begins with two consecutive forward slash characters, then the first directory name following the two slashes should be interpreted as a volume name or drive letter, as appropriate for the operating system.
- A file name shall not contain any of the characters \ : * ? " < > | having ASCII values 92, 58, 42, 63, 34, 60, 62, and 124, respectively.

Animation

The Animation structure contains animation data for a single node in the scene. Each animation structure is directly contained inside a node structure (Node, BoneNode, GeometryNode, CameraNode, or LightNode) or Texture structure, and it contains the data needed to modify its sibling transform structures (Transform, Translation, Rotation, and Scale) or sibling MorphWeight structures over time.

Tracks

An Animation structure contains one or more Track structures, and each track contains the data needed to modify the data contained in one structure, the track's target. A track identifies its target by specifying the local OpenDDL name of a structure contained inside the same node structure that contains the Animation structure. In Listing 2.1, the Translation structure named %xpos is targeted by the animation track, and this track contains five keys that define the value that the translation should attain at five points in time. In this case, the animation causes the geometry to move four units of distance in the positive x direction for each unit of time.

Listing 2.1. The animation track in this example modifies the data inside the Translation structure having the local OpenDDL name %xpos.

```
GeometryNode
{
  Translation %xpos (kind = "x")
  {
    float {0.0}
  }

  Animation
  {
    Track (target = %xpos)
    {
      Time
      {
        Key {float {0.0, 0.5, 1.0, 1.5, 2.0}}
      }

      Value
      {
        Key {float {0.0, 2.0, 4.0, 6.0, 8.0}}
      }
    }
  }
}
```

A track always contains a `Time` structure and a `Value` structure, and the data points contained inside those structures can be interpolated in a variety of ways. See the `Track` structure for information about the different types of key value interpolation.

Properties

The properties listed in the following table may be specified for an `Animation` structure.

Property	Type	Default	Description
<code>clip</code>	<code>uint32</code>	0	The clip index for the animation.
<code>begin</code>	<code>float</code>	–	The time at which the animation begins.
<code>end</code>	<code>float</code>	–	The time at which the animation ends.

The `clip` property specifies the animation clip index. The set of all `Animation` structures in an OpenGEX file having the same clip index constitute one complete animation clip for the entire scene. If a structure contains any animation at all, then it is not a requirement that the same structure contains an animation corresponding to each clip index present somewhere in the file. Information pertaining to the animation clip as a whole can be stored in a `Clip` structure.

The `begin` and `end` properties specify the times at which the animation begins and ends. The values of these properties are multiplied by the global time metric to obtain times measured in seconds. (See the `Metric` structure.) If either property is not specified, then the `begin` and/or `end` time for the animation is determined by the earliest and latest time values present in the `Track` structures belonging to the animation. An animation track may include key times that lie outside the interval specified by the `begin` and `end` properties, and the corresponding keys could still be used to calculate interpolated values inside the animation's time range.

A structure may contain multiple `Animation` structures belonging to the same animation clip (that is, with the same value for the `clip` property). However, each track belonging to the set of `Animation` structures having the same clip index must have a unique target structure, as given by the `target` property of the `Track` structure, among all tracks belonging to all animations in the set.

Structure Data

The following structures may compose the data stored inside an `Animation` structure.

Substructure	Min	Max	Description
<code>Track</code>	1	–	An <code>Animation</code> structure must contain one or more tracks that each hold animation keys for a single target.

Hierarchy

An Animation structure may be contained inside the following structures.

Containing Structure	Description
Node BoneNode GeometryNode CameraNode LightNode	Animation structures can be contained inside any node structure.
Texture	Animation structures can be contained inside a Texture structure.

Atten

The `Atten` structure specifies an attenuation function for a light object. A light source may have multiple attenuation functions applied to it, and the values produced by all of them are multiplied together to determine the intensity of light reaching any particular point in space.

Properties

The properties listed in the following table may be specified for an `Atten` structure.

Property	Type	Default	Description
<code>kind</code>	string	"distance"	The kind of attenuation.
<code>curve</code>	string	"linear"	The function defining the attenuation curve.

The `kind` property specifies the input to the attenuation function and must have one of the following values.

- A value of "distance" indicates that the input to the attenuation function is the radial distance from the light source.
- A value of "angle" indicates that the input to the attenuation function is the angle formed between the negative z axis and the direction to the point being illuminated in object space.
- A value of "cos_angle" indicates that the input to the attenuation function is the cosine of the angle formed between the negative z axis and the direction to the point being illuminated in object space.

The `curve` property determines the general formula for the attenuation function and must have one of the following values.

- A value of "linear" indicates that the attenuation is given by the linear function

$$a_{\text{linear}}(t) = \text{sat}\left(\frac{e-t}{e-b}\right),$$

where b is the input value at which the attenuation begins, and e is the input value at which the attenuation ends. The beginning and ending values are specified by parameters as described below. The default beginning value is 0.0, and the default ending value is 1.0.

- A value of "smooth" indicates that the attenuation is given by the cubic smooth-step function

$$a_{\text{smooth}}(t) = 3(a_{\text{linear}}(t))^2 - 2(a_{\text{linear}}(t))^3,$$

which is dependent on the same beginning and ending input values b and e as the linear attenuation function.

- A value of "inverse" indicates that the attenuation is given by the inverse function

$$a_{\text{inverse}}(t) = \text{sat}\left(\frac{s}{k_l t + k_c s} + o\right),$$

where s and o are scale and offset values specified by parameters as described below. The default scale value is 1.0, and the default offset value is 0.0. The linear coefficient k_l and constant coefficient k_c are also specified by parameters, and they have the default values 1.0 and 0.0, respectively.

- A value of "inverse_square" indicates that the attenuation is given by the inverse square function

$$a_{\text{inverse_square}}(t) = \text{sat}\left(\frac{s^2}{k_q t^2 + k_l s t + k_c s^2} + o\right),$$

where s and o are scale and offset values specified by parameters as described below. The default scale value is 1.0, and the default offset value is 0.0. The quadratic coefficient k_q , linear coefficient k_l , and constant coefficient k_c are also specified by parameters, and they have the default values 1.0, 0.0, and 0.0, respectively. (Note that the default value of k_l is different for the inverse and inverse square attenuation functions.)

Structure Data

The following structures may compose the data stored inside an Atten structure.

Substructure	Min	Max	Description
Param	0	–	An Atten structure may contain parameters.

For each Param substructure that is present, the value of its `attrib` property determines the meaning of the parameter data. The following parameter `attrib` values are defined by this specification for use with an attenuation function.

- A value of "begin" or "end" indicates that the parameter is the distance from the light source at which the attenuation function begins or ends. These are multiplied by the global distance metric to obtain values measured in meters. (See the Metric structure.) These parameter values apply only if the curve property has a value of "linear" or "smooth".
- A value of "scale" indicates that the parameter is a scale to be used with inverse distance attenuation functions. This is multiplied by the global distance metric to obtain a value measured in meters. (See the Metric structure.) This parameter applies only if the curve property has a value of "inverse" or "inverse_square".
- A value of "offset" indicates that the parameter is an offset to be used with inverse distance attenuation functions. This parameter applies only if the curve property has a value of "inverse" or "inverse_square". (Note that the offset is *not* multiplied by the global distance metric.)
- A value of "constant", "linear", or "quadratic" indicates that the parameter is the coefficient k_c , k_l , or k_q , respectively, used with inverse distance attenuation functions. The k_c and k_l parameters

apply only if the curve property has a value of "inverse" or "inverse_square", and the k_q parameter applies only if the curve property has a value of "inverse_square".

- A value of "power" indicates that the parameter is a power to be used with angular attenuation functions. This parameter applies only if the kind property of the Atten structure has a value of "angle" or "cos_angle". The result produced by the attenuation function should be raised to the value of the power parameter.

A writer may include Param substructures with application-defined attrib values. If a reader encounters any of these for which the attrib value is either unsupported or unrecognized, then the substructure should be ignored.

Hierarchy

An Atten structure may be contained inside the following structures.

Containing Structure	Description
LightObject	Atten structures can be contained inside a LightObject structure.

BoneCountArray

The BoneCountArray structure contains bone count data for a skinned mesh. For each vertex belonging to a mesh, the bone count array specifies the number of bones the influence the vertex. See the Skin structure for details about skinning calculations.

Structure Data

The following structures may compose the data stored inside a BoneCountArray structure.

Substructure	Min	Max	Description
uint8 uint16 uint32 uint64	1	1	A BoneCountArray structure must contain an array of per-vertex bone counts.

The size of the bone count array must match the number of vertices specified in the containing mesh's VertexArray structures.

Hierarchy

A BoneCountArray structure may be contained inside the following structures.

Containing Structure	Description
Skin	A single BoneCountArray structure can be contained inside a Skin structure.

BoneIndexArray

The BoneIndexArray structure contains bone index data for a skinned mesh. For each vertex belonging to a mesh, the bone index array contains n entries, where n is the number of bones influencing the vertex. See the Skin structure for details about skinning calculations.

Structure Data

The following structures may compose the data stored inside a BoneIndexArray structure.

Substructure	Min	Max	Description
uint8 uint16 uint32 uint64	1	1	A BoneIndexArray structure must contain an array of bone indexes.

The total number of entries in a bone index array is equal to the sum of the counts specified in the BoneCountArray structure for the same skin.

Hierarchy

A BoneIndexArray structure may be contained inside the following structures.

Containing Structure	Description
Skin	A single BoneIndexArray structure can be contained inside a Skin structure.

BoneNode

The BoneNode structure represents a single bone node in the scene. Because it is a specific type of node, it possesses all of the characteristics of a generic node, such as an optional name, transform, and animation. See the Node structure for more information.

The collection of bone nodes forming the complete skeleton for a skinned mesh is referenced by a BoneRefArray structure contained inside a Skeleton structure.

Structure Data

The following structures may compose the data stored inside a BoneNode structure.

Substructure	Min	Max	Description
Name	0	1	A BoneNode structure may have a name.
Transform Translation Rotation Scale	0	–	A BoneNode structure may have any number of transformations applied to it.
Animation	0	–	A BoneNode structure may contain animation.
Node BoneNode GeometryNode CameraNode LightNode	0	–	A BoneNode structure may have any number of subnodes.

Hierarchy

A BoneNode structure may be contained inside the following structures.

Containing Structure	Description
–	BoneNode structures can be top-level structures.
Node BoneNode GeometryNode CameraNode LightNode	BoneNode structures can be contained inside any other node structure.

BoneRefArray

The BoneRefArray structure contains the list of bone nodes belonging to a skeleton. See the Skeleton structure for more information.

Structure Data

The following structures may compose the data stored inside a BoneRefArray structure.

Substructure	Min	Max	Description
ref	1	1	A BoneRefArray structure must contain an array of references to BoneNode structures.

Hierarchy

A BoneRefArray structure may be contained inside the following structures.

Containing Structure	Description
Skeleton	A single BoneRefArray structure can be contained inside a Skeleton structure.

BoneWeightArray

The BoneWeightArray structure contains bone weight data for a skinned mesh. For each vertex belonging to a mesh, the bone weight array contains n entries, where n is the number of bones influencing the vertex. See the Skin structure for details about skinning calculations.

Structure Data

The following structures may compose the data stored inside a BoneWeightArray structure.

Substructure	Min	Max	Description
float	1	1	A BoneWeightArray structure must contain an array of bone weights.

The total number of entries in a bone weight array is equal to the sum of the counts specified in the BoneCountArray structure for the same skin.

Hierarchy

A BoneWeightArray structure may be contained inside the following structures.

Containing Structure	Description
Skin	A single BoneWeightArray structure can be contained inside a Skin structure.

CameraNode

The CameraNode structure represents a single camera node in the scene. Because it is a specific type of node, it possesses all of the characteristics of a generic node, such as an optional name, transform, and animation. See the Node structure for more information.

Object Reference

A camera node must contain an ObjectRef structure that references a CameraObject structure. The camera object contains the information necessary to construct the properly configured camera.

Structure Data

The following structures may compose the data stored inside a CameraNode structure.

Substructure	Min	Max	Description
Name	0	1	A CameraNode structure may have a name.
ObjectRef	1	1	A CameraNode structure must contain a reference to a CameraObject structure.
Transform Translation Rotation Scale	0	–	A CameraNode structure may have any number of transformations applied to it.
Animation	0	–	A CameraNode structure may contain animation.
Node BoneNode GeometryNode CameraNode LightNode	0	–	A CameraNode structure may have any number of subnodes.

Hierarchy

A CameraNode structure may be contained inside the following structures.

Containing Structure	Description
–	CameraNode structures can be top-level structures.
Node BoneNode GeometryNode CameraNode LightNode	CameraNode structures can be contained inside any other node structure.

CameraObject

The `CameraObject` structure contains data for a camera object. Multiple `CameraNode` structures may reference a single camera object, and this allows a scene to contain multiple instances of the same camera with different transforms.

In object space, a camera points in the direction of the negative z axis.

Structure Data

The following structures may compose the data stored inside a `CameraObject` structure.

Substructure	Min	Max	Description
Param	0	–	A <code>CameraObject</code> structure may contain parameters.

For each `Param` substructure that is present, the value of its `attrib` property determines the meaning of the parameter data. The following parameter `attrib` values are defined by this specification for use with a camera object.

- A value of "fov" or "fov x " indicates that the parameter is the horizontal field-of-view angle. This is multiplied by the global angle metric to obtain a value measured in radians. (See the `Metric` structure.)
- A value of "fovy" indicates that the parameter is the vertical field-of-view angle. This is multiplied by the global angle metric to obtain a value measured in radians. (See the `Metric` structure.)
- A value of "near" or "far" indicates that the parameter is the positive distance to the near plane or far plane. These are multiplied by the global distance metric to obtain values measured in meters. (See the `Metric` structure.)

If any of the above parameters are not present, then their default values are application-defined.

A writer may include `Param` substructures with application-defined `attrib` values. If a reader encounters any of these for which the `attrib` value is either unsupported or unrecognized, then the substructure should be ignored.

Hierarchy

A `CameraObject` structure may be contained inside the following structures.

Containing Structure	Description
–	<code>CameraObject</code> structures must be top-level structures.

Clip

The `Clip` structure holds information about an animation clip. The collection of all `Animation` structures in an OpenGEX file having the same value for the `clip` property constitutes a complete animation clip. A `Clip` structure having the matching value for its `index` property can specify a name and frame rate for that animation clip. `Clip` structures always appear as top-level structures.

Name

If a `Clip` structure contains a `Name` structure, then it defines the externally-visible name of the animation clip that should be displayed to the user in applications such as a level editor.

Properties

The properties listed in the following table may be specified for a `Clip` structure.

Property	Type	Default	Description
<code>index</code>	<code>uint32</code>	0	The clip index.

The `index` property specifies the animation clip to which the `Clip` structure pertains. This index is matched to the values of the `clip` properties of the `Animation` structures contained throughout the OpenGEX file.

Structure Data

The following structures may compose the data stored inside a `Clip` structure.

Substructure	Min	Max	Description
<code>Name</code>	0	1	A <code>Clip</code> structure may have a name.
<code>Param</code>	0	–	A <code>Clip</code> structure may contain any number of parameters.

For each `Param` substructure that is present, the value of its `attrib` property determines the meaning of the parameter data. The following parameter `attrib` value is defined by this specification for use with an attenuation function.

- A value of "rate" indicates that the parameter is the frame rate at which the animation is intended to play. The parameter value is divided by the global time metric to obtain a value measured in frames per second. (See the `Metric` structure.)

A writer may include `Param` substructures with application-defined `attrib` values. If a reader encounters any of these for which the `attrib` value is either unsupported or unrecognized, then the substructure should be ignored.

Hierarchy

A Clip structure may be contained inside the following structures.

Containing Structure	Description
–	Clip structures must be top-level structures.

Color

The `Color` structure holds a single color value.

The red, green, and blue components of a color may need to be converted to the global color space. (See the `Metric` structure.)

Properties

The properties listed in the following table may be specified for a `Color` structure.

Property	Type	Default	Description
<code>attrib</code>	<code>string</code>	–	The color attribute.

The `attrib` property is required, and it specifies the meaning of the color. See the containing structures for information about the specific types of attributes that are defined.

Structure Data

The following structures may compose the data stored inside a `Color` structure.

Substructure	Min	Max	Description
<code>float[3]</code> <code>float[4]</code>	1	1	A <code>Color</code> structure must contain an RGB color or RGBA color in a <code>float</code> substructure having the corresponding array size.

Hierarchy

A `Color` structure may be contained inside the following structures.

Containing Structure	Description
<code>Material</code>	<code>Color</code> structures can be contained inside a <code>Material</code> structure to specify material attributes.
<code>LightObject</code>	A single <code>Color</code> structure can be contained inside a <code>LightObject</code> structure to specify the light's color.

GeometryNode

The `GeometryNode` structure represents a single geometry node in the scene. Because it is a specific type of node, it possesses all of the characteristics of a generic node, such as an optional name, transform, and animation. See the `Node` structure for more information.

Object Reference

A geometry node must contain an `ObjectRef` structure that references a `GeometryObject` structure. The geometry object contains all of the required mesh data and optional skinning data.

Material References

A geometry node may contain one or more `MaterialRef` structures that reference `Material` structures. The `index` property of each material reference specifies to which part a mesh the material is applied by matching it with the `material` property of each `IndexArray` structure in the mesh.

Morph Weights

If the geometry object referenced by the geometry node contains vertex data for multiple morph targets, then the geometry node may contain one or more `MorphWeight` structures that specify the blending weight for each target. When morph weights are present, each `MorphWeight` structure may be the target of a `Track` structure in the animation belonging to the geometry node.

Properties

The properties listed in the following table may be specified for a `GeometryNode` structure to control various rendering options that may be supported by an application.

Property	Type	Default	Description
<code>visible</code>	<code>bool</code>	–	Whether the geometry is visible.
<code>shadow</code>	<code>bool</code>	–	Whether the geometry casts shadows.
<code>motion_blur</code>	<code>bool</code>	–	Whether the geometry is rendered with motion blur.

If any of the `visible`, `shadow`, or `motion_blur` properties is specified, then each overrides the corresponding property belonging to the referenced `GeometryObject` structure.

Structure Data

The following structures may compose the data stored inside a GeometryNode structure.

Substructure	Min	Max	Description
Name	0	1	A GeometryNode structure may have a name.
ObjectRef	1	1	A GeometryNode structure must contain a reference to a GeometryObject structure.
MaterialRef	0	–	A GeometryNode structure may contain references to Material structures.
MorphWeight	0	–	A GeometryNode structure may contain morph weights.
Transform Translation Rotation Scale	0	–	A GeometryNode structure may have any number of transformations applied to it.
Animation	0	–	A GeometryNode structure may contain animation.
Node BoneNode GeometryNode CameraNode LightNode	0	–	A GeometryNode structure may have any number of subnodes.

Hierarchy

A GeometryNode structure may be contained inside the following structures.

Containing Structure	Description
–	GeometryNode structures can be top-level structures.
Node BoneNode GeometryNode CameraNode LightNode	GeometryNode structures can be contained inside any other node structure.

GeometryObject

The GeometryObject structure contains data for a geometry object. Multiple GeometryNode structures may reference a single geometry object, and this allows a scene to contain multiple instances of the same geometry with different transforms and materials.

Properties

The properties listed in the following table may be specified for a GeometryObject structure to control various rendering options that may be supported by an application.

Property	Type	Default	Description
visible	bool	true	Whether the geometry is visible.
shadow	bool	true	Whether the geometry casts shadows.
motion_blur	bool	true	Whether the geometry is rendered with motion blur.

If the visible property is false, then the geometry should not be rendered but should still participate in collision detection, if applicable.

If the shadow property is false, then the geometry should not cast shadows, if supported by the application.

If the motion_blur property is false, then the geometry should not be rendered with motion blur, if supported by the application.

The visible, shadow, and motion_blur properties can be overridden by any GeometryNode structure referencing the GeometryObject structure. The value of any one of these properties takes effect for a particular geometry node only when the same property is not specified for the geometry node.

Structure Data

The following structures may compose the data stored inside a GeometryObject structure.

Substructure	Min	Max	Description
Mesh	1	–	A GeometryObject structure must contain one or more meshes.
Morph	0	–	A GeometryObject structure may contain any number of Morph structures.

A geometry object contains one Mesh structure for each level of detail.

A geometry object may contain a Morph structure for each morph target for which vertex data exists inside the Mesh structures.

Hierarchy

A GeometryObject structure may be contained inside the following structures.

Containing Structure	Description
–	GeometryObject structures must be top-level structures.

IndexArray

The `IndexArray` structure contains index array data for a mesh. See the `Mesh` structure for information about how arrays are used in a mesh.

Properties

The properties listed in the following table may be specified for an `IndexArray` structure.

Property	Type	Default	Description
<code>material</code>	<code>uint32</code>	0	The material index for the primitives constructed from the index array.
<code>restart</code>	<code>uint64</code>	-	The primitive restart index for triangle strips.
<code>front</code>	<code>string</code>	"ccw"	Whether front faces are wound clockwise or counterclockwise.

The `material` property specifies a material index for the list of primitives contained in the `IndexArray` structure. The actual set of materials to be assigned to each list of primitives is specified by the `MaterialRef` structures contained in the `GeometryNode` structure that references the `GeometryObject` structure containing the index array as part of a mesh. The value of the `index` property for each `MaterialRef` structure is matched to the value of the index array's `material` property.

The `restart` property can only be specified when the `primitive` property of the `Mesh` structure containing the `IndexArray` structure is either `"line_strip"` or `"triangle_strip"`. If this property is present, then its value defines the index that signals the end of a strip. When this index is encountered, it does not cause a new vertex to be added to the current strip but instead starts a new strip with the next index. If the `restart` property is not specified at all, then there is no index that causes a new strip to be started.

The `front` property must be either `"cw"` or `"ccw"`, and it specifies whether front-facing primitives are wound clockwise or counterclockwise, respectively. A reader is free to reorder indexes to follow its own conventions or to enforce a consistent winding direction.

Structure Data

The following structures may compose the data stored inside an IndexArray structure.

Substructure	Min	Max	Description
uint8 uint16 uint32 uint64 uint8[2] uint16[2] uint32[2] uint64[2] uint8[3] uint16[3] uint32[3] uint64[3] uint8[4] uint16[4] uint32[4] uint64[4]	1	1	An IndexArray structure must contain an array of vertex indexes, possibly grouped in subarrays whose size corresponds to the type of geometric primitive specified by the containing mesh. For points, line strips, and triangle strips, there are no subarrays. For independent lines, the subarray size must be 2, for independent triangles, the subarray size must be 3, and for independent quads, the subarray size must be 4.

Hierarchy

An IndexArray structure may be contained inside the following structures.

Containing Structure	Description
Mesh	IndexArray structures can be contained inside a Mesh structure.

Key

The Key structure contains key data for an animation track.

Properties

The properties listed in the following table may be specified for a Key structure.

Property	Type	Default	Description
kind	string	"value"	The kind of data.

The kind property specifies the type of key data and must have one of the following values.

- A value of "value" indicates that the data contained in the Key structure provides the actual values of the keys. If the Key structure is contained inside a Time structure, then the key values are times. (Times are multiplied by the global time scale to obtain seconds. See the Metric structure.) Otherwise, if the Key structure is contained inside a Value structure, then the key values are coordinate values, rotation angles, etc., depending on the target of the animation track.
- A value of "-control" or "+control" indicates that the data contained in the Key structure provides the incoming or outgoing control points for the keys, respectively. Control point data is valid only inside Time and Value structures having a curve property value of "bezier".
- A value of "tension", "continuity", or "bias" indicates that the data contained in the Key structure provides the tension, continuity, or bias parameters for the keys. This data is valid only inside Value structures having a curve property value of "tcb".

Structure Data

The following structures may compose the data stored inside a Key structure.

Substructure	Min	Max	Description
float float[3] float[4] float[16]	1	1	A Key structure must contain an array of floating-point key values.

For Key structures contained inside a Time structure, the data is always scalar, so the substructure must be of type float with no array size.

For Key structures contained inside a Value structure, the data type must match the dimensionality of the data stored in the target of the animation track when the key's kind property has a value of "value", "-control", or "+control". The data type is always scalar when the kind property has a value of

"tension", "continuity", or "bias". See the Track structure for information about the meaning of the key data and how it is used to calculate interpolated values.

Hierarchy

A Key structure may be contained inside the following structures.

Containing Structure	Description
Time Value	Key structures can be contained inside Time and Value structures belonging to an animation track.

LightNode

The `LightNode` structure represents a single light node in the scene. Because it is a specific type of node, it possesses all of the characteristics of a generic node, such as an optional name, transform, and animation. See the `Node` structure for more information.

Object Reference

A light node must contain an `ObjectRef` structure that references a `LightObject` structure. The light object contains the information necessary to construct the proper type of light source.

Properties

The properties listed in the following table may be specified for a `LightNode` structure.

Property	Type	Default	Description
shadow	bool	–	Whether the light casts shadows.

If the shadow property is specified, then it overrides the value of the shadow property belonging to the referenced `LightObject` structure.

Structure Data

The following structures may compose the data stored inside a `LightNode` structure.

Substructure	Min	Max	Description
Name	0	1	A <code>LightNode</code> structure may have a name.
ObjectRef	1	1	A <code>LightNode</code> structure must contain a reference to a <code>LightObject</code> structure.
Transform Translation Rotation Scale	0	–	A <code>LightNode</code> structure may have any number of transformations applied to it.
Animation	0	–	A <code>LightNode</code> structure may contain animation.
Node BoneNode GeometryNode CameraNode LightNode	0	–	A <code>LightNode</code> structure may have any number of subnodes.

Hierarchy

A LightNode structure may be contained inside the following structures.

Containing Structure	Description
–	LightNode structures can be top-level structures.
Node BoneNode GeometryNode CameraNode LightNode	LightNode structures can be contained inside any other node structure.

LightObject

The `LightObject` structure contains data for a light object. Multiple `LightNode` structures may reference a single light object, and this allows a scene to contain multiple instances of the same light with different transforms.

Properties

The properties listed in the following table may be specified for a `LightObject` structure.

Property	Type	Default	Description
type	string	–	The type of light.
shadow	bool	true	Whether the light casts shadows.

The `type` property is required and defines the type of light being described by the light object. It must specify one of the following values.

- A value of "infinite" indicates that the light source is to be treated as if it were infinitely far away so that its rays are parallel. In object space, the rays of an infinite light point in the direction of the negative z axis.
- A value of "point" indicates that the light source is a point light that radiates in all directions.
- A value of "spot" indicates that the light source is a spot light that radiates from a single point but in a limited range of directions. In object space, the primary direction of radiation for a spot light is the negative z axis.

If the `shadow` property is `false`, then the light source should not cast shadows, if supported by the application. The `shadow` property can be overridden by any `LightNode` structure referencing the `LightObject` structure. The value of this property takes effect for a particular light node only when the same property is not specified for the light node.

Structure Data

The following structures may compose the data stored inside a `LightObject` structure.

Substructure	Min	Max	Description
Color	0	1	A <code>LightObject</code> structure may have a color.
Param	0	1	A <code>LightObject</code> structure may have a luminous intensity parameter (point lights and spot lights) or an illuminance parameter (infinite lights).
Spectrum	0	1	A <code>LightObject</code> structure may have an emission spectrum.
Texture	0	1	A <code>LightObject</code> structure may have a projected texture.
Atten	0	–	A <code>LightObject</code> structure may have any number of attenuation functions applied to it.

For each `Color` substructure that is present, the value of its `attrib` property determines the meaning of the color data. The following color `attrib` value is defined by this specification for use with a light object.

- A value of "light" indicates that the color is the main color of light emitted by the light source. If this color is not present and no `Spectrum` substructure with the `attrib` property set to "light" is present, then the default light color should be white with RGB value (1.0, 1.0, 1.0).

For each `Param` substructure that is present, the value of its `attrib` property determines the meaning of the parameter data. The following parameter `attrib` values are defined by this specification for use with a light object.

- A value of "intensity" indicates that the parameter is the luminous intensity of a point light or spot light. This value is measured in candela (cd), which is equivalent to lumens per steradian, and it is *not* affected by the global angle metric. If this parameter is not present, then the default intensity is 1.0 cd. This value of the `attrib` property does not apply to infinite lights.
- A value of "illuminance" indicates that the parameter is the constant illuminance of an infinite light. This value is measured in lux (lx), which is equivalent to lumens per square meter, and it is *not* affected by the global distance metric. If this parameter is not present, then the default illuminance is 1.0 lx. This value of the `attrib` property does not apply to point lights or spot lights.

For each `Spectrum` substructure that is present, the value of its `attrib` property determines the meaning of the color data. The following color `attrib` value is defined by this specification for use with a light object.

- A value of "light" indicates that the spectrum is the main emission spectrum of the light source. If a `Color` substructure with the `attrib` property set to "light" is also present, then the `Spectrum` substructure takes precedence.

For each Texture substructure that is present, the value of its `attrib` property determines the meaning of the texture map. The following texture `attrib` value is defined by this specification for use with a light object.

- A value of "projection" indicates that the texture map is a spot light projection. The texture map should be oriented so that right direction is aligned to the object-space positive x axis and the up direction is aligned to the object-space positive y axis.

A writer may include `Color`, `Param`, `Spectrum`, and `Texture` substructures with application-defined `attrib` values. If a reader encounters any of these for which the `attrib` value is either unsupported or unrecognized, then the substructure should be ignored.

The `Atten` substructures determine the attenuation function for a point light or spot light. The attenuation function has units of steradians per square meter ($\text{sr} \cdot \text{m}^{-2}$) and transforms the luminous intensity of the light source, measured in candela, into the illuminance at a particular location, measured in lux. Because the brightness of an infinite light is specified directly as an illuminance, attenuation does not apply to infinite lights, and any `Atten` substructures present should be ignored.

Hierarchy

A `LightObject` structure may be contained inside the following structures.

Containing Structure	Description
–	<code>LightObject</code> structures must be top-level structures.

Material

The `Material` structure contains information about a material. `Material` structures are referenced by geometry nodes through `MaterialRef` structures belonging to `GeometryNode` structures.

Name

If a `Material` structure contains a `Name` structure, then it defines the externally-visible name of the material that should be displayed to the user in applications such as a level editor. (This name should not be confused with an `OpenDDL` name that could be assigned to the `Material` structure in an `OpenGEX` file.)

Properties

The properties listed in the following table may be specified for a `Material` structure.

Property	Type	Default	Description
<code>two_sided</code>	<code>bool</code>	<code>false</code>	Whether the material is two-sided.

If the `two_sided` property is true, then the geometry to which the material is applied should be rendered two-sided without any back-face culling.

Structure Data

The following structures may compose the data stored inside a `Material` structure.

Substructure	Min	Max	Description
<code>Name</code>	0	1	A <code>Material</code> structure may have a name.
<code>Color</code>	0	–	A <code>Material</code> structure may contain any number of colors.
<code>Param</code>	0	–	A <code>Material</code> structure may contain any number of parameters.
<code>Spectrum</code>	0	–	A <code>Material</code> structure may contain any number of spectra.
<code>Texture</code>	0	–	A <code>Material</code> structure may contain any number of textures.

For each `Color`, `Param`, `Spectrum`, and `Texture` substructure that is present, the value of its `attrib` property determines the meaning of the data it contains. The `attrib` values listed in Table 2.1 are defined by this specification for use with a material. Each `attrib` value may be assigned only to the specific types of substructures indicated by check marks in the table under the columns C, P, S, and T representing `Color`, `Param`, `Spectrum`, and `Texture`.

This specification does not require any particular implementation of the material attributes. A reader is free to apply attributes using any preferred rendering techniques, and it may ignore attributes that are not supported. The default values used by a reader for any attributes that are not present must match the default values listed in Table 2.1.

Whenever a `Texture` substructure corresponding to a scalar-valued material attribute is present, the first channel of the texture map, after a possible swizzle, always specifies the attribute's value. The channels of a texture map can be permuted by the `swizzle` property of the `Texture` structure.

If both `Color` and `Spectrum` substructures having the same `attrib` property value appear in a material, then the `Spectrum` substructure takes precedence, and the `Color` substructure is used only if the reader does not support spectra.

If both `Color` and `Texture` substructures having the same `attrib` property value appear in a material, then they have a multiplicative effect. Likewise, If both `Param` and `Texture` substructures having the same `attrib` property value appear in a material, then they have a multiplicative effect.

Normal maps must contain tangent-space normal vectors. In the case that a normal map's format has three or more channels, the red, green, and blue channels correspond to the x , y , and z components of the normal vector. In the case that a normal map's format has two channels, the red and green channels correspond to the x and y components of the normal vector, and the z component is given by $\sqrt{x^2 + y^2}$.

If a height map is present, it may be used to construct a normal map or additional maps containing information such as parallax, horizon, or ambient occlusion data. Height values read from a height map must be normalized to the range $[0, 1]$ and then multiplied by the height scale provided by a `Param` substructure having the `attrib` value "height_scale" (if present).

A writer may include `Color`, `Param`, `Spectrum`, and `Texture` substructures with application-defined `attrib` values. If a reader encounters any of these for which the `attrib` value is either unsupported or unrecognized, then the substructure should be ignored.

Hierarchy

A `Material` structure may be contained inside the following structures.

Containing Structure	Description
–	Material structures must be top-level structures.

Attribute	C	P	S	T	Description
"diffuse"	✓		✓	✓	Diffuse reflection color, base color, or albedo. The default is the RGB color (1.0, 1.0, 1.0).
"specular"	✓		✓	✓	Specular reflection color. The default is the RGB color (0.0, 0.0, 0.0).
"specular_power"		✓		✓	Specular power applied to $\mathbf{N} \cdot \mathbf{H}$. The default is 1.0.
"emission"	✓		✓	✓	Emission color. The default is the RGB color (0.0, 0.0, 0.0).
"opacity"		✓		✓	Base opacity. The default is 1.0.
"transparency"	✓		✓	✓	Base transparency color. The default is the RGB color (0.0, 0.0, 0.0).
"normal"				✓	Base normal map.
"height"				✓	Base height map.
"height_scale"		✓			Scale applied to height map. The default is 1.0.
"occlusion"				✓	Precomputed ambient occlusion map.
"roughness"		✓		✓	Base surface roughness in the range 0.0 to 1.0. The default is 0.0.
"metalness"		✓		✓	Metalness in the range 0.0 to 1.0. The default is 0.0.
"clearcoat"	✓		✓	✓	Clearcoat color. The default is the RGB color (0.0, 0.0, 0.0).
"clearcoat_roughness"		✓		✓	Clearcoat surface roughness in the range 0.0 to 1.0. The default is 0.0.
"clearcoat_normal"				✓	Clearcoat normal map.
"sheen"	✓		✓	✓	Sheen color. The default is the RGB color (0.0, 0.0, 0.0).
"sheen_roughness"		✓		✓	Sheen surface roughness in the range 0.0 to 1.0. The default is 0.0.
"ior"		✓			Index of refraction. The default is 1.0.

Table 2.1. This table describes the attrib property values of Color, Param, Spectrum, and Texture substructures that may appear inside a Material structure. Each attribute value may be applied only to the specific types of substructures for which a check mark appears in the columns C, P, S, and T.

MaterialRef

The `MaterialRef` structure holds a reference to a `Material` structure.

Properties

The properties listed in the following table may be specified for a `MaterialRef` structure.

Property	Type	Default	Description
index	uint32	0	The material index.

The `index` property specifies the material index to which the referenced material is bound. This index is matched to the values of the `material` properties of the `IndexArray` structures contained in the meshes belonging to the `GeometryObject` referenced by the `GeometryNode` structure containing the `MaterialRef` structure.

Structure Data

The following structures may compose the data stored inside a `MaterialRef` structure.

Substructure	Min	Max	Description
ref	1	1	A <code>MaterialRef</code> structure must contain a reference to a <code>Material</code> structure.

Hierarchy

A `MaterialRef` structure may be contained inside the following structures.

Containing Structure	Description
<code>GeometryNode</code>	Multiple <code>MaterialRef</code> structures can be contained inside a <code>GeometryNode</code> structure, but each must have a different value for the <code>index</code> property.

Mesh

The Mesh structure contains data for a single geometric mesh, and a GeometryObject structure contains one mesh for each level of detail. Each mesh typically contains several arrays of per-vertex data and one or more index arrays as shown in Listing 2.2.

A mesh may contain vertex data for multiple morph targets. The morph target to which each vertex array belongs is determined by the value of its morph property. See the VertexArray structure for details about determining which vertex arrays belong to each morph target.

A mesh may also contain a single Skin structure that holds the skeleton and bone influence data needed for skinning.

Listing 2.2. This mesh structure contains per-vertex positions, normals, and texture coordinates, and it contains an index array that determines how triangle primitives are assembled.

```
Mesh (primitive = "triangles")
{
  VertexArray (attrib = "position")
  {
    float[3] {...}
  }

  VertexArray (attrib = "normal")
  {
    float[3] {...}
  }

  VertexArray (attrib = "texcoord")
  {
    float[2] {...}
  }

  IndexArray (material = 0)
  {
    uint16[3] {...}
  }
}
```

Properties

The properties listed in the following table may be specified for a Mesh structure.

Property	Type	Default	Description
lod	uint32	0	The level of detail. A value of 0 corresponds to the highest level of detail.
primitive	string	"triangles"	The primitive type.

The lod property specifies the level of detail to which the mesh corresponds. A GeometryObject structure may contain any number of Mesh structures as long as they each have a unique level of detail. The highest level of detail is number 0, and successively lower levels of detail count upward.

The primitive property specifies the type of geometric primitive used by the mesh. It must have one of the values shown in Table 2.2, and it must be the same value for each level of detail.

Primitive	Description
"points"	The mesh is composed of a set of independent points. The number of points is n , and point i is given by vertex i .
"lines"	The mesh is composed of a set of independent lines. The number of lines is $n/2$, and line i is composed of vertices $2i$ and $2i + 1$.
"line_strip"	The mesh is composed of one or more line strips. The number of lines is $n - 1$, and line i is composed of vertices i and $i + 1$.
"triangles"	The mesh is composed of a set of independent triangles. The number of triangles is $n/3$, and triangle i is composed of vertices $3i$, $3i + 1$, and $3i + 2$.
"triangle_strip"	The mesh is composed of one or more triangle strips. The number of triangles is $n - 2$, and triangle i is composed of vertices i , $i + 1$, and $i + 2$ when i is even or vertices i , $i + 2$, and $i + 1$ when i is odd, in the orders listed.
"quads"	The mesh is composed of a set of individual quads. The number of quads is $n/4$, and quad i is composed of vertices $4i$, $4i + 1$, $4i + 2$, and $4i + 3$.

Table 2.2. This table describes the geometric primitives corresponding to each possible value of the primitive property. The value of n is the total number of indexes if an IndexArray structure is present or the total number of vertices in each VertexArray structure, otherwise. Primitives are indexed by the letter i , starting at zero.

For line strips and triangle strips, the restart property of the IndexArray structure may be used to construct multiple independent strips.

Structure Data

The following structures may compose the data stored inside a Mesh structure.

Substructure	Min	Max	Description
VertexArray	1	–	A Mesh structure must contain one or more vertex arrays.
IndexArray	0	–	A Mesh structure may contain one or more index arrays.
Skin	0	1	A Mesh structure may contain skinning data.

All VertexArray structures belonging to a mesh must specify data for the same number of vertices.

Each IndexArray structure specifies how the vertices are assembled into geometric primitives. For lines, triangles, and quads, the index array contains subarrays that each specify the indexes of the vertices composing a single primitive. For points, line strips, and triangle strips, the index data is stored as a single array.

If a mesh does not contain an IndexArray structure, then it is as if an index array with the default properties existed and contained each index between 0 and $n - 1$ in order and grouped into subarrays as necessary, where n is the number of vertices in each VertexArray structure.

Hierarchy

A Mesh structure may be contained inside the following structures.

Containing Structure	Description
GeometryObject	Multiple Mesh structures can be contained inside a GeometryObject structure, but each must have a different lod value.

Metric

The `Metric` structure specifies global measurement and orientation properties such as the distance scale and up direction. It can also specify the chromaticities of the primary colors.

Every `Metric` structure in an OpenGEX file must precede all structures of any other type.

Up Direction

One of the properties that can be defined by the `Metric` structure is the global up direction, which can be the positive z axis or the positive y axis. If a reader's preferred up direction does not match the up direction used by an OpenGEX file, then most vectors and matrices need to be adjusted to account for the difference.

In the case that a reader's preferred up direction is the positive z axis and the OpenGEX file specifies the positive y axis, each 3D vector \mathbf{v} representing any position or direction should be transformed into a vector \mathbf{v}' using the formula

$$\mathbf{v}' = (v_x, -v_z, v_y),$$

and each 4×4 matrix \mathbf{M} representing any coordinate space transformation should be transformed into a matrix \mathbf{M}' using the formula

$$\mathbf{M}' = \begin{bmatrix} M_{11} & -M_{13} & M_{12} & M_{14} \\ -M_{31} & M_{33} & -M_{32} & -M_{34} \\ M_{21} & -M_{23} & M_{22} & M_{24} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

In the opposite case that a reader's preferred up direction is the positive y axis and the OpenGEX file specifies the positive z axis, each 3D vector \mathbf{v} representing any position or direction should be transformed into a vector \mathbf{v}' using the formula

$$\mathbf{v}' = (v_x, v_z, -v_y),$$

and each 4×4 matrix \mathbf{M} representing any coordinate space transformation should be transformed into a matrix \mathbf{M}' using the formula

$$\mathbf{M}' = \begin{bmatrix} M_{11} & M_{13} & -M_{12} & M_{14} \\ M_{31} & M_{33} & -M_{32} & M_{34} \\ -M_{21} & -M_{23} & M_{22} & -M_{24} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

These transforms affect the position, normal, tangent, and bitangent vectors stored in a `VertexArray` structure, all transforms applied to any node structure, the array of bind-pose transforms stored in a `Skeleton` structure, and the bind-pose transform stored in a `Skin` structure.

Color Space

The Metric structure can define the xy chromaticity coordinates of the red, green, and blue primary colors as well as the white point. If these are omitted, then the default values correspond to the sRGB color space.

If the chromaticities of the primary colors or white point do not match those of the reader's preferred color space, then color values specified inside Color structures must be converted by passing through the XYZ color space. Given chromaticity coordinates (x_R, y_R) , (x_G, y_G) , and (x_B, y_B) corresponding to the red, green, and blue primary colors and chromaticity coordinates (x_W, y_W) corresponding to the white point, the primary luminances Y_R , Y_G , and Y_B are determined by the calculation

$$\begin{bmatrix} Y_R \\ Y_G \\ Y_B \end{bmatrix} = \begin{bmatrix} x_R/y_R & x_G/y_G & x_B/y_B \\ 1 & 1 & 1 \\ z_R/y_R & z_G/y_G & z_B/y_B \end{bmatrix}^{-1} \begin{bmatrix} x_W/y_W \\ 1 \\ z_W/y_W \end{bmatrix},$$

where $z = 1 - x - y$ for each of the subscripts R , G , B , and W .

Any color (R, G, B) based on the specified red, green, and blue chromaticities is then converted to coordinates (X, Y, Z) in the XYZ color space by the formula

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} \frac{x_R}{y_R} Y_R & \frac{x_G}{y_G} Y_G & \frac{x_B}{y_B} Y_B \\ Y_R & Y_G & Y_B \\ \frac{z_R}{y_R} Y_R & \frac{z_G}{y_G} Y_G & \frac{z_B}{y_B} Y_B \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}.$$

The conversion from XYZ coordinates to an arbitrary RGB color space is accomplished by inverting this equation. In the case of the sRGB color space, a color (R, G, B) is given by

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 3.240970 & -1.537383 & -0.498611 \\ -0.969244 & 1.875968 & 0.041555 \\ 0.055630 & -0.203977 & 1.056972 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix},$$

where the chromaticity coordinates are defined as

$$(x_R, y_R) = (0.64, 0.33)$$

$$(x_G, y_G) = (0.30, 0.60)$$

$$(x_B, y_B) = (0.15, 0.06)$$

$$(x_W, y_W) = (0.3127, 0.3290)$$

and the primary luminances are thus calculated to be

$$Y_R = 0.212639$$

$$Y_G = 0.715169$$

$$Y_B = 0.072192.$$

Properties

The properties listed in the following table may be specified for a *Metric* structure.

Property	Type	Default	Description
key	string	–	The metric identifier.

The key property is required and specifies the type of metric being defined. It must specify one of the values shown in Table 2.3.

Key	Description
"distance"	The factor by which all distance values should be multiplied to obtain a value measured in meters.
"angle"	The factor by which all angle values should be multiplied to obtain a value measured in radians.
"time"	The factor by which all time values should be multiplied to obtain a value measured in seconds.
"up"	The world-space axis that corresponds to the up direction.
"forward"	The world-space axis that corresponds to the forward direction for models such as characters and vehicles.
"red"	The <i>xy</i> chromaticity coordinates of the red primary color.
"green"	The <i>xy</i> chromaticity coordinates of the green primary color.
"blue"	The <i>xy</i> chromaticity coordinates of the blue primary color.
"white"	The <i>xy</i> chromaticity coordinates of the white point.

Table 2.3. This table describes the meaning of each possible value of the key property.

Structure Data

The following structures may compose the data stored inside a *Metric* structure.

Substructure	Min	Max	Description
float float[2] string	1	1	A <i>Metric</i> structure must contain one data structure holding the value of the metric.

For the "distance", "angle", and "time" metrics, the data contained in the structure must be a single floating-point value. The default value for each of these three metrics is 1.0 if any are not specified in an OpenGEX file.

For the "up" metric, the data contained in the structure must be a single string value equal to either "y" or "z". The default value for the up direction is "z" if it is not specified in an OpenGEX file.

For the "forward" metric, the data contained in the structure must be a single string value equal to "x", "-x", "y", "-y", "z", or "-z". The default value for the forward direction is "x" if it is not specified in an OpenGEX file.

For the "red", "green", "blue", and "white" metrics, the data contained in the structure must be a single subarray of two floating-point values. If any of these metrics are not specified in an OpenGEX file, then the default value for the red primary chromaticity is (0.64, 0.33), the default value for the green primary chromaticity is (0.3, 0.6), the default value for the blue primary chromaticity is (0.15, 0.06), and the default value for the white point chromaticity is (0.3127, 0.329).

The default metrics would be defined in an OpenGEX file as shown in Listing 2.3.

Listing 2.3. These Metric structures define the default units of measurement and default directions.

```
Metric (key = "distance") {float {1.0}}
Metric (key = "angle") {float {1.0}}
Metric (key = "time") {float {1.0}}

Metric (key = "up") {string {"z"}}
Metric (key = "forward") {string {"x"}}

Metric (key = "red") {float[2] {{0.64, 0.33}}}
Metric (key = "green") {float[2] {{0.3, 0.6}}}
Metric (key = "blue") {float[2] {{0.15, 0.06}}}
Metric (key = "white") {float[2] {{0.3127, 0.329}}}
```

Hierarchy

A Metric structure may be contained inside the following structures.

Containing Structure	Description
—	Metric structures must be top-level structures.

Morph

The Morph structure holds information about a morph target belonging to a GeometryObject structure. See the VertexArray structure for a description of what vertex data constitutes a complete morph target.

Name

If a Morph structure contains a Name structure, then it defines the externally-visible name of the morph target that should be displayed to the user in applications such as a level editor.

Properties

The properties listed in the following table may be specified for a Morph structure.

Property	Type	Default	Description
index	uint32	0	The morph target index.
base	uint32	–	The base morph target index for a relative morph target.

The index property specifies the morph target index. This index is matched to the values of the morph properties of the VertexArray structures contained inside the same GeometryObject structure as the Morph structure.

The base property is optional and, if specified, indicates that the morph target is relative and intended to be applied as a difference with the morph target having the given base index. If the base property is not specified, then the morph target is absolute. Any morph target that does not have a corresponding Morph structure is absolute.

If a base index is specified to indicate a relative morph target, then the morph target to which the base index refers must be an absolute morph target.

See the MorphWeight structure for details about calculating vertex attributes for absolute and relative morph targets.

Structure Data

The following structures may compose the data stored inside a Morph structure.

Substructure	Min	Max	Description
Name	0	1	A Morph structure may have a name.

Hierarchy

A Morph structure may be contained inside the following structures.

Containing Structure	Description
GeometryObjectTrack	Multiple Morph structures can be contained inside a GeometryObject structure, but each must have a different value for the index property.

MorphWeight

The MorphWeight structure holds a single morph weight for a GeometryNode structure that references a GeometryObject structure containing vertex data for multiple morph targets.

A GeometryNode structure typically contains a MorphWeight structure for each morph target stored in the GeometryObject structure, but this is not a requirement. If a geometry node contains any morph weight data at all, then the weight for any unreferenced morph target shall be assumed to be zero. If a geometry node contains no morph weight data, then the weight for the morph target having index 0 shall be one, and the weights for all other morph targets shall be zero.

A MorphWeight structure can be the target of a track stored inside an Animation structure.

Morphing

When a mesh is deformed by the morphing operation, the morphed attribute $\mathbf{A}_{\text{morphed}}$ of a vertex (where an attribute can mean a position, normal, etc.) is given by

$$\mathbf{A}_{\text{morphed}} = \sum_i w_i \mathbf{M}_i,$$

where w_i is the weight for the morph target having index i , and the summation is taken over all indexes for which a morph target exists and a weight is specified. The symbol \mathbf{M}_i represents the input vertex attribute value for morph target i , and it depends on whether the morph target is absolute or relative as determined by the presence of the base property in the corresponding Morph structure. Let \mathbf{A}_i be the value of the vertex attribute specified for morph target i inside the appropriate VertexArray structure belonging to the morph target. If the morph target is absolute, then $\mathbf{M}_i = \mathbf{A}_i$. If the morph target is relative, then $\mathbf{M}_i = \mathbf{A}_i - \mathbf{A}_{b(i)}$, where the function $b(i)$ produces the base index for the morph target given by the base property of its Morph structure.

Properties

The properties listed in the following table may be specified for a MorphWeight structure.

Property	Type	Default	Description
index	uint32	0	The morph target index.

The index property specifies the morph target index to which the morph weight applies. If the GeometryObject structure contains no vertex data corresponding to this morph target index, then the MorphWeight structure should be ignored. Each MorphWeight structure belonging to any particular GeometryNode structure must have a unique morph target index among all morph weights belonging to that geometry node.

Structure Data

The following structures may compose the data stored inside a MorphWeight structure.

Substructure	Min	Max	Description
float	1	1	A MorphWeight structure must contain a single floating-point value.

Hierarchy

A MorphWeight structure may be contained inside the following structures.

Containing Structure	Description
GeometryNodeTrack	Multiple MorphWeight structures can be contained inside a GeometryNode structure.

Name

The Name structure holds the name of a node, morph target, material, or animation clip.

Structure Data

The following structures may compose the data stored inside a Name structure.

Substructure	Min	Max	Description
string	1	1	A Name structure must contain a name string.

Hierarchy

A Name structure may be contained inside the following structures.

Containing Structure	Description
Node BoneNode GeometryNode CameraNode LightNode	A single Name structure can be contained inside any node structure.
Morph	A single Name structure can be contained inside a Morph structure.
Material	A single Name structure can be contained inside a Material structure.
Clip	A single Name structure can be contained inside a Clip structure.

Node

The Node structure represents a single generic node in the scene with no associated object. The various types of node structures (Node, BoneNode, GeometryNode, CameraNode, and LightNode) are allowed to be top-level structures in an OpenGEX file, and each one can contain any number of other node structures of any type. This organization of nodes forms the main tree hierarchy of the scene. Every node can possess a name, a set of transforms, and a set of animations.

Name

If a Node structure contains a Name structure, then it defines the externally-visible name of the node that should be displayed to the user in applications such as a level editor. (This name should not be confused with an OpenDDL name that could be assigned to the Node structure in an OpenGEX file.)

Transforms

A Node structure may contain any number of Transform, Translation, Rotation, and Scale structures, and these collectively define the local transform for the node. Each transform can be designated as a node transform or an object transform (based on the value of its object property), and these divide the complete local transform into two factors. The node transform is inherited by subnodes, meaning that the local transform of a subnode is relative only to the node transform factor of its parent node. The object transform is applied only to the node to which it belongs and is not inherited by any subnodes.

The node transform is calculated by converting all of the transforms having an object property value of false to a 4×4 matrix and multiplying them together in the order that they appear as substructures. Similarly, the object transform is calculated by multiplying matrices together for the transforms having an object property value of true in the order that they appear as substructures. Any interleaving of transforms having different object property values has no meaning.

Animation

A node structure may contain one or more animation clips. Each clip can contain a set of animation tracks that each target one of the transforms contained by the node structure (or a MorphWeight structure in the case of a geometry node). Animation tracks often target a single component of a node's transform, such as the x coordinate of the node position or a rotation about a particular axis, and this is the reason that multiple transform structures are supported and expected to be used in the ordinary application of animation. See the Animation and Track structures for more information.

Structure Data

The following structures may compose the data stored inside a Node structure.

Substructure	Min	Max	Description
Name	0	1	A Node structure may have a name.
Transform Translation Rotation Scale	0	–	A Node structure may have any number of transformations applied to it.
Animation	0	–	A Node structure may contain animation tracks that are applied to the node transformations or morph weights.
Node BoneNode GeometryNode CameraNode LightNode	0	–	A Node structure may have any number of subnodes.

Hierarchy

A Node structure may be contained inside the following structures.

Containing Structure	Description
–	Node structures can be top-level structures.
Node BoneNode GeometryNode CameraNode LightNode	Node structures can be contained inside any other node structure.

ObjectRef

The ObjectRef structure holds a reference to an object structure. Object references are required by the GeometryNode, CameraNode, and LightNode structures, and they link these types of nodes to the objects they represent in the scene. A single object may be referenced by multiple nodes, and this allows an object to be instanced multiple times in the scene with different transforms and animations.

Structure Data

The following structures may compose the data stored inside a ObjectRef structure.

Substructure	Min	Max	Description
ref	1	1	An ObjectRef structure must contain a reference to a GeometryObject structure, LightObject structure, or CameraObject structure.

Hierarchy

A ObjectRef structure may be contained inside the following structures.

Containing Structure	Description
GeometryNode CameraNode LightNode	A single ObjectRef structure must be contained inside every GeometryNode, CameraNode, and LightNode structure.

Param

The Param structure holds a single parameter value.

Properties

The properties listed in the following table may be specified for a Param structure.

Property	Type	Default	Description
attrib	string	–	The parameter attribute.

The `attrib` property is required, and it specifies the meaning of the parameter. See the containing structures for information about the specific types of parameters that are defined.

Structure Data

The following structures may compose the data stored inside a Param structure.

Substructure	Min	Max	Description
float	1	1	A Param structure must contain one float substructure holding the value of the parameter.

Hierarchy

A Param structure may be contained inside the following structures.

Containing Structure	Description
Material	Param structures can be contained inside a Material structure to specify material attributes.
CameraObject LightObject	Param structures can be contained inside a CameraObject or LightObject structure to specify object parameters.
Atten	Param structures can be contained inside an Atten structure to specify attenuation parameters.
Clip	Param structures can be contained inside a Clip structure to specify animation clip attributes.

Rotation

The `Rotation` structure holds a rotation transformation in one of several possible variants.

When contained inside a node structure, a `Rotation` structure can be the target of a track stored inside an `Animation` structure.

Properties

The properties listed in the following table may be specified for a `Rotation` structure.

Property	Type	Default	Description
kind	string	"axis"	The kind of rotation.
object	bool	false	Whether the rotation is applied to the object only.

The `kind` property specifies the particular variant of the rotation transformation, and it must have one of the following values.

- A value of "x", "y", or "z" indicates that the rotation occurs about the x , y , or z axis. For these variants, the data contained inside the `Rotation` structure must be a single floating-point value representing the angle of rotation. For a particular angle θ , a rotation about the x , y , or z axis is converted to a 4×4 matrix by using the following formulas.

$$\mathbf{M}_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{M}_y = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{M}_z = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- A value of "axis" indicates that the rotation occurs about an arbitrary axis. For this variant, the data contained inside the `Rotation` structure must be a single array of four floating-point values. The first entry in the array is the angle of rotation, and the remaining three entries define the x , y , and z components of the axis of rotation. For a particular angle θ , a rotation about an axis \mathbf{A} is converted to a 4×4 matrix by using the following formula.

$$\mathbf{M}_{\text{axis}} = \begin{bmatrix} \cos \theta + A_x^2(1 - \cos \theta) & A_x A_y(1 - \cos \theta) - A_z \sin \theta & A_x A_z(1 - \cos \theta) + A_y \sin \theta & 0 \\ A_x A_y(1 - \cos \theta) + A_z \sin \theta & \cos \theta + A_y^2(1 - \cos \theta) & A_y A_z(1 - \cos \theta) - A_x \sin \theta & 0 \\ A_x A_z(1 - \cos \theta) - A_y \sin \theta & A_y A_z(1 - \cos \theta) + A_x \sin \theta & \cos \theta + A_z^2(1 - \cos \theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

It is not a requirement that the specified axis have unit length, so a reader should normalize the axis before calculating a rotation matrix.

- A value of "quaternion" indicates that the rotation is given by a quaternion. For this variant, the data contained inside the `Rotation` structure must be a single array of four floating-point values (x, y, z, w) that define the quaternion $xi + yj + zk + w$. For a particular unit quaternion (x, y, z, w) , the corresponding 4×4 rotation matrix is given by the following formula.

$$\mathbf{M}_{\text{quaternion}} = \begin{bmatrix} 1 - 2y^2 - 2z^2 & 2xy - 2wz & 2xz + 2wy & 0 \\ 2xy + 2wz & 1 - 2x^2 - 2z^2 & 2yz - 2wx & 0 \\ 2xz - 2wy & 2yz + 2wx & 1 - 2x^2 - 2y^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

It is not a requirement that the specified quaternion have length one, so a reader should unitize the quaternion before calculating a rotation matrix.

For the rotation variants that directly include an angle value, the angle is multiplied by the global angle metric to obtain an angle measured in radians. (See the `Metric` structure.) Note that a positive angle corresponds to a counterclockwise rotation when the axis points toward the viewer.

The object property specifies whether the rotation transformation applies to the node or to the object. See the `Node` structure for a discussion of node transforms and object transforms.

Structure Data

The following structures may compose the data stored inside a `Rotation` structure.

Substructure	Min	Max	Description
float float[4]	1	1	A <code>Rotation</code> structure must contain one float substructure holding the value of the rotation.

If the `kind` property is "x", "y", or "z", then the `Rotation` structure must contain a single floating-point value representing the angle of rotation. If the `kind` property is "xyz" or "quaternion", then the `Rotation` structure must contain a single array of four floating-point values.

Hierarchy

A Rotation structure may be contained inside the following structures.

Containing Structure	Description
Node BoneNode GeometryNode CameraNode LightNode	Rotation structures can be contained inside any node structure.
Texture	Rotation structures can be contained inside a Texture structure to specify texture coordinate transformations.

Scale

The `Scale` structure holds a scale transformation in one of several possible variants.

When contained inside a node structure, a `Scale` structure can be the target of a track stored inside an `Animation` structure.

Properties

The properties listed in the following table may be specified for a `Scale` structure.

Property	Type	Default	Description
<code>kind</code>	string	"xyz"	The kind of scale.
<code>object</code>	bool	false	Whether the scale is applied to the object only.

The `kind` property specifies the particular variant of the scale transformation, and it must have one of the following values.

- A value of "x", "y", or "z" indicates that the scale occurs along only the x , y , or z axis. For these variants, the data contained inside the `Scale` structure must be a single floating-point value representing the scale. For a particular scale s , a scale transformation along the x , y , or z axis is converted to a 4×4 matrix by using the following formulas.

$$\mathbf{M}_x = \begin{bmatrix} s & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{M}_y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{M}_z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- A value of "xyz" indicates that the scale occurs along all three coordinate axes. For this variant, the data contained inside the `Scale` structure must be a single array of three floating-point values representing the scale along each of the x , y , and z axes. For a particular scale \mathbf{S} , a scale transformation is converted to a 4×4 matrix by using the following formula.

$$\mathbf{M}_{xyz} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The `object` property specifies whether the scale transformation applies to the node or to the object. See the `Node` structure for a discussion of node transforms and object transforms.

Structure Data

The following structures may compose the data stored inside a Scale structure.

Substructure	Min	Max	Description
float float[3]	1	1	A Scale structure must contain one float substructure holding the value of the scale.

If the kind property is "x", "y", or "z", then the Scale structure must contain a single floating-point value representing the scale along one axis. If the kind property is "xyz", then the Scale structure must contain a single array of three floating-point values representing the scales along all three axes.

Hierarchy

A Scale structure may be contained inside the following structures.

Containing Structure	Description
Node BoneNode GeometryNode CameraNode LightNode	Scale structures can be contained inside any node structure.
Texture	Scale structures can be contained inside a Texture structure to specify texture coordinate transformations.

Skeleton

The Skeleton structure contains information about the bones belonging to a skeleton. The bone nodes belonging to a skeleton are identified by an array of OpenDDL references contained in a BoneRefArray substructure. The bind-pose transforms of those bone nodes are specified by an array of 4×4 matrices contained in a Transform substructure. See the Skin structure for details about how these are used in skinning calculations.

Structure Data

The following structures may compose the data stored inside a Skeleton structure.

Substructure	Min	Max	Description
BoneRefArray	1	1	A Skeleton structure must contain a bone reference array.
Transform	1	1	A Skeleton structure must contain an array of bind-pose transforms.

The number of matrices specified in the Transform structure must match the number of bones referenced in the BoneRefArray structure.

Hierarchy

A Skeleton structure may be contained inside the following structures.

Containing Structure	Description
Skin	A single Skeleton structure must be contained inside every Skin structure.

Skin

The Skin structure contains information about a skeleton and the per-vertex bone influence data for a skinned mesh. Each Mesh structure may contain a single Skin structure, which is required to contain all of the substructures shown in Listing 2.4.

The Skeleton structure contains an array of OpenDDL references to the full set of bone nodes that make up the skeleton and an array of their bind-pose transforms.

Listing 2.4. A Skin structure is required to contain the substructures shown here.

```
Skin
{
    Skeleton
    {
        BoneRefArray // References to the bone nodes.
        {
            ref {$bone1, $bone2, ...}
        }

        Transform // Bind-pose transforms for all bones.
        {
            float[16]
            {
                ...
            }
        }
    }

    BoneCountArray // Number of bones influencing each vertex.
    {
        uint8 {...}
    }

    BoneIndexArray // Bone index per influence per vertex.
    {
        uint8 {...}
    }

    BoneWeightArray // Weight per influence per vertex.
    {
        float {...}
    }
}
```

The `BoneCountArray` structure contains an array of counts that specify the number of bones influencing each vertex in the mesh. The size of this array must match the number of vertices contained in each `VertexArray` structure belonging to the mesh.

The `BoneIndexArray` and `BoneWeightArray` structures contain arrays of bone indexes and weighting values for each influence affecting each vertex. The size of both of these arrays must be equal to the sum of the bone counts contained in the `BoneCountArray` structure. Each index contained in the `BoneIndexArray` structure must be in the range $[0, N - 1]$, where N is the total number of bones referenced by the skeleton.

Skinning

When a mesh is deformed by the skinning operation, the skinned position $\mathbf{P}_{\text{skinned}}$ of a vertex is given by

$$\mathbf{P}_{\text{skinned}} = \sum_{i=0}^{n-1} w_i \mathbf{M}_{k(i)} \mathbf{B}_{k(i)}^{-1} \mathbf{T}_{\text{bind}} \mathbf{P}_{\text{bind}},$$

where \mathbf{P}_{bind} is the bind-pose position of the vertex (having an implicit w coordinate of one), \mathbf{T}_{bind} is the bind-pose transform of the skin given by the `Transform` substructure (if present), and the other symbols have the following meanings:

- n is the number of bones influencing the vertex, as given by the corresponding entry in the `BoneCountArray` structure.
- The function $k(i)$ produces the absolute bone index for the i -th influence, as given by the i -th entry corresponding to the vertex in the `BoneIndexArray` structure.
- $\mathbf{B}_{k(i)}$ is the bind-pose transform of the i -th influence, as given by entry $k(i)$ in the `Transform` substructure of the `Skeleton` structure.
- $\mathbf{M}_{k(i)}$ is the current transform of the i -th influence, equal to the transform of the bone node referenced by entry $k(i)$ in the `BoneRefArray` substructure of the `Skeleton` structure.
- w_i is the weight of the i -th influence, as given by the i -th entry corresponding to the vertex in the `BoneWeightArray` structure.

Normal vectors (treated as row vectors here) are calculated in a similar manner, but using the inverses of the matrices involved. The skinned vertex normal $\mathbf{N}_{\text{skinned}}$ is given by

$$\mathbf{N}_{\text{skinned}} = \sum_{i=0}^{n-1} w_i \mathbf{N}_{\text{bind}} \mathbf{T}_{\text{bind}}^{-1} \mathbf{B}_{k(i)} \mathbf{M}_{k(i)}^{-1},$$

where \mathbf{N}_{bind} is the bind-pose normal of the vertex (having an implicit w coordinate of zero). Of course, if the upper-left 3×3 portions of the matrices are known to be orthogonal, then normals can be calculated using the non-inverted matrices.

Structure Data

The following structures may compose the data stored inside a Skin structure.

Substructure	Min	Max	Description
Transform	0	1	A Skin structure may contain a bind-pose transform for the mesh.
Skeleton	1	1	A Skin structure must contain a skeleton.
BoneCountArray	1	1	A Skin structure must contain a bone count array.
BoneIndexArray	1	1	A Skin structure must contain a bone index array.
BoneWeightArray	1	1	A Skin structure must contain a bone weight array.

Hierarchy

A Skin structure may be contained inside the following structures.

Containing Structure	Description
Mesh	A single Skin structure can be contained inside a Mesh structure.

Spectrum

The `Spectrum` structure holds a table of data representing the spectral distribution for emission or reflectance.

A spectrum is specified by a table of n values $S(\lambda)$ that are evenly spaced between minimum and maximum wavelengths λ_{\min} and λ_{\max} , measured in nanometers (nm). This data can be converted to an XYZ color by calculating the three sums

$$\begin{aligned} X &= \sum_{\lambda} S(\lambda) \bar{x}(\lambda) \Delta\lambda \\ Y &= \sum_{\lambda} S(\lambda) \bar{y}(\lambda) \Delta\lambda \\ Z &= \sum_{\lambda} S(\lambda) \bar{z}(\lambda) \Delta\lambda, \end{aligned}$$

where $\Delta\lambda = (\lambda_{\max} - \lambda_{\min}) / (n - 1)$ is the difference between consecutive wavelengths. The functions $\bar{x}(\lambda)$, $\bar{y}(\lambda)$, and $\bar{z}(\lambda)$ are the standard CIE XYZ color matching functions defined by the values tabulated in Tables 2.4, 2.5, and 2.6. The color matching functions are zero for all wavelengths less than 360 nm or greater than 830 nm. For wavelengths that are not a multiple of 5 nm, a reader may use any preferred method of interpolation to determine the values of $\bar{x}(\lambda)$, $\bar{y}(\lambda)$, and $\bar{z}(\lambda)$.

An XYZ color can be transformed into the particular RGB color space used by an OpenGEX file through the method described under the `Metric` structure.

The units of the data values $S(\lambda)$ are inverse nanometers (nm^{-1}), and this is not affected by the global distance metric.

A spectrum should ordinarily be normalized so that its luminance Y is unity. The luminous intensity of a light source is specified separately by a `Param` structure inside the `LightObject` structure. The luminance of an emissive material is specified separately by a `Param` structure inside the `Material` structure.

Properties

The properties listed in the following table may be specified for a `Spectrum` structure.

Property	Type	Default	Description
<code>attrib</code>	<code>string</code>	–	The spectrum attribute.
<code>min</code>	<code>uint32</code>	–	The minimum wavelength, in nanometers.
<code>max</code>	<code>uint32</code>	–	The maximum wavelength, in nanometers

The `attrib` property is required, and it specifies the meaning of the spectrum. See the containing structures for information about the specific types of attributes that are defined.

The `min` and `max` properties are both required, and they specify the range of wavelengths over which the spectrum is defined. The maximum wavelength must be greater than the minimum wavelength.

Structure Data

The following structures may compose the data stored inside a `Spectrum` structure.

Substructure	Min	Max	Description
float	1	1	A <code>Spectrum</code> structure must contain one <code>float</code> substructure holding the spectral distribution data.

Hierarchy

A `Spectrum` structure may be contained inside the following structures.

Containing Structure	Description
<code>Material</code>	<code>Spectrum</code> structures can be contained inside a <code>Material</code> structure to specify material attributes.
<code>LightObject</code>	A single <code>Spectrum</code> structure can be contained inside a <code>LightObject</code> structure to specify the emission spectrum of the light.

λ	$\bar{x}(\lambda)$	λ	$\bar{x}(\lambda)$	λ	$\bar{x}(\lambda)$	λ	$\bar{x}(\lambda)$	λ	$\bar{x}(\lambda)$
360	0.0001299	455	0.3187	550	0.43345	645	0.3608	740	0.0006900786
365	0.0002321	460	0.2908	555	0.51205	650	0.2835	745	0.0004760213
370	0.0004149	465	0.2511	560	0.5945	655	0.2187	750	0.0003323011
375	0.0007416	470	0.19536	565	0.6784	660	0.1649	755	0.0002348261
380	0.001368	475	0.1421	570	0.7621	665	0.1212	760	0.0001661505
385	0.002236	480	0.09564	575	0.8425	670	0.0874	765	0.000117413
390	0.004243	485	0.05795	580	0.9163	675	0.0636	770	0.00008307527
395	0.00765	490	0.03201	585	0.9786	680	0.04677	775	0.00005870652
400	0.01431	495	0.0147	590	1.0263	685	0.0329	780	0.00004150994
405	0.02319	500	0.0049	595	1.0567	690	0.0227	785	0.00002935326
410	0.04351	505	0.0024	600	1.0622	695	0.01584	790	0.00002067383
415	0.07763	510	0.0093	605	1.0456	700	0.01135916	795	0.00001455977
420	0.13438	515	0.0291	610	1.0026	705	0.008110916	800	0.00001025398
425	0.21477	520	0.06327	615	0.9384	710	0.005790346	805	0.000007221456
430	0.2839	525	0.1096	620	0.85445	715	0.004109457	810	0.000005085868
435	0.3285	530	0.1655	625	0.7514	720	0.002899327	815	0.000003581652
440	0.34828	535	0.22575	630	0.6424	725	0.00204919	820	0.000002522525
445	0.34806	540	0.2904	635	0.5419	730	0.001439971	825	0.000001776509
450	0.3362	545	0.3597	640	0.4479	735	0.0009999493	830	0.000001251141

Table 2.4. This table contains the values of the color matching function $\bar{x}(\lambda)$ for wavelengths λ measured in nanometers (nm).

λ	$\bar{y}(\lambda)$	λ	$\bar{y}(\lambda)$	λ	$\bar{y}(\lambda)$	λ	$\bar{y}(\lambda)$	λ	$\bar{y}(\lambda)$
360	0.000003917	455	0.048	550	0.99495	645	0.1382	740	0.0002492
365	0.000006965	460	0.06	555	1.0	650	0.107	745	0.0001719
370	0.00001239	465	0.0739	560	0.995	655	0.0816	750	0.00012
375	0.00002202	470	0.09098	565	0.9786	660	0.061	755	0.0000848
380	0.000039	475	0.1126	570	0.952	665	0.04458	760	0.00006
385	0.000064	480	0.13902	575	0.9154	670	0.032	765	0.0000424
390	0.000120	485	0.1693	580	0.87	675	0.0232	770	0.00003
395	0.000217	490	0.20802	585	0.8163	680	0.017	775	0.0000212
400	0.000396	495	0.2586	590	0.757	685	0.01192	780	0.000015
405	0.00064	500	0.323	595	0.6949	690	0.00821	785	0.0000106
410	0.00121	505	0.4073	600	0.631	695	0.005723	790	0.0000074657
415	0.00218	510	0.503	605	0.5668	700	0.004102	795	0.0000052578
420	0.004	515	0.6082	610	0.503	705	0.002929	800	0.0000037029
425	0.0073	520	0.71	615	0.4412	710	0.002091	805	0.0000026078
430	0.0116	525	0.7932	620	0.381	715	0.001484	810	0.0000018366
435	0.01684	530	0.862	625	0.321	720	0.001047	815	0.0000012934
440	0.023	535	0.91485	630	0.265	725	0.00074	820	0.00000091093
445	0.0298	540	0.954	635	0.217	730	0.00052	825	0.00000064153
450	0.038	545	0.9803	640	0.175	735	0.0003611	830	0.00000045181

Table 2.5. This table contains the values of the color matching function $\bar{y}(\lambda)$ for wavelengths λ measured in nanometers (nm).

λ	$\bar{z}(\lambda)$	λ	$\bar{z}(\lambda)$	λ	$\bar{z}(\lambda)$	λ	$\bar{z}(\lambda)$	λ	$\bar{z}(\lambda)$
360	0.0006061	455	1.7441	550	0.00875	645	0.00001	740	0.0
365	0.001086	460	1.6692	555	0.00575	650	0.0	745	0.0
370	0.001946	465	1.5281	560	0.0039	655	0.0	750	0.0
375	0.003486	470	1.28764	565	0.00275	660	0.0	755	0.0
380	0.00645	475	1.0419	570	0.0021	665	0.0	760	0.0
385	0.01055	480	0.81295	575	0.0018	670	0.0	765	0.0
390	0.02005	485	0.6162	580	0.00165	675	0.0	770	0.0
395	0.03621	490	0.46518	585	0.0014	680	0.0	775	0.0
400	0.06785	495	0.3533	590	0.0011	685	0.0	780	0.0
405	0.1102	500	0.272	595	0.001	690	0.0	785	0.0
410	0.2074	505	0.2123	600	0.0008	695	0.0	790	0.0
415	0.3713	510	0.1582	605	0.0006	700	0.0	795	0.0
420	0.6456	515	0.1117	610	0.00034	705	0.0	800	0.0
425	1.03905	520	0.07825	615	0.00024	710	0.0	805	0.0
430	1.3856	525	0.05725	620	0.00019	715	0.0	810	0.0
435	1.62296	530	0.04216	625	0.0001	720	0.0	815	0.0
440	1.74706	535	0.02984	630	0.00005	725	0.0	820	0.0
445	1.7826	540	0.0203	635	0.00003	730	0.0	825	0.0
450	1.77211	545	0.0134	640	0.00002	735	0.0	830	0.0

Table 2.6. This table contains the values of the color matching function $\bar{z}(\lambda)$ for wavelengths λ measured in nanometers (nm).

Texture

The Texture structure holds information about a single texture map and how it is accessed with texture coordinates.

Texture Coordinate Transform

A Texture structure may contain any number of Transform, Translation, Rotation, and Scale structures, and these collectively define the transformation applied to texture coordinates before they are used to access the texture map. The texture coordinate transform is calculated by converting each of the transforms to a 4×4 matrix and multiplying them together in the order that they appear as substructures. (The object property of each transform structure is ignored in this case.)

The texture coordinate transformations may be animated through the presence of Animation substructures whose tracks target the specific transform structures.

Properties

The properties listed in the following table may be specified for a Texture structure.

Property	Type	Default	Description
attrib	string	–	The texture attribute.
texcoord	uint32	0	The index of the texture coordinate set associated with the texture.
swizzle	string	"i"	The component swizzle.
x_address	string	"repeat"	The addressing mode for the x coordinate.
y_address	string	"repeat"	The addressing mode for the y coordinate.
z_address	string	"repeat"	The addressing mode for the z coordinate.
border	string	"zero"	The border color.

The `attrib` property is required, and it specifies the meaning of the texture. See the containing structures for information about the specific types of attributes that are defined.

The `texcoord` property specifies which texture coordinate set belonging to a mesh should be used to access the texture.

The `swizzle` property specifies the order into which the components of each texel are permuted. The value assigned to this property must be a string of length one or four, and each character in the string must be one from the set `{'r', 'g', 'b', 'a', 'x', 'y', 'z', 'w', '0', '1', 'i'}`. The characters `'r'` and `'x'` correspond to the first component, the characters `'g'` and `'y'` correspond to the second

component, the characters 'b' and 'z' correspond to the third component, and the characters 'a' and 'w' correspond to the fourth component. The characters '0' and '1' mean the constant values zero and one. The character 'i' indicates the identity component for its position in the string. The identity string for a four-channel texture is "rgba", the identity string for a three-channel texture is "rgb1", the identity string for a two-channel texture is "rg01", and the identity string for a one-channel texture is "r001". The meaning of a string containing a single character is equivalent to a string containing that same character repeated four times.

The `x_address`, `y_address`, and `z_address` properties specify the addressing modes that are applied when any of the `x`, `y`, and `z` coordinates used to sample the texture fall outside the range $[0, 1]$. The values that can be assigned to these properties are described in Table 2.7.

Addressing Mode	Description
"repeat"	The texture is accessed with $\text{frac}(c)$, causing the image to repeat at each integer.
"clamp"	The texture is accessed with $\text{sat}(c)$, causing the image to be clamped at its boundary.
"border"	If c is in the range $[0, 1]$, then the texture is accessed with the coordinate c . Otherwise, the border color is produced.
"mirror_repeat"	The texture is accessed with $1 - 2\left \text{frac}\left(\frac{c}{2}\right) - \frac{1}{2}\right $, causing the image to repeat with an alternating mirroring pattern.
"mirror_clamp"	The texture is accessed with $\text{sat}(c)$, causing the image to be mirrored about zero and clamped at its boundary.
"mirror_border"	If $ c $ is in the range $[0, 1]$, then the texture is accessed with the coordinate $ c $. Otherwise, the border color is produced.

Table 2.7. This table describes the addressing modes that can be assigned to the `x_address`, `y_address`, and `z_address` properties. The coordinate c represents one of the `x`, `y`, and `z` coordinates used to sample the texture. The function $\text{frac}(c)$ is the fraction defined by $\text{frac}(c) = c - |c|$, and the function $\text{sat}(c)$ is the saturation defined by $\text{sat}(c) = \min(\max(c, 0), 1)$.

The border property specifies one of three constant colors to be produced when a texture is accessed with a coordinate falling outside the range $[0, 1]$ and the addressing mode corresponding to that coordinate is "border" or "mirror_border". The value of this property may be "black" for a border color of $(0, 0, 0, 1)$, "white" for a border color of $(1, 1, 1, 1)$, or "zero" for a border color of $(0, 0, 0, 0)$.

Structure Data

The following structures may compose the data stored inside a Texture structure.

Substructure	Min	Max	Description
string	1	1	A Texture structure must contain one <code>string</code> substructure holding the file name of the texture.
Transform Translation Rotation Scale	0	–	A Texture structure may contain any number of transformations that are applied to the texture coordinates of a mesh when they are used to fetch from the texture map.
Animation	0	–	A Texture structure may contain animation tracks that are applied to the texture coordinate transformations.

Hierarchy

A Texture structure may be contained inside the following structures.

Containing Structure	Description
Material	Texture structures can be contained inside a <code>Material</code> structure to specify material attributes.
LightObject	A single Texture structure can be contained inside a <code>LightObject</code> structure to specify a projected texture for a spot light.

Time

The `Time` structure contains key time data in an animation track.

Properties

The properties listed in the following table may be specified for a `Time` structure.

Property	Type	Default	Description
curve	string	"linear"	The function defining the interpolation curve.

The curve property specifies the manner in which time values are interpolated and must have one of the following values.

- A value of "linear" indicates that times are interpolated linearly.
- A value of "bezier" indicates that times are interpolated on a one-dimensional cubic Bézier curve.

See the `Track` structure for information about calculating interpolated key values.

Structure Data

The following structures may compose the data stored inside a `Time` structure.

Substructure	Min	Max	Description
Key	1	3	A <code>Time</code> structure must contain one or three key substructures, depending on the curve property, holding the time curve data.

The `Time` structure must contain a `Key` structure whose `kind` property is "value". The data inside this `Key` structure must consist of a monotonically increasing sequence of time values.

If the curve property is "bezier", then the `Time` structure must contain two additional `Key` structures whose `kind` properties are "-control" and "+control". These hold the incoming and outgoing control points for the time curve, respectively. Each incoming control point must be less than its corresponding time value but greater than the outgoing control point for the preceding time value, if any. Likewise, each outgoing control point must be greater than its corresponding time value but less than the incoming control point for the succeeding time value, if any.

Hierarchy

A Time structure may be contained inside the following structures.

Containing Structure	Description
Track	A single Time structure must be contained inside every Track structure.

Track

The Track structure contains animation key data for a single transformation structure (Transform, Translation, Rotation, and Scale) or a single MorphWeight structure. The key data is separated into time and value curves having an equal number of data points.

Interpolation

Given a time t that falls between two key times t_1 and t_2 stored in the Time structure, a parameter s in the range $[0, 1)$ can be calculated and used to interpolate between the corresponding values v_1 and v_2 stored in the Value structure.

If the curve property for the Time structure is "linear", then the parameter s is given by

$$s(t) = \frac{t - t_1}{t_2 - t_1}.$$

If the curve property for the Time structure is "bezier", then the parameter s must be that for which the one-dimensional cubic Bézier polynomial produces the time t . This can be determined by solving the equation

$$(1 - s)^3 t_1 + 3s(1 - s)^2 c_1 + 3s^2(1 - s)c_2 + s^3 t_2 - t = 0,$$

where c_1 is the outgoing control point (in the Key structure with a kind property of "+control") corresponding to the time t_1 , and c_2 is the incoming control point (in the Key structure with a kind property of "-control") corresponding to the time t_2 . Because the times and control points are required to satisfy $t_1 < c_1 < c_2 < t_2$, there is only one real solution, and it can quickly be found through an iterative application of Newton's method, beginning with $s_0 = s(t)$ as in the linear case. Refined values of s are given by the formula

$$s_{i+1} = s_i - \frac{(t_2 - 3c_2 + 3c_1 - t_1)s_i^3 + 3(c_2 - 2c_1 + t_1)s_i^2 + 3(c_1 - t_1)s_i + t_i - t}{3(t_2 - 3c_2 + 3c_1 - t_1)s_i^2 + 6(c_2 - 2c_1 + t_1)s_i + 3(c_1 - t_1)}.$$

Once the parameter s has been calculated, it is exclusively used to interpolate key values, and the time t is no longer needed.

If the curve property for the Value structure is "constant", then the interpolated value v is trivially given by $v(s) = v_1$.

If the curve property for the Value structure is "linear", then the interpolated value v is given by

$$v(s) = (1 - s)v_1 + sv_2.$$

If the curve property for the Value structure is "bezier", then v is given by

$$v(s) = (1 - s)^3 v_1 + 3s(1 - s)^2 p_1 + 3s^2(1 - s)p_2 + s^3 v_2,$$

where p_1 is the outgoing control point (in the Key structure with a kind property of "+control") corresponding to the value v_1 , and p_2 is the incoming control point (in the Key structure with a kind property of "-control") corresponding to the value v_2 .

If the curve property for the Value structure is "tcb", then v is given by the Hermite curve

$$v(s) = (1 - 3s^2 + 2s^3)v_1 + s^2(3 - 2s)v_2 + s(s - 1)^2 u_1 + s^2(s - 1)u_2,$$

where u_1 and u_2 are tangents derived from tension, continuity, and bias parameters τ_i , χ_i , and β_i corresponding to each key value v_i . These tangents are given by the formulas

$$u_1 = \frac{(1 - \tau_1)(1 + \chi_1)(1 + \beta_1)}{2}(v_1 - v_0) + \frac{(1 - \tau_1)(1 - \chi_1)(1 - \beta_1)}{2}(v_2 - v_1)$$

and

$$u_2 = \frac{(1 - \tau_2)(1 - \chi_2)(1 + \beta_2)}{2}(v_2 - v_1) + \frac{(1 - \tau_2)(1 + \chi_2)(1 - \beta_2)}{2}(v_3 - v_2).$$

The value v_0 is the one preceding v_1 in the Key structure. If v_1 is the first key value, then $v_0 = v_1$, and the first term of the formula for u_1 is eliminated. Likewise, the value v_3 is the one following v_2 in the Key structure. If v_2 is the last key value, then $v_3 = v_2$, and the second term of the formula for u_2 is eliminated.

The tension, continuity, and bias parameters are always scalars and are stored alongside the key values (inside the Value structure) in additional Key structures having kind properties of "tension", "continuity", and "bias".

The key values v_i in all cases, and the control points p_i in the case of Bézier curves, can be scalars or vectors. For vectors, interpolated values are calculated componentwise with the above formulas.

Properties

The properties listed in the following table may be specified for a Track structure.

Property	Type	Default	Description
target	ref	–	The target structure of the animation track.

The target property is required and specifies the particular structure that is animated by the track. The target structure must be a Transform, Translation, Rotation, Scale, or MorphWeight structure.

Structure Data

The following structures may compose the data stored inside a `Track` structure.

Substructure	Min	Max	Description
Time	1	1	A <code>Track</code> structure must contain a set of time keys.
Value	1	1	A <code>Track</code> structure must contain a set of value keys.

A `Track` structure must always contain exactly one `Time` structure and one `Value` structure. These each contain one or more `Key` structures that contain the actual animation data. The number of time keys and the number of values keys must be equal.

Hierarchy

A `Track` structure may be contained inside the following structures.

Containing Structure	Description
Animation	Track structures can be contained inside an <code>Animation</code> structure.

Transform

The Transform structure holds one or more 4×4 transformation matrices. In the cases that a Transform structure is contained inside any type of node structure, a Texture structure, or a Skin structure, it must contain a single matrix. In the case that a Transform structure is contained inside a Skeleton structure, it must contain an array of matrices with one entry for each bone referenced by the skeleton.

When contained inside a node structure or a Texture structure, a Transform structure can be the target of a track stored inside an Animation structure.

Properties

The properties listed in the following table may be specified for a Transform structure.

Property	Type	Default	Description
object	bool	false	Whether the transform is applied to the object only.

The object property specifies whether the transformation matrix applies to the node or to the object. See the Node structure for a discussion of node transforms and object transforms. The object property is allowed but must be ignored for Transform structures not contained inside a node structure.

Structure Data

The following structures may compose the data stored inside a Transform structure.

Substructure	Min	Max	Description
float[16] float[12] float[9] float[6] float[4]	1	1	A Transform structure must contain one float substructure holding the values of one or more transformation matrices in subarrays. The subarray size must be one of 16, 12, 9, 6, or 4.

The number of entries stored for each matrix inside a Transform structure can have one of five possible values, which must be specified as the subarray size of the float substructure. Each size corresponds to a specific submatrix of a full 4×4 transformation. In all cases, the matrix entries are specified in column-major order such that all entries in one column occur before the first entry in the next column.

If the number of entries is 16, then the entire 4×4 transformation is specified by the values $\{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p\}$, forming the matrix

$$\begin{bmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{bmatrix}.$$

If the number of entries is 12, then the first three rows of the 4×4 transformation are specified by the values $\{a, b, c, d, e, f, g, h, i, j, k, l\}$, forming the matrix

$$\begin{bmatrix} a & d & g & j \\ b & e & h & k \\ c & f & i & l \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

If the number of entries is 9, then the upper-left 3×3 portion of the 4×4 transformation is specified by the values $\{a, b, c, d, e, f, g, h, i\}$, forming the matrix

$$\begin{bmatrix} a & d & g & 0 \\ b & e & h & 0 \\ c & f & i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

If the number of entries is 6, then the first, second, and fourth entries in the first two rows of the 4×4 transformation are specified by the values $\{a, b, c, d, e, f\}$, forming the matrix

$$\begin{bmatrix} a & c & 0 & e \\ b & d & 0 & f \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

If the number of entries is 4, then the upper-left 2×2 portion of the 4×4 transformation is specified by the values $\{a, b, c, d\}$, forming the matrix

$$\begin{bmatrix} a & c & 0 & 0 \\ b & d & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Hierarchy

A Transform structure may be contained inside the following structures.

Containing Structure	Description
Node BoneNode GeometryNode CameraNode LightNode	Transform structures can be contained inside any node structure.
Skin	A single Transform structure can be contained inside a Skin structure to specify a bind-pose transform for the mesh.
Skeleton	A single Transform structure must be contained inside a Skeleton structure to specify an array of bind-pose transforms for the bones.
Texture	Transform structures can be contained inside a Texture structure to specify texture coordinate transformations.

Translation

The Translation structure holds a translation transformation in one of several possible variants.

When contained inside a node structure, a Translation structure can be the target of a track stored inside an Animation structure.

Properties

The properties listed in the following table may be specified for a Translation structure.

Property	Type	Default	Description
kind	string	"xyz"	The kind of translation.
object	bool	false	Whether the translation is applied to the object only.

The kind property specifies the particular variant of the translation transformation, and it must have one of the following values.

- A value of "x", "y", or "z" indicates that the translation occurs along only the x , y , or z axis. For these variants, the data contained inside the Translation structure must be a single floating-point value representing the displacement. For a particular displacement d , a translation along the x , y , or z axis is converted to a 4×4 matrix by using the following formulas.

$$\mathbf{M}_x = \begin{bmatrix} 1 & 0 & 0 & d \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{M}_y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & d \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{M}_z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- A value of "xyz" indicates that the translation occurs along all three coordinate axes. For this variant, the data contained inside the Translation structure must be a single array of three floating-point values representing the displacement along each of the x , y , and z axes. For a particular displacement \mathbf{D} , a translation is converted to a 4×4 matrix by using the following formula.

$$\mathbf{M}_{xyz} = \begin{bmatrix} 1 & 0 & 0 & D_x \\ 0 & 1 & 0 & D_y \\ 0 & 0 & 1 & D_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The object property specifies whether the translation transformation applies to the node or to the object. See the Node structure for a discussion of node transforms and object transforms.

Structure Data

The following structures may compose the data stored inside a Translation structure.

Substructure	Min	Max	Description
float float[3]	1	1	A Translation structure must contain one float substructure holding the value of the translation.

If the kind property is "x", "y", or "z", then the Translation structure must contain a single floating-point value representing the displacement along one axis. If the kind property is "xyz", then the Translation structure must contain a single array of three floating-point values representing the displacement along all three axes.

Hierarchy

A Translation structure may be contained inside the following structures.

Containing Structure	Description
Node BoneNode GeometryNode CameraNode LightNode	Translation structures can be contained inside any node structure.
Texture	Translation structures can be contained inside a Texture structure to specify texture coordinate transformations.

Value

The Value structure contains key value data in an animation track.

Properties

The properties listed in the following table may be specified for a Value structure.

Property	Type	Default	Description
curve	string	"linear"	The function defining the interpolation curve.

The curve property specifies the manner in which values are interpolated and must have one of the following values.

- A value of "constant" indicates that values are not interpolated but remain constant until the next key time.
- A value of "linear" indicates that values are interpolated linearly.
- A value of "bezier" indicates that values are interpolated on a cubic Bézier curve.
- A value of "tcb" indicates that values are interpolated on a tension-continuity-bias (TCB) spline.

See the Track structure for information about calculating interpolated key values.

Structure Data

The following structures may compose the data stored inside a Value structure.

Substructure	Min	Max	Description
Key	1	4	A Value structure must contain one, three, or four key substructures, depending on the curve property, holding the value curve data.

The Value structure must contain a Key structure whose kind property is "value". The data inside this Key structure must have the same dimensionality as the target of the enclosing Track structure.

If the curve property is "bezier", then the Value structure must contain two additional Key structures whose kind properties are "-control" and "+control". These hold the incoming and outgoing control points for the value curve, respectively, and they must have the same dimensionality as the key values.

If the curve property is "tcb", then the Value structure must contain three additional Key structures whose kind properties are "tension", "bias", and "continuity". The data contained inside these three Key structures is always scalar.

Hierarchy

A Value structure may be contained inside the following structures.

Containing Structure	Description
Track	A single Value structure must be contained inside every Track structure.

VertexArray

The `VertexArray` structure contains array data for a single vertex attribute in a mesh. See the `Mesh` structure for information about how arrays are used in a mesh.

Properties

The properties listed in the following table may be specified for a `VertexArray` structure.

Property	Type	Default	Description
<code>attrib</code>	<code>string</code>	–	The vertex attribute.
<code>index</code>	<code>uint32</code>	0	The attribute index.
<code>morph</code>	<code>uint32</code>	0	The morph target index.

The `attrib` property specifies the meaning of the data contained in the vertex array. Its value must be a string containing a legal OpenDDL identifier. The vertex array `attrib` values defined by this specification are shown in Table 2.8. A writer may include `VertexArray` structures with application-defined `attrib` values. If a reader encounters any of these for which the `attrib` value is either unsupported or unrecognized, then the vertex array should be ignored.

Attribute	Meaning
"position"	The vertex position.
"normal"	The normal vector.
"tangent"	The tangent vector aligned to the <i>x</i> texture coordinate.
"bitangent"	The tangent vector aligned to the <i>y</i> texture coordinate.
"color"	The vertex color.
"texcoord"	The texture coordinates.

Table 2.8. This table lists the vertex attribute types defined by OpenGEX.

The `index` property allows an attribute index to be specified for any vertex array. This provides a mechanism through which multiple arrays with the same `attrib` property can be included in a mesh. For example, a mesh having two sets of texture coordinates would specify them using two `VertexArray` structures with the `attrib` property set to "texcoord", but they would have their `index` properties set to 0 and 1. If the attribute index is omitted, then it is equivalent to specifying an index of 0.

Normal arrays, tangent arrays, and bitangent arrays should generally contain three-dimensional data. If a tangent array contains four-dimensional data, then the fourth component should contain the sign of the volume $\mathbf{T} \wedge \mathbf{B} \wedge \mathbf{N}$, where \mathbf{T} , \mathbf{B} , and \mathbf{N} are the corresponding three-dimensional entries for the same vertex taken from the tangent, bitangent, and normal arrays.

The `texcoord` property of the `Texture` structure specifies which texture coordinate array to use when accessing a texture map. If multiple tangent and bitangent arrays are present, they should have `index` property values corresponding to the attribute index of the associated texture coordinate arrays so that a matching tangent frame can be determined for each set of texture coordinates.

The `morph` property specifies the index of the morph target to which the vertex array belongs. A mesh may contain multiple vertex arrays having the same values for the `attrib` and `index` properties only if they have different values for their `morph` properties. A complete morph target having the index k is composed of the following two sets of vertex arrays:

- All of the vertex arrays for which the `morph` property has a value of k .
- For any values of the `attrib` and `index` properties that are specified for a vertex array of *any* morph target but it not specified for a vertex array of the morph target having index k , the vertex array having the same values for the `attrib` and `index` properties and the minimum value for the `morph` property (which may be the default value of 0).

The blending weights applied to the morph targets are specified inside a `Morph` structure contained in a `GeometryNode` structure that references a `GeometryObject` structure containing the mesh. For any geometry node not containing any `MorphWeight` structures but referencing a geometry object containing multiple morph targets, all vertex arrays for which the `morph` property is not 0 should be ignored.

Structure Data

The following structures may compose the data stored inside a `VertexArray` structure.

Substructure	Min	Max	Description
half half[2] half[3] half[4] float float[2] float[3] float[4] double double[2] double[3] double[4]	1	1	A <code>VertexArray</code> structure must contain one substructure holding the vertex array data, and it must have a floating-point primitive data type.

The data for each type of vertex array attribute may consist of an array of floating-point values or an array of subarrays containing two, three, or four values each. When fewer than four components are specified in the data for any vertex array, the second, third, and fourth components shall implicitly be given the values 0, 0, and 1, respectively.

Hierarchy

A VertexArray structure may be contained inside the following structures.

Containing Structure	Description
Mesh	VertexArray structures can be contained inside a Mesh structure.

A

OpenDDL Reference

A.1 Introduction

The Open Data Description Language (OpenDDL) is a generic text-based language that is designed to store arbitrary data in a concise human-readable format. It can be used as a means for easily exchanging information among many programs or simply as a method for storing a program's data in an editable format. Each unit of data in an OpenDDL file has an explicitly specified type, and this eliminates ambiguity and fragile inferencing methods that can impact the integrity of the data. This strong typing is further supported by the specification of an exact number of bits required to store numerical data values when converted to a binary representation.

The data structures in an OpenDDL file are organized as a collection of trees. The language includes a built-in mechanism for making references from one data structure to any other data structure, effectively allowing the contents of a file to take the form of a directed graph.

As a foundation for higher-level data formats, OpenDDL is intended to be minimalistic. It assigns no meaning whatsoever to any data beyond its hierarchical organization, and it imposes no restrictions on the composition of data structures. Semantics and validation are left to be defined by specific higher-level formats derived from OpenDDL. The core language is designed to place as little burden as possible on readers so that it's easy to write programs that understand OpenDDL.

The OpenDDL syntax is illustrated in the "railroad diagrams" found throughout this specification, and it is designed to feel familiar to C/C++ programmers. Whitespace never has any meaning, so OpenDDL files can be formatted in any manner preferred.

A.2 Structures

An OpenDDL file is composed of a sequence of *structures*. A single structure consists of a type identifier followed by an optional name, an optional list of properties, and its data payload enclosed in braces, as shown in Figure A.1. There are two general classes of structures called *primitive* structures and *derived* structures. Primitive structures have types that are defined by OpenDDL itself, and they contain primitive data such as integers, floating-point numbers, or strings. Derived structures represent custom data types defined by a derivative file format, and they can contain other structures, primitive or derived, that can be organized in a hierarchical manner.

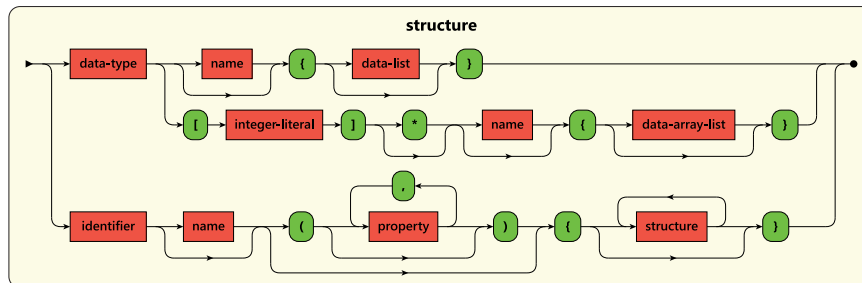


Figure A.1. An OpenDDL file contains a sequence of structures that follow the production rule shown here.

[*Example* — Suppose that a derivative file format defined a data type called `Vertex` that contains the 3D coordinates of a single vertex position. This could be written as follows.

```
Vertex
{
  float {1.0, 2.0, 3.0}
}
```

The `Vertex` identifier represents a derived structure defined by the file format, and it contains another structure of type `float`, which is a primitive data type defined by OpenDDL. The data in the `float` structure consists of the three values 1.0, 2.0, and 3.0. — *End example*]

If a structure has a type that is not recognized by an implementation, then that structure and all of the data it contains must be ignored without producing an error. This allows extensions to be added to a data format without breaking compatibility with implementations that do not support them.

A.3 Data Types

OpenDDL defines the 16 *primitive data types* listed in Table A.1, and they can be specified by the long identifiers and short identifiers shown in Figure A.2. There is no difference in the meaning between the long and short identifiers, so they can be used interchangeably. The three floating-point data types each have additional long and short identifiers, making it possible to specify those types with four different identifiers that all have equivalent meanings.

When used as the identifier for a structure, each entry in the Table A.1 indicates that the structure is a primitive structure and its data payload is composed of an array of literal values. Primitive structures cannot have substructures.

Long Identifier	Short Identifier	Description
bool	b	Boolean type that can have the value true or false.
int8	i8	8-bit signed integer that can have values in the range $[-2^7, 2^7 - 1]$.
int16	i16	16-bit signed integer that can have values in the range $[-2^{15}, 2^{15} - 1]$.
int32	i32	32-bit signed integer that can have values in the range $[-2^{31}, 2^{31} - 1]$.
int64	i64	64-bit signed integer that can have values in the range $[-2^{63}, 2^{63} - 1]$.
uint8	u8	An 8-bit unsigned integer that can have values in the range $[0, 2^8 - 1]$.
uint16	u16	16-bit unsigned integer that can have values in the range $[0, 2^{16} - 1]$.
uint32	u32	32-bit unsigned integer that can have values in the range $[0, 2^{32} - 1]$.
uint64	u64	64-bit unsigned integer that can have values in the range $[0, 2^{64} - 1]$.
half, float16	h, f16	16-bit floating-point type in the standard S1-E5-M10 format.
float, float32	f, f32	32-bit floating-point type in the standard S1-E8-M23 format.
double, float64	d, f64	64-bit floating-point type in the standard S1-E11-M52 format.
string	s	Double-quoted character string with contents encoded in UTF-8.
ref	r	Sequence of structure names, or the keyword null.
type	t	Type whose values are identifiers naming the types in this table.
base64	z	Generic binary data encoded as base64.

Table A.1. These are the 16 primitive data types defined by OpenDDL.

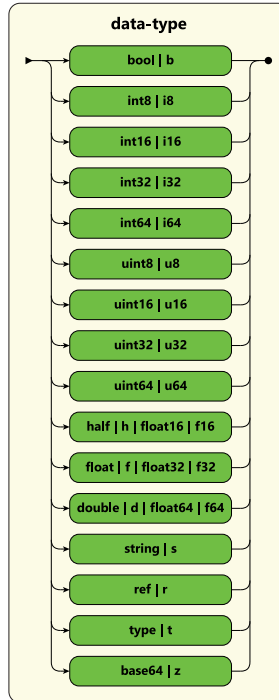


Figure A.2. These are the 16 primitive data types defined by OpenDDL.

There is no implicit type conversion in OpenDDL. Data belonging to a primitive structure must be directly parsable as literal values corresponding to the structure's data type.

The type data type is convenient for schemas built upon OpenDDL itself in order to define valid type usages in derivative file formats.

A.3.1 Booleans

A boolean value is one of the keywords `false` or `true`, as shown in Figure A.3. The numerical values `0` and `1` may also be specified, and they are equivalent to `false` or `true`, respectively.

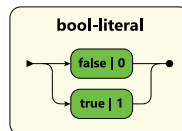


Figure A.3. A boolean value is one of the keywords `false` or `true` or one of the equivalent numerical values `0` or `1`.

A.3.2 Integers

Integers can be specified as a decimal number, a hexadecimal number, an octal number, a binary number, or a single-quoted character literal.

Between any two consecutive digits of each type of integer literal, a single underscore character may be inserted as a separator to enhance readability. The presence of underscore characters and their positions have no significance, and they do not affect the value of a literal.

A decimal literal is simply composed of a sequence of numerical digits, as shown in Figure A.4, and leading zeros are permitted.

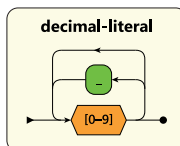


Figure A.4. A decimal literal is any sequence of numerical digits.

A hexadecimal literal is specified by prefixing a number with `0x` or `0X`, as shown in Figure A.5. This is followed, without any intervening whitespace, by any number of hexadecimal digits, shown in Figure A.6, that don't cause the underlying integer type to overflow. The letters A–F in a hexadecimal literal are not case sensitive.

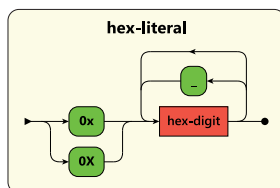


Figure A.5. A hexadecimal literal starts with `0x` or `0X` and continues with one or more hexadecimal digits.

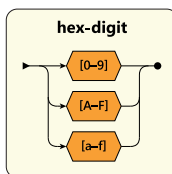


Figure A.6. A hexadecimal digit is a numerical digit 0–9 or a letter A–F (with no regard for case).

An octal literal is specified by prefixing a number with `0o` or `0O`, as shown in Figure A.7. This is followed, without any intervening whitespace, by any number of digits between 0 and 7, inclusive, that don't cause the underlying integer type to overflow.

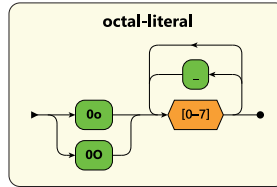


Figure A.7. An octal literal starts with `0o` or `0O` and continues with one or more octal digits.

A binary literal is specified by prefixing a number with `0b` or `0B`, as shown in Figure A.8. This is followed, without any intervening whitespace, by any number of zeros and ones that don't cause the underlying integer type to overflow.

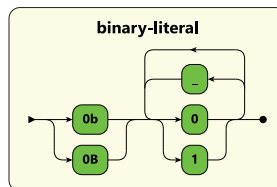


Figure A.8. A binary literal starts with `0b` or `0B` and continues with one or more binary digits.

A character literal is specified by a sequence of printable ASCII characters enclosed in single quotes, as shown in Figure A.9. OpenDDL supports the escape sequences listed in Table A.2 and illustrated in Figure A.10. Escape sequences may be used to generate control characters or arbitrary byte values. The single quote (`'`) and backslash (`\`) characters cannot be represented directly and must be encoded with escape sequences. The `\x` escape sequence is always followed by exactly two hexadecimal digits. Each character (after resolving escape sequences) corresponds to exactly one byte in the resulting integer value, and the right-most character corresponds to the least significant byte.

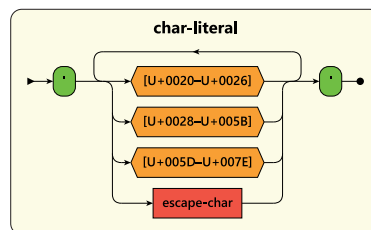


Figure A.9. A character literal is composed of a sequence of printable ASCII characters enclosed in single quotes. The single quote (`'`) and backslash (`\`) characters cannot be represented directly and must be encoded with escape sequences.

Escape Sequence	ASCII Code	Description
\"	0x22	Double quote
\'	0x27	Single quote
\?	0x3F	Question mark
\\	0x5C	Backslash
\a	0x07	Bell
\b	0x08	Backspace
\f	0x0C	Formfeed
\n	0x0A	Newline
\r	0x0D	Carriage return
\t	0x09	Horizontal tab
\v	0x0B	Vertical tab
\xhh	–	Byte value specified by the two hex digits <i>hh</i>

Table A.2. These are the escape sequences supported by OpenDDL for character literals.

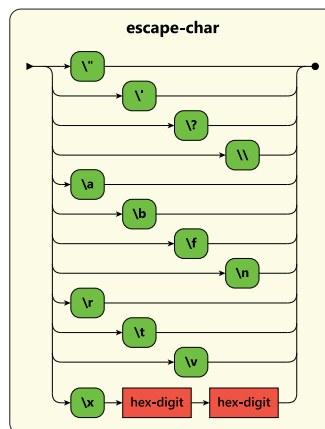


Figure A.10. An escape character consists of a backslash (\) followed by a single character code. In the case of the \x character code, the escape sequence includes exactly two additional hexadecimal digits.

An integer literal is composed of an optional plus or minus sign followed by a decimal, hexadecimal, octal, binary, or character literal, as shown in Figure A.11.

[*Example* — In the following code, the same 32-bit unsigned integer value is repeated five times using different literal types: a decimal literal, a hexadecimal literal, an octal literal, a binary literal, and a character literal.

```
uint32
{
    1094861636,
    0x41424344,
    0o10120441504,
    0b0100_0001_0100_0010_0100_0011_0100_0100,
    'ABCD'
}
```

— *End example*]

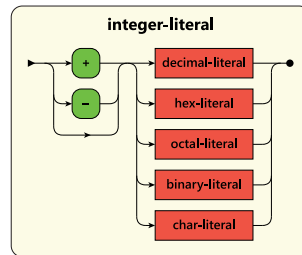


Figure A.11. An integer literal is composed of an optional sign followed by a decimal, hexadecimal, octal, binary, or character literal.

A.3.3 Floating-Point Numbers

Floating-point numbers can be specified as a decimal number with or without a decimal point and fraction, and with or without a trailing exponent, as shown in Figure A.12. Floating-point numbers may also be specified as hexadecimal, octal, or binary literals representing the underlying bit pattern of the number. This is particularly useful for lossless exchange of floating-point data since round-off errors possible in the conversion to and from a decimal representation are avoided. Using a hexadecimal, octal, or binary representation is also the only way to specify a floating-point infinity or not-a-number (NaN) value.

As with integer literals, an underscore character may be inserted between any two consecutive numerical digits in a floating-point literal to enhance readability. Underscore characters are ignored and do not affect the value of a literal.

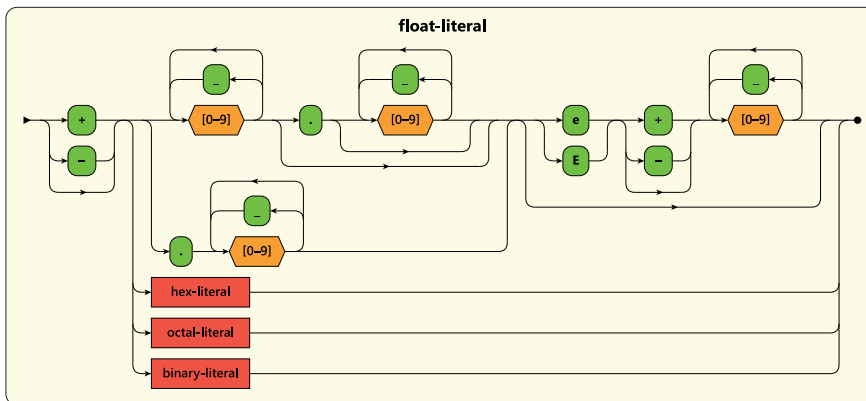


Figure A.12. A floating-point literal is composed of an optional sign followed by a number with or without a decimal point and an optional exponent. Hexadecimal, octal, and binary literals representing the underlying bit pattern are also accepted.

A.3.4 Strings

Strings are composed of a sequence of characters enclosed in double quotes, as shown in Figure A.13. Unicode values (encoded as UTF-8) in the following ranges may be directly included in a string literal:

- [U+0020, U+0021]
- [U+0023, U+005B]
- [U+005D, U+007E]
- [U+00A0, U+D7FF]
- [U+E000, U+FFFF]
- [U+010000, U+10FFFF]

This is the only place where non-ASCII characters are allowed other than in comments.

A string may contain the escape sequences defined for character literals (see Figure A.10). The double quote (") and backslash (\) characters cannot be represented directly and must be encoded with escape sequences. String literals also support the `\u` escape sequence, which specifies a nonzero Unicode character using exactly four hexadecimal digits immediately following the `u`. In order to support Unicode characters outside the Basic Multilingual Plane (BMP), a six-digit code can be specified by using an uppercase `U`. The `\U` escape sequence must be followed by exactly six hexadecimal digits that specify a value in the range `[0x000001, 0x10FFFF]`.

Multiple string literals may be placed adjacent to each other with or without intervening whitespace, and this results in concatenation.

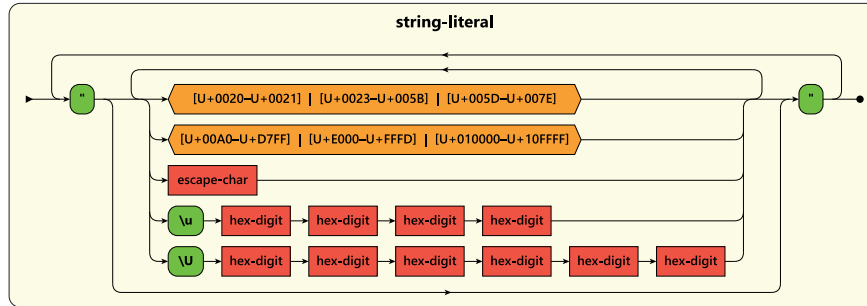


Figure A.13. A string literal is composed of a sequence of Unicode characters enclosed in double quotes. The double-quote ("), backslash (\), and non-printing control characters are excluded from the set of characters that can be directly represented. A string may contain the same escape characters as a character literal as well as additional Unicode escape sequences. Adjacent strings are concatenated.

A.3.5 Base64 Data

Raw binary data can be expressed in the Base64 format. As shown in Figure A.14, Base64 data consists of a sequence of characters composed from the set $\{A-Z, a-z, 0-9, +, /\}$. Each of the 64 characters in the encoding set corresponds to the 6-bit value assigned to it in Table A.3.

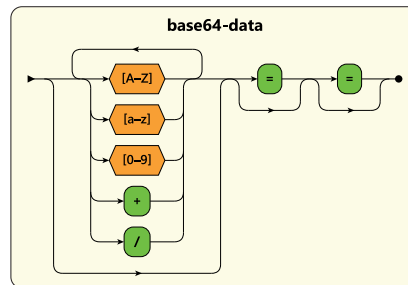


Figure A.14. Base64 data is composed of uppercase and lowercase letters, numbers, the plus symbol, and the slash symbol. There may be up to two equal signs for padding at the end.

The number of encoded characters in a block of Base64 data must be 0, 2, or 3 modulo 4. Each group of four characters corresponds to exactly three decoded bytes having values determined as follows.

- The value of the first byte is given by the six bits encoded by the first character concatenated with the two most significant bits encoded by the second character.
- The value of the second byte is given by the four least significant bits encoded by the second character concatenated with the four most significant bits encoded by the third character.
- The value of the third byte is given by the two least significant bits encoded by the third character concatenated with the six bits encoded by the fourth character.

Char	Value	Char	Value	Char	Value	Char	Value
A	0	Q	16	g	32	w	48
B	1	R	17	h	33	x	49
C	2	S	18	i	34	y	50
D	3	T	19	j	35	z	51
E	4	U	20	k	36	0	52
F	5	V	21	l	37	1	53
G	6	W	22	m	38	2	54
H	7	X	23	n	39	3	55
I	8	Y	24	o	40	4	56
J	9	Z	25	p	41	5	57
K	10	a	26	q	42	6	58
L	11	b	27	r	43	7	59
M	12	c	28	s	44	8	60
N	13	d	29	t	45	9	61
O	14	e	30	u	46	+	62
P	15	f	31	v	47	/	63

Table A.3. These are the 64 character values used in Base64 data.

If the number of encoded characters is 2 modulo 4, then the final two characters produce a single byte of decoded data, and the four least significant bits encoded by the second character are discarded. In this case, the encoded Base64 data may end with two equals sign characters as padding to make the total number of encoded characters a multiple of four. This padding is not required, and it is ignored if it is present.

If the number of encoded characters is 3 modulo 4, then the final three characters produce two bytes of decoded data, and the two least significant bits encoded by the third character are discarded. In this case, the encoded Base64 data may end with one equals sign character as padding to make the total number of encoded characters a multiple of four. This padding is not required, and it is ignored if it is present.

Whitespace may appear anywhere inside Base64 data, and it is ignored. Because the forward slash character has a specific meaning in the Base64 format, comments are not permitted to occur inside Base64 data.

A.4 Identifiers

An *identifier* is a sequence of characters composed from the set {A–Z, a–z, 0–9, _}, as shown in Figure A.15. That is, an identifier is composed of uppercase and lowercase roman letters, the numbers 0 through 9, and the underscore. An identifier cannot begin with a number.

Identifiers are used to specify structure types, names, properties, and data states. The identifiers used for the 16 primitive data types listed in Table A.1 are reserved as structure types, but they can still be used as names, properties, and data states.

All identifiers consisting of a single lowercase letter followed by zero or more numerical digits are reserved as structure types for future use by the language. A derivative format may define any other identifier as the type of a derived structure.

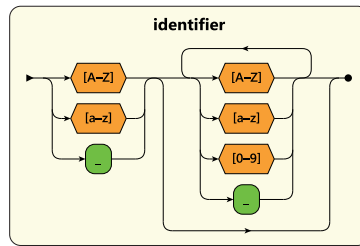


Figure A.15. An identifier is composed of uppercase and lowercase roman letters, the numbers 0 through 9, and the underscore.

A.5 Names

Any structure may have a *name*. Names are used to identify specific structures so they can be referenced from within primitive structures or through property values. A name can be a global name or a local name. Each global name must be unique among all global names used inside the file containing it, and each local name must be unique among all local names used by its siblings in the structure tree. Local names can be reused inside different structures, and they can duplicate global names.

As shown in Figure A.16, a name is composed of either a dollar sign character (\$) or percent sign character (%) followed by an identifier with no intervening whitespace. A name that begins with a dollar sign is a global name, and a name that begins with a percent sign is a local name. A name is assigned to a structure by placing it immediately after the structure identifier (and no whitespace is technically required before the dollar sign). [*Example* —

```
Vertex $apex
{
  float {1.0, 2.0, 3.0}
}
```

The `Vertex` structure has the global name `$apex`. This structure can be referenced from elsewhere in the file by using the name `$apex` as a value of the `ref` type. — *End example*]

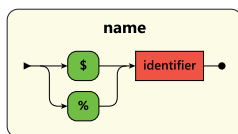


Figure A.16. A name is composed of either a dollar sign character (\$) or a percent sign character (%) followed by an identifier with no intervening whitespace.

A.6 References

A *reference* is a value that forms a link to a specific structure within an OpenDDL file. If the target structure has a global name, then the value of a reference to it is simply the name of the structure, beginning with the dollar sign character. If the target structure has a local name, then the value of a reference to it depends on the scope in which the reference appears. If the reference appears inside a structure that is a sibling of the target structure, then its value is the name of the target structure, beginning with the percent sign character. Otherwise, the value of the reference consists of a sequence of names, as shown in Figure A.17, that identify a sequence of structures along a branch in a tree of structures. Only the first name in the sequence can be a global name, and the rest must be local names.

The value of a reference can also be keyword `null` to indicate that a reference has no target structure.

[*Example* — In the following code, the structure types `Person`, `Name`, and `Friends` are defined by a derivative format. References are used to link people to the data structures representing their friends.

```

Person $charles
{
  Name {string {"Charles"}}
  Friends {ref {$alice, $bob}}
}

Person $alice {...}
Person $bob {...}

```

— *End example*]

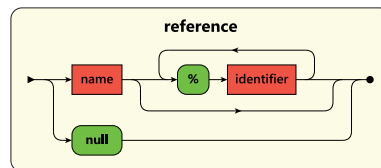


Figure A.17. A reference is either the name of a structure or the keyword `null`. A structure may be identified by a sequence of names providing the path to the target along a branch in a tree of structures.

A.7 Primitive Data

Primitive structures contain homogeneous data of a single primitive data type in one of three possible forms.

A.7.1 Flat Data

The data contained in a primitive structure may consist of a flat, comma-separated list individual literal values, as shown in Figure A.18. The size of the list is unbounded. In this case, the structure identifier is not followed by brackets, but only an optional name and the data itself enclosed in braces.

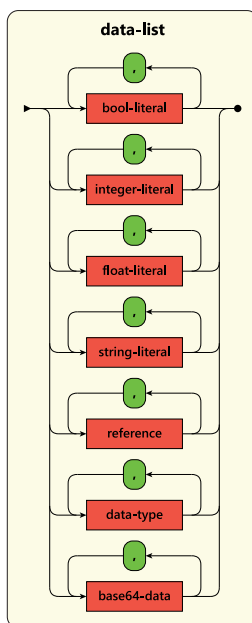


Figure A.18. The data payload of a primitive structure may be a homogeneous list of literal values separated by commas.

Note that an implementation would use its knowledge of the primitive structure's data type to choose only a single rule in Figure A.18, as opposed to allowing any of the types of data to appear inside the braces. (It is also not possible to disambiguate among the numerical data types without some extra information.)

A.7.2 Subarray Data

The data contained in a primitive structure may also be specified as a comma-separated list of *subarrays* of literal values, as shown in Figure A.19. The size of the subarrays is specified by placing a positive

integer value inside brackets immediately following the structure identifier, preceding the structure's optional name. The data belonging to each subarray is then specified as a comma-separated list of values enclosed in braces. In this case, the identifiers shown in Figure A.19 do not apply and may not appear in the structure's data.

As before, an implementation would choose only a single rule in Figure A.19 based on the type of the primitive structure.

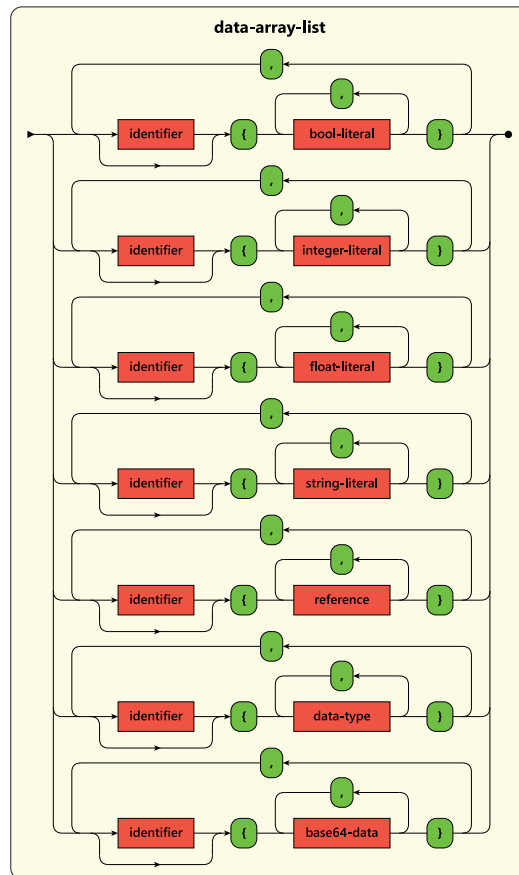


Figure A.19. A data payload may consist of a list of subarrays separated by commas. Each subarray contains a homogeneous array of values enclosed in braces.

The number of elements in each subarray must always match the array size specified inside the brackets following the primitive type identifier. If the array size is one, then the braces are still required. While the size of the subarrays is fixed, the total number of subarrays is unbounded.

[*Example* — Suppose that a `VertexArray` structure expects to contain an array of 3D positions, each of which is specified as an array of three floating-point values. This would be written as follows.

```
VertexArray
{
  float[3]
  {
    {1.0, 2.0, 3.0}, {0.5, 0.0, 0.5}, {0.0, -1.0, 4.0}
  }
}
```

— *End example*]

A.7.3 Data States

Primitive structures containing subarrays may also specify *data states* that associate a particular state with each subarray. The meaning of data states are defined by a derivative format. The presence of data states is indicated by writing an asterisk (*) immediately after the subarray size enclosed in brackets. In this case, a state identifier may precede any subarray in the data payload. If a state identifier is omitted for any particular subarray, then the state associated with the preceding subarray continues to apply. The initial state is defined by the derivative format.

[*Example* — The following `Path` structure provides an example in which data states are used to specify how individual points in a list are to be interpreted. In this example, the state `M` causes the drawing position to be moved to the associated point, the state `L` causes a line to be drawn to the associated point, and the state `C` causes a cubic Bézier curve to be drawn using the preceding point and the following three points.

```
Path
{
  float[2]*
  {
    M{1.0, 1.0}, L{2.0, 1.0}, C{3.0, 1.0}, {3.0, 2.0}, {2.0, 3.0}
  }
}
```

— *End example*]

A.8 Properties

A derived structure may accept one or more *properties* that can be specified separately from the data contained inside the structure. Properties are written in a comma-separated list inside parentheses following the name of the structure, or just following the structure identifier if there is no name. As shown in Figure A.20, each property is composed of a property identifier followed by an equals sign character (=) and the literal value of the property. The type of the property's value must be specified by some external source of information such as a schema or the implementation of the derivative format. For example, a string cannot be specified for a property that was expecting an integer. The specified type determines which subrule in Figure A.20 is applied, and a mismatch must be detected at the time that the property is parsed.

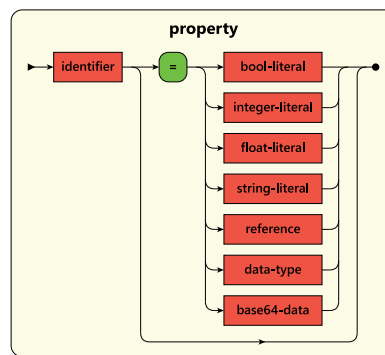


Figure A.20. A property is composed of an identifier followed by an equals character (=) and the value of the property.

[*Example* — Suppose that a data structure called Mesh accepts a property called lod that takes an integer representing the level of detail to which it pertains. This property would be specified as follows.

```
Mesh (lod = 2)
{
    ...
}
```

If another property called part existed and accepted a string (perhaps to identify a body part), then that property could be added to the list as follows.

```
Mesh (lod = 2, part = "Left Hand")
{
    ...
}
```

— *End example*]

The order in which properties are listed is not significant. Derivative formats may require that certain properties always be specified. Optional properties must always have a default value or be specially handled as being in an unspecified state. The same property can be specified more than once in the same property list, and in such a case, all but the final value specified for the same property must be ignored.

Boolean properties allow a special syntax in which the assignment of a value of `true` or `false` can be omitted. In this case, the presence of the property implies that its value is `true`. This is a useful shorthand notation for Boolean properties having a default value of `false`. Properties having any other type must include an assigned value.

A structure is allowed to have properties that are not recognized by the implementation of a derivative format in order to support extensions. Unrecognized properties must be ignored and must not generate an error. However, the value assigned to such a property must still be parsable as one of the primitive data types.

The syntax does not allow primitive structures to have a property list.

A.9 Comments and Whitespace

The language supports C++-style block comments and single-line comments as follows:

- Any occurrence of `/*` begins a comment that ends immediately after the next occurrence of `*/`. Such comments do not nest.
- Any occurrence of `//` begins a comment that ends immediately after the next newline character.

If any sequence `/*`, `*/`, or `//` appears inside a character literal or string literal, then it is part of the literal value and not treated as a comment.

Comments may include any Unicode characters encoded as UTF-8. The only other place where non-ASCII characters are allowed is inside a string literal (see Section A.3.4).

Comments cannot occur inside base64 data (see Section A.3.5).

All characters having a value in the range `[1, 32]` (which includes the space, tab, newline, and carriage return characters), as well as all characters belonging to comments, are considered to be whitespace in OpenDDL. Any arbitrarily long contiguous sequence of whitespace characters is equivalent to a single space character.

A.10 Formal Grammar

For reference, the formal grammar defining the OpenDDL syntax using Backus-Naur Form and regular expressions is shown in Listing A.1. The figures displayed throughout this specification precisely correspond to this grammar. A syntactically valid OpenDDL file satisfies the `file` rule at the end of the listing.

Listing A.1. This is the formal grammar defining the OpenDDL syntax.

<code>identifier</code>	<code>::= [A-Za-z_] [0-9A-Za-z_]*</code>
<code>name</code>	<code>::= (" \$" "%") identifier</code>
<code>reference</code>	<code>::= name ("% identifier)* "null"</code>
<code>hex-digit</code>	<code>::= [0-9A-Fa-f]</code>
<code>escape-char</code>	<code>::= '\ ' '\ ' '\?' '\\ ' '\a' '\b' '\f' '\n' '\r' '\t' '\v' '\x' hex-digit hex-digit</code>
<code>bool-literal</code>	<code>::= "false" "0" "true" "1"</code>
<code>decimal-literal</code>	<code>::= [0-9] ("_"? [0-9])*</code>
<code>hex-literal</code>	<code>::= ("0x" "0X") hex-digit ("_"? hex-digit)*</code>
<code>octal-literal</code>	<code>::= ("0o" "0O") [0-7] ("_"? [0-7])*</code>
<code>binary-literal</code>	<code>::= ("0b" "0B") ("0" "1") ("_"? ("0" "1"))*</code>
<code>char-literal</code>	<code>::= "' ' ([#x20-#x26#x28-#x5B#x5D-#x7E] escape-char)+ "'</code>
<code>integer-literal</code>	<code>::= ("+" "-")? (decimal-literal hex-literal octal-literal binary-literal char-literal)</code>
<code>float-literal</code>	<code>::= ("+" "-")? (([0-9] ("_"? [0-9])* ("." [0-9] ("_"? [0-9])*)?)? "." [0-9] ("_"? [0-9])*) (("e" "E") ("+" "-")? [0-9] ("_"? [0-9])*)? hex-literal octal-literal binary-literal)</code>
<code>string-literal</code>	<code>::= (' ' ([#x20-#x21#x23-#x5B#x5D-#x7E#xA0-#xD7FF#xE000- #xFFFF#x010000-#x10FFFF] escape-char "\u" hex-digit hex-digit hex-digit hex-digit</code>

```

| "\U" hex-digit hex-digit hex-digit hex-digit
hex-digit hex-digit)* '''')+

data-type ::= "bool" | "b" | "int8" | "i8" | "int16" | "i16"
| "int32" | "i32" | "int64" | "i64" | "uint8" | "u8"
| "uint16" | "u16" | "uint32" | "u32" | "uint64" | "u64"
| "half" | "h" | "float" | "f" | "double" | "d" | "float16"
| "f16" | "float32" | "f32" | "float64" | "f64" | "string"
| "s" | "ref" | "r" | "type" | "t" | "base64" | "z"

base64-data ::= [A-Za-z0-9]* "="? "="?

data-list ::= bool-literal ("," bool-literal)*
| integer-literal ("," integer-literal)*
| float-literal ("," float-literal)*
| string-literal ("," string-literal)*
| reference ("," reference)*
| data-type ("," data-type)*
| base64-data ("," base64-data)*

data-array-list ::= identifier? "{" bool-literal (","
bool-literal)* "}" ("," identifier? "{"
bool-literal ("," bool-literal)* "}")*
| identifier? "{" integer-literal (","
integer-literal)* "}" ("," identifier? "{"
integer-literal ("," integer-literal)* "}")*
| identifier? "{" float-literal (","
float-literal)* "}" ("," identifier? "{"
float-literal ("," float-literal)* "}")*
| identifier? "{" string-literal (","
string-literal)* "}" ("," identifier? "{"
string-literal ("," string-literal)* "}")*
| identifier? "{" reference (","
reference)* "}" ("," identifier? "{"
reference ("," reference)* "}")*
| identifier? "{" data-type (","
data-type)* "}" ("," identifier? "{"
data-type ("," data-type)* "}")*
| identifier? "{" base64-data (","
base64-data)* "}" ("," identifier? "{"
base64-data ("," base64-data)* "}")*

property ::= identifier ("=" (bool-literal | integer-literal
| float-literal | string-literal | reference
| data-type | base64-data))?

structure ::= data-type (name? "{" data-list? "}"
| "[" integer-literal "]" "*" name?

```

```
        "{" data-array-list? "}")
    | identifier name? ("(" (property ("," property)*)?
        ")")? "{" structure* "}"

file      ::= structure*
```


B

Revision History

This appendix lists the changes that have been made to the OpenGEX format since the original specification of version 1.0.

Version 3.0

The following changes were made in OpenGEX version 3.0.

- The Open Data Description Language was updated to version 3.0.
- The "red", "green", "blue", and "white" keys were added to the `Metric` structure to specify the chromaticity coordinates of the red, green, and blue primary colors and the white point of the color space used by an OpenGEX file.
- The `Spectrum` structure was added. This can appear anywhere that a `Color` structure can appear.
- Many new attribute types were added to the `Color`, `Param`, `Spectrum`, and `Texture` substructures that can appear inside a `Material` structure.
- The `swizzle`, `x_address`, `y_address`, `z_address`, and `border` properties were added to the `Texture` structure.
- The `Transform` structure now accepts matrix data having 16, 12, 9, 6, or 4 entries.
- The `Extension` structure was removed because it is no longer necessary under OpenDDL 3.0.

Version 2.0

The following changes were made in OpenGEX version 2.0.

- The Open Data Description Language was updated to version 2.0.
- The "forward" key was added to the `Metric` structure to specify the forward direction for models such as characters and vehicles.

- The exact calculations necessary for converting between y and z up directions was added to the specification for the `Metric` structure.
- The "fov x " and "fov y " parameters were added to the `CameraObject` structure.

Version 1.1

The following changes were made in OpenGEX version 1.1.

- The `Extension` structure was added. This can appear anywhere in an OpenGEX file as a container for application-specific data.
- The `Clip` structure was added. This is used to store information that applies to an animation clip as a whole, and it appears at the top level of an OpenGEX file.
- It is no longer a requirement that multiple `Animation` structures belonging to the same parent structure have unique values for their `clip` properties.
- The `curve` property of the `Value` structure now accepts "constant" to indicate that the value remains constant until the next time key.
- As a substructure of the `Material` structure, the `attrib` property of the `Texture` structure now accepts "specular_power" to indicate that the texture modulates the specular power.
- The data contained inside a `VertexArray` structure may now use any floating-point type, whereas only 32-bit floating-point values were previously accepted.
- The `MorphWeight` structure was added. This can appear inside a `GeometryNode` structure to specify the weight of a single morph target, and it can be the target of an animation track.
- The purpose of the `Morph` structure has been changed. This is now used to store information that applies to a morph target as a whole, and it can appear inside a `GeometryObject` structure.