



BSI Standards Publication

Information technology — Runtime 3D asset delivery format — Khronos glTF™ 2.0

National foreword

This British Standard is the UK implementation of ISO/IEC 12113:2022.

The UK participation in its preparation was entrusted to Technical Committee ICT/1, Information systems co-ordination.

A list of organizations represented on this committee can be obtained on request to its committee manager.

Contractual and legal considerations

This publication has been prepared in good faith, however no representation, warranty, assurance or undertaking (express or implied) is or will be made, and no responsibility or liability is or will be accepted by BSI in relation to the adequacy, accuracy, completeness or reasonableness of this publication. All and any such responsibility and liability is expressly disclaimed to the full extent permitted by the law.

This publication is provided as is, and is to be used at the recipient's own risk.

The recipient is advised to consider seeking professional guidance with respect to its use of this publication.

This publication is not intended to constitute a contract. Users are responsible for its correct application.

© The British Standards Institution 2024
Published by BSI Standards Limited 2024

ISBN 978 0 539 19881 2

ICS 35.040.30

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 31 October 2024.

Amendments/corrigenda issued since publication

Date	Text affected
------	---------------

Information technology — Runtime 3D asset delivery format — Khronos glTF™ 2.0

*Technologies de l'information — Format de livraison d'actifs 3D
d'exécution — Khronos glTF™ 2.0*



Reference number
ISO/IEC 12113:2022(E)

AVA Documents
Document Sharing Hub
© ISO/IEC 2022



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2022

All rights reserved. Unless otherwise specified, or required in the context of its implementation, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents) or the IEC list of patent declarations received (see <https://patents.iec.ch>).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT), see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Khronos (as glTF™ 2.0 Specification) and drafted in accordance with its editorial rules. It was adopted, under the JTC 1 PAS procedure, by Joint Technical Committee ISO/IEC JTC 1, *Information technology*.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

Table of Contents

1. Foreword	1
2. Introduction	2
2.1. General	2
2.2. Document Conventions	2
2.2.1. Normative Terminology and References	2
2.2.2. Informative Language	2
2.2.3. Technical Terminology	3
2.2.4. Normative References	5
2.2.4.1. External Specifications	5
2.2.4.2. Media Type Registrations	6
2.3. Motivation and Design Goals (Informative)	7
2.4. glTF Basics	8
2.5. Versioning	8
2.6. File Extensions and Media Types	8
2.7. JSON Encoding	9
2.8. URIs	11
3. Concepts	13
3.1. General	13
3.2. Asset	13
3.3. Indices and Names	14
3.4. Coordinate System and Units	15
3.5. Scenes	15
3.5.1. Overview	15
3.5.2. Nodes and Hierarchy	16
3.5.3. Transformations	17
3.6. Binary Data Storage	19
3.6.1. Buffers and Buffer Views	19
3.6.1.1. Overview	19
3.6.1.2. GLB-stored Buffer	21
3.6.2. Accessors	22
3.6.2.1. Overview	22
3.6.2.2. Accessor Data Types	23
3.6.2.3. Sparse Accessors	24
3.6.2.4. Data Alignment	25
3.6.2.5. Accessors Bounds	28
3.7. Geometry	28
3.7.1. Overview	28
3.7.2. Meshes	28

3.7.2.1. Overview	28
3.7.2.2. Morph Targets	32
3.7.3. Skins	35
3.7.3.1. Overview	35
3.7.3.2. Joint Hierarchy	36
3.7.3.3. Skinned Mesh Attributes	37
3.7.4. Instantiation	39
3.8. Texture Data	41
3.8.1. Overview	41
3.8.2. Textures	42
3.8.3. Images	42
3.8.4. Samplers	44
3.8.4.1. Overview	44
3.8.4.2. Filtering	44
3.8.4.3. Wrapping	45
3.8.4.4. Example	45
3.8.4.5. Non-power-of-two Textures	46
3.9. Materials	46
3.9.1. Overview	46
3.9.2. Metallic-Roughness Material	47
3.9.3. Additional Textures	49
3.9.4. Alpha Coverage	51
3.9.5. Double Sided	52
3.9.6. Default Material	52
3.9.7. Point and Line Materials	52
3.10. Cameras	53
3.10.1. Overview	53
3.10.2. View Matrix	53
3.10.3. Projection Matrices	53
3.10.3.1. Overview	53
3.10.3.2. Infinite perspective projection	54
3.10.3.3. Finite perspective projection	55
3.10.3.4. Orthographic projection	55
3.11. Animations	55
3.12. Specifying Extensions	61
4. GLB File Format Specification	63
4.1. General (Informative)	63
4.2. Structure	63
4.3. File Extension & Media Type	63
4.4. Binary glTF Layout	63
4.4.1. Overview	63

4.4.2. Header	64
4.4.3. Chunks	64
4.4.3.1. Overview	64
4.4.3.2. Structured JSON Content	65
4.4.3.3. Binary buffer	65
5. Properties Reference	66
5.1. Accessor	66
5.1.1. accessor.bufferView	67
5.1.2. accessor.byteOffset	67
5.1.3. accessor.componentType	67
5.1.4. accessor.normalized	68
5.1.5. accessor.count	68
5.1.6. accessor.type	68
5.1.7. accessor.max	68
5.1.8. accessor.min	69
5.1.9. accessor.sparse	69
5.1.10. accessor.name	69
5.1.11. accessor.extensions	69
5.1.12. accessor.extras	69
5.2. Accessor Sparse	70
5.2.1. accessor.sparse.count	70
5.2.2. accessor.sparse.indices	70
5.2.3. accessor.sparse.values	71
5.2.4. accessor.sparse.extensions	71
5.2.5. accessor.sparse.extras	71
5.3. Accessor Sparse Indices	71
5.3.1. accessor.sparse.indices.bufferView	72
5.3.2. accessor.sparse.indices.byteOffset	72
5.3.3. accessor.sparse.indices.componentType	72
5.3.4. accessor.sparse.indices.extensions	73
5.3.5. accessor.sparse.indices.extras	73
5.4. Accessor Sparse Values	73
5.4.1. accessor.sparse.values.bufferView	74
5.4.2. accessor.sparse.values.byteOffset	74
5.4.3. accessor.sparse.values.extensions	74
5.4.4. accessor.sparse.values.extras	74
5.5. Animation	74
5.5.1. animation.channels	75
5.5.2. animation.samplers	75
5.5.3. animation.name	76
5.5.4. animation.extensions	76

5.5.5. animation.extras	76
5.6. Animation Channel	76
5.6.1. animation.channel.sampler	77
5.6.2. animation.channel.target	77
5.6.3. animation.channel.extensions	77
5.6.4. animation.channel.extras	77
5.7. Animation Channel Target	77
5.7.1. animation.channel.target.node	78
5.7.2. animation.channel.target.path	79
5.7.3. animation.channel.target.extensions	79
5.7.4. animation.channel.target.extras	79
5.8. Animation Sampler	79
5.8.1. animation.sampler.input	80
5.8.2. animation.sampler.interpolation	80
5.8.3. animation.sampler.output	81
5.8.4. animation.sampler.extensions	81
5.8.5. animation.sampler.extras	81
5.9. Asset	81
5.9.1. asset.copyright	82
5.9.2. asset.generator	82
5.9.3. asset.version	82
5.9.4. asset.minVersion	82
5.9.5. asset.extensions	83
5.9.6. asset.extras	83
5.10. Buffer	83
5.10.1. buffer.uri	83
5.10.2. buffer.byteLength	84
5.10.3. buffer.name	84
5.10.4. buffer.extensions	84
5.10.5. buffer.extras	84
5.11. Buffer View	84
5.11.1. bufferView.buffer	85
5.11.2. bufferView.byteOffset	85
5.11.3. bufferView.byteLength	85
5.11.4. bufferView.byteStride	86
5.11.5. bufferView.target	86
5.11.6. bufferView.name	86
5.11.7. bufferView.extensions	86
5.11.8. bufferView.extras	86
5.12. Camera	87
5.12.1. camera.orthographic	87

5.12.2. camera.perspective	88
5.12.3. camera.type	88
5.12.4. camera.name	88
5.12.5. camera.extensions	88
5.12.6. camera.extras	88
5.13. Camera Orthographic	89
5.13.1. camera.orthographic.xmag	89
5.13.2. camera.orthographic.ymag	90
5.13.3. camera.orthographic.zfar	90
5.13.4. camera.orthographic.znear	90
5.13.5. camera.orthographic.extensions	90
5.13.6. camera.orthographic.extras	90
5.14. Camera Perspective	91
5.14.1. camera.perspective.aspectRatio	91
5.14.2. camera.perspective.yfov	91
5.14.3. camera.perspective.zfar	92
5.14.4. camera.perspective.znear	92
5.14.5. camera.perspective.extensions	92
5.14.6. camera.perspective.extras	92
5.15. Extension	92
5.16. Extras	93
5.17. glTF	93
5.17.1. glTF.extensionsUsed	94
5.17.2. glTF.extensionsRequired	94
5.17.3. glTF.accessors	94
5.17.4. glTF.animations	94
5.17.5. glTF.asset	94
5.17.6. glTF.buffers	95
5.17.7. glTF.bufferViews	95
5.17.8. glTF.cameras	95
5.17.9. glTF.images	95
5.17.10. glTF.materials	95
5.17.11. glTF.meshes	95
5.17.12. glTF.nodes	96
5.17.13. glTF.samplers	96
5.17.14. glTF.scene	96
5.17.15. glTF.scenes	96
5.17.16. glTF.skins	96
5.17.17. glTF.textures	96
5.17.18. glTF.extensions	96
5.17.19. glTF.extras	97

5.18. Image	97
5.18.1. image.uri	98
5.18.2. image.mimeType	98
5.18.3. image.bufferView	98
5.18.4. image.name	98
5.18.5. image.extensions	98
5.18.6. image.extras	99
5.19. Material	99
5.19.1. material.name	100
5.19.2. material.extensions	100
5.19.3. material.extras	100
5.19.4. material.pbrMetallicRoughness	100
5.19.5. material.normalTexture	100
5.19.6. material.occlusionTexture	101
5.19.7. material.emissiveTexture	101
5.19.8. material.emissiveFactor	101
5.19.9. material.alphaMode	101
5.19.10. material.alphaCutoff	102
5.19.11. material.doubleSided	102
5.20. Material Normal Texture Info	102
5.20.1. material.normalTextureInfo.index	103
5.20.2. material.normalTextureInfo.texCoord	103
5.20.3. material.normalTextureInfo.scale	103
5.20.4. material.normalTextureInfo.extensions	103
5.20.5. material.normalTextureInfo.extras	104
5.21. Material Occlusion Texture Info	104
5.21.1. material.occlusionTextureInfo.index	104
5.21.2. material.occlusionTextureInfo.texCoord	104
5.21.3. material.occlusionTextureInfo.strength	105
5.21.4. material.occlusionTextureInfo.extensions	105
5.21.5. material.occlusionTextureInfo.extras	105
5.22. Material PBR Metallic Roughness	105
5.22.1. material.pbrMetallicRoughness.baseColorFactor	106
5.22.2. material.pbrMetallicRoughness.baseColorTexture	106
5.22.3. material.pbrMetallicRoughness.metallicFactor	106
5.22.4. material.pbrMetallicRoughness.roughnessFactor	107
5.22.5. material.pbrMetallicRoughness.metallicRoughnessTexture	107
5.22.6. material.pbrMetallicRoughness.extensions	107
5.22.7. material.pbrMetallicRoughness.extras	107
5.23. Mesh	108
5.23.1. mesh.primitives	108

5.23.2. mesh.weights	108
5.23.3. mesh.name	108
5.23.4. mesh.extensions	109
5.23.5. mesh.extras	109
5.24. Mesh Primitive	109
5.24.1. mesh.primitive.attributes	110
5.24.2. mesh.primitive.indices	110
5.24.3. mesh.primitive.material	110
5.24.4. mesh.primitive.mode	110
5.24.5. mesh.primitive.targets	111
5.24.6. mesh.primitive.extensions	111
5.24.7. mesh.primitive.extras	111
5.25. Node	111
5.25.1. node.camera	113
5.25.2. node.children	113
5.25.3. node.skin	113
5.25.4. node.matrix	113
5.25.5. node.mesh	114
5.25.6. node.rotation	114
5.25.7. node.scale	114
5.25.8. node.translation	114
5.25.9. node.weights	114
5.25.10. node.name	114
5.25.11. node.extensions	115
5.25.12. node.extras	115
5.26. Sampler	115
5.26.1. sampler.magFilter	115
5.26.2. sampler.minFilter	116
5.26.3. sampler.wrapS	116
5.26.4. sampler.wrapT	116
5.26.5. sampler.name	117
5.26.6. sampler.extensions	117
5.26.7. sampler.extras	117
5.27. Scene	117
5.27.1. scene.nodes	118
5.27.2. scene.name	118
5.27.3. scene.extensions	118
5.27.4. scene.extras	118
5.28. Skin	118
5.28.1. skin.inverseBindMatrices	119
5.28.2. skin.skeleton	119

5.28.3. skin.joints	119
5.28.4. skin.name	120
5.28.5. skin.extensions	120
5.28.6. skin.extras	120
5.29. Texture	120
5.29.1. texture.sampler	121
5.29.2. texture.source	121
5.29.3. texture.name	121
5.29.4. texture.extensions	122
5.29.5. texture.extras	122
5.30. Texture Info	122
5.30.1. textureInfo.index	122
5.30.2. textureInfo.texCoord	123
5.30.3. textureInfo.extensions	123
5.30.4. textureInfo.extras	123
6. Acknowledgments (Informative)	124
6.1. Editors	124
6.2. Khronos 3D Formats Working Group and Alumni	124
6.3. Special Thanks	124
Appendix A: JSON Schema Reference (Informative)	126
A.1. JSON Schema for Accessor	126
A.2. JSON Schema for Accessor Sparse	130
A.3. JSON Schema for Accessor Sparse Indices	131
A.4. JSON Schema for Accessor Sparse Values	133
A.5. JSON Schema for Animation	134
A.6. JSON Schema for Animation Channel	135
A.7. JSON Schema for Animation Channel Target	136
A.8. JSON Schema for Animation Sampler	137
A.9. JSON Schema for Asset	139
A.10. JSON Schema for Buffer	140
A.11. JSON Schema for Buffer View	141
A.12. JSON Schema for Camera	143
A.13. JSON Schema for Camera Orthographic	145
A.14. JSON Schema for Camera Perspective	146
A.15. JSON Schema for Extension	147
A.16. JSON Schema for Extras	148
A.17. JSON Schema for glTF	149
A.18. JSON Schema for glTF Child of Root Property	153
A.19. JSON Schema for glTF Id	154
A.20. JSON Schema for glTF Property	155
A.21. JSON Schema for Image	159

A.22. JSON Schema for Material	158
A.23. JSON Schema for Material Normal Texture Info	161
A.24. JSON Schema for Material Occlusion Texture Info	162
A.25. JSON Schema for Material PBR Metallic Roughness	163
A.26. JSON Schema for Mesh	165
A.27. JSON Schema for Mesh Primitive	166
A.28. JSON Schema for Node	169
A.29. JSON Schema for Sampler	172
A.30. JSON Schema for Scene	175
A.31. JSON Schema for Skin	176
A.32. JSON Schema for Texture	177
A.33. JSON Schema for Texture Info	178
Appendix B: BRDF Implementation	179
B.1. General	179
B.2. Material Structure	179
B.2.1. Metals	179
B.2.2. Dielectrics	180
B.2.3. Microfacet Surfaces	181
B.2.4. Complete Model	182
B.3. Sample Implementation (Informative)	183
B.3.1. Overview	183
B.3.2. Specular BRDF	183
B.3.3. Diffuse BRDF	184
B.3.4. Fresnel	184
B.3.5. Metal BRDF and Dielectric BRDF	185
B.3.6. Discussion	185
B.3.6.1. Masking-Shadowing Term and Multiple Scattering	185
B.3.6.2. Schlick's Fresnel Approximation	185
B.3.6.3. Coupling Diffuse and Specular Reflection	186
B.4. References	187
Appendix C: Animation Sampler Interpolation Modes	188
C.1. Overview	188
C.2. Step Interpolation	188
C.3. Linear Interpolation	188
C.4. Spherical Linear Interpolation	188
C.5. Cubic Spline Interpolation	189

Chapter 1. Foreword

Copyright 2013-2021 The Khronos Group Inc.

This specification is protected by copyright laws and contains material proprietary to Khronos. Except as described by these terms, it or any components may not be reproduced, republished, distributed, transmitted, displayed, broadcast, or otherwise exploited in any manner without the express prior written permission of Khronos.

This specification has been created under the Khronos Intellectual Property Rights Policy, which is Attachment A of the Khronos Group Membership Agreement available at https://www.khronos.org/files/member_agreement.pdf. Khronos grants a conditional copyright license to use and reproduce the unmodified specification for any purpose, without fee or royalty, EXCEPT no licenses to any patent, trademark or other intellectual property rights are granted under these terms. Parties desiring to implement the specification and make use of Khronos trademarks in relation to that implementation, and receive reciprocal patent license protection under the Khronos IP Policy must become Adopters under the process defined by Khronos for this specification; see <https://www.khronos.org/conformance/adopters/file-format-adopter-program>.

Some parts of this Specification are non-normative through being explicitly identified as purely informative, and do not define requirements necessary for compliance and so are outside the Scope of this Specification.

Where this Specification includes normative references to external documents, only the specifically identified sections and functionality of those external documents are in Scope. Requirements defined by external documents not created by Khronos may contain contributions from non-members of Khronos not covered by the Khronos Intellectual Property Rights Policy.

Khronos makes no, and expressly disclaims any, representations or warranties, express or implied, regarding this specification, including, without limitation: merchantability, fitness for a particular purpose, non-infringement of any intellectual property, correctness, accuracy, completeness, timeliness, and reliability. Under no circumstances will Khronos, or any of its Promoters, Contributors or Members, or their respective partners, officers, directors, employees, agents or representatives be liable for any damages, whether direct, indirect, special or consequential damages for lost revenues, lost profits, or otherwise, arising from or in connection with these materials.

Khronos® and Vulkan® are registered trademarks, and ANARI™, WebGL™, glTF™, NNEF™, OpenVX™, SPIR™, SPIR-V™, SYCL™, OpenVG™ and 3D Commerce™ are trademarks of The Khronos Group Inc. OpenXR™ is a trademark owned by The Khronos Group Inc. and is registered as a trademark in China, the European Union, Japan and the United Kingdom. OpenCL™ is a trademark of Apple Inc. and OpenGL® is a registered trademark and the OpenGL ES™ and OpenGL SC™ logos are trademarks of Hewlett Packard Enterprise used under license by Khronos. ASTC is a trademark of ARM Holdings PLC. All other product names, trademarks, and/or company names are used solely for identification and belong to their respective owners.

Chapter 2. Introduction

2.1. General

This document, referred to as the “glTF Specification” or just the “Specification” hereafter, describes the glTF file format.

glTF is an API-neutral runtime asset delivery format. glTF bridges the gap between 3D content creation tools and modern graphics applications by providing an efficient, extensible, interoperable format for the transmission and loading of 3D content.

2.2. Document Conventions

The glTF Specification is intended for use by both implementers of the asset exporters or converters (e.g., digital content creation tools) and application developers seeking to import or load glTF assets, forming a basis for interoperability between these parties.

Specification text can address either party; typically, the intended audience can be inferred from context, though some sections are defined to address only one of these parties.

Any requirements, prohibitions, recommendations, or options defined by [normative terminology](#) are imposed only on the audience of that text.

2.2.1. Normative Terminology and References

The key words **MUST**, **MUST NOT**, **REQUIRED**, **SHALL**, **SHALL NOT**, **SHOULD**, **SHOULD NOT**, **RECOMMENDED**, **MAY**, and **OPTIONAL** in this document are to be interpreted as described in [BCP 14](#).

These key words are highlighted in the specification for clarity.

References to external documents are considered normative if the Specification uses any of the normative terms defined in this section to refer to them or their requirements, either as a whole or in part.

2.2.2. Informative Language

Some language in the specification is purely informative, intended to give background or suggestions to implementers or developers.

If an entire chapter or section contains only informative language, its title is suffixed with “(Informative)”. If not designated as informative, all chapters, sections, and appendices in this document are normative.

All Notes, Implementation notes, and Examples are purely informative.

2.2.3. Technical Terminology

The glTF Specification makes use of linear algebra terms such as **axis**, **matrix**, **vector**, etc. to identify certain math constructs and their behaviors as defined in the [International Electrotechnical Vocabulary](#).

The glTF Specification makes use of common engineering and graphics terms such as **image**, **buffer**, **texture**, etc. to identify and describe certain *glTF* constructs and their attributes, states, and behaviors. This section defines the basic meanings of these terms in the context of the Specification. The Specification text provides fuller definitions of the terms and elaborates, extends, or clarifies the definitions. When a term defined in this section is used in normative language within the Specification, the definitions within the Specification govern and supersede any meanings the terms may have in other technical contexts (i.e. outside the Specification).

accessor

An object describing the number and the format of data elements stored in a binary buffer.

animation

An object describing the keyframe data, including timestamps, and the target property affected by it.

back-facing

See facingness.

buffer

An external or embedded resource that represents a linear array of bytes.

buffer view

An object that represents a range of a specific buffer, and optional metadata that controls how the buffer's content is interpreted.

camera

An object defining the projection parameters that are used to render a scene.

facingness

A classification of a triangle as either front-facing or back-facing, depending on the orientation (winding order) of its vertices.

front-facing

See facingness.

image

A two dimensional array of pixels encoded as a standardized bitstream, such as [PNG](#).

indexed geometry

A mesh primitive that uses a separate source of data (index values) to assemble the primitive's topology.

linear blend skinning

A skinning method that computes a per-vertex transformation matrix as a linear weighted sum of transformation matrices of the designated nodes.

material

A parametrized approximation of visual properties of the real-world object being represented by a mesh primitive.

mesh

A collection of mesh primitives.

mesh primitive

An object binding indexed or non-indexed geometry with a material.

mipmap

A set of image representations consecutively reduced by the factor of 2 in each dimension.

morph target

An altered state of a mesh primitive defined as a set of difference values for its vertex attributes.

node

An object defining the hierarchy relations and the local transform of its content.

non-indexed geometry

A mesh primitive that uses linear order of vertex attribute values to assemble the primitive's topology.

normal

A unit XYZ vector defining the perpendicular to the surface.

root node

A node that is not a child of any other node.

sampler

An object that controls how image data is sampled.

scene

An object containing a list of root nodes to render.

skinning

The process of computing and applying individual transforms for each vertex of a mesh primitive.

tangent

A unit XYZ vector defining a tangential direction on the surface.

texture

An object that combines an image and its sampler.

topology type

State that controls how vertices are assembled, e.g. as lists of triangles, strips of lines, etc.

vertex attribute

A property associated with a vertex.

winding order

The relative order in which vertices are defined within a triangle

wrapping

A process of selecting an image pixel based on normalized texture coordinates.

2.2.4. Normative References

The following documents are referenced by normative sections of the specification:

2.2.4.1. External Specifications

Bradner, S., *Key words for use in RFCs to Indicate Requirement Levels*, BCP 14, RFC 2119, March 1997. Leiba, B., *Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words*, BCP 14, RFC 8174, May 2017. <https://www.rfc-editor.org/info/bcp14>

IEC 60050-102 *International Electrotechnical Vocabulary (IEV) - Part 102: Mathematics - General concepts and linear algebra* <https://webstore.iec.ch/publication/160>

IEC 60050-103 *International Electrotechnical Vocabulary (IEV) - Part 103: Mathematics - Functions* <https://webstore.iec.ch/publication/161>

*Note*

An online version of these standards is available at <https://www.electropedia.org/>

The Unicode Consortium, *The Unicode Standard* <https://www.unicode.org/versions/latest/>

Bray, T., Ed., *The JavaScript Object Notation (JSON) Data Interchange Format*, STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>

ISO/IEC 60559 *Floating-point arithmetic* <https://www.iso.org/standard/80985.html>

ISO/IEC 15948 *Portable Network Graphics (PNG): Functional specification* <https://www.iso.org/standard/29581.html>

*Note*

A free version of this standard is available from W3C: <https://www.w3.org/TR/PNG/>

ISO/IEC 10918-1 *Digital compression and coding of continuous-tone still images: Requirements and guidelines* <https://www.iso.org/standard/18902.html>

*Note*

An earlier edition of this standard called ITU Recommendation T.81 is available from W3C: <https://www.w3.org/Graphics/JPEG/itu-t81.pdf>

ISO/IEC 10918-5 *Digital compression and coding of continuous-tone still images: JPEG File Interchange Format (JFIF)* <https://www.iso.org/standard/54989.html>

*Note*

An earlier edition of this standard is available from W3C: <https://www.w3.org/Graphics/JPEG/jif3.pdf>

CIPA DC-008-Translation-2019 *Exchangeable image file format for digital still cameras* https://www.cipa.jp/std/documents/download_e.html?DC-008-Translation-2019-E

Masinter, L., *The "data" URL scheme*, RFC 2397, DOI 10.17487/RFC2397, August 1998, <https://www.rfc-editor.org/info/rfc2397>

Berners-Lee, T., Fielding, R., and L. Masinter, *Uniform Resource Identifier (URI): Generic Syntax*, STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>

Duerst, M. and M. Suignard, *Internationalized Resource Identifiers (IRIs)*, RFC 3987, DOI 10.17487/RFC3987, January 2005, <https://www.rfc-editor.org/info/rfc3987>

Fielding, R., Ed., and J. Reschke, Ed., *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*, RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>

IEC 61966-2-1 *Default RGB colour space - sRGB* <https://webstore.iec.ch/publication/6169>

*Note*

The encoding characteristics of sRGB are freely available from ICC: <https://www.color.org/chardata/rgb/srgb.xalter>

Recommendation ITU-R BT.709-6 *Parameter values for the HDTV standards for production and international programme exchange* <https://www.itu.int/rec/R-REC-BT.709-6-201506-I>

MikkTSpace <https://github.com/mmikk/MikkTSpace>

Thomas Porter and Tom Duff. 1984. *Compositing digital images*. SIGGRAPH Comput. Graph. 18, 3 (July 1984), 253–259. DOI: <https://doi.org/10.1145/964965.808606>

*Note*

A free version of this paper is available from Pixar: <https://graphics.pixar.com/library/Compositing/>

2.2.4.2. Media Type Registrations

IANA. *model/gltf+json Media Type*. <https://www.iana.org/assignments/media-types/model/gltf+json>

IANA. *model/gltf-binary Media Type*. <https://www.iana.org/assignments/media-types/model/gltf-binary>

IANA. *application/gltf-buffer* Media Type. <https://www.iana.org/assignments/media-types/application/gltf-buffer>

IANA. *application/octet-stream* Media Type. <https://www.iana.org/assignments/media-types/application/octet-stream>

Freed, N. and N. Borenstein, *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*, RFC 2046, DOI 10.17487/RFC2046, November 1996, <https://www.rfc-editor.org/info/rfc2046>

IANA. *image/png* Media Type. <https://www.iana.org/assignments/media-types/image/png>

2.3. Motivation and Design Goals (Informative)

glTF is an open interoperable 3D asset ‘transmission’ format that is compact, and efficient to process and render at runtime. glTF 2.0 is designed to be vendor- and runtime-neutral, usable by a wide variety of native and web-based engines and applications regardless of underlying platforms and 3D graphics APIs.

glTF’s focus on run-time efficiency is a different design goal than typical 3D ‘authoring’ formats. Authoring formats are typically more verbose, with higher processing overheads, to carry authoring data that is no longer needed after iterative design is complete. glTF is complementary to authoring formats, providing a common, interoperable distillation target for publishing 3D assets to a wide audience of end users.

A primary goal of glTF is to be deployable on a wide range of devices and platforms, including the web and mobile devices with limited processing and memory resources. glTF can be evolved, to keep pace with growing compute capabilities over time. This helps to foster broad industry consensus on 3D functionality that can be used ubiquitously, including Physically Based Rendering.

glTF combines an easily parsable JSON scene description with one or more binary resources representing geometry, animations, and other rich data. These binary resources can often be loaded directly into GPU buffers with no additional parsing or processing, combining the faithful preservation of full hierarchical scenes, nodes, meshes, cameras, materials, and animations with efficient delivery and fast loading.

glTF has been designed to meet the following goals:

- **Compact file sizes.** The plain text glTF JSON file description is compact and rapid to parse. All large data such as geometry, textures and animations are stored in binary files that are significantly smaller than equivalent text representations.
- **Runtime-independence.** glTF is purely an asset format and does not mandate any runtime behavior. This enables its use by any application for any purpose, including display using any rendering technology, up to and including path tracing renderers.
- **Complete 3D scene representation.** Not restricted to single objects, glTF can represent entire scenes, including nodes, transformations, transform hierarchy, meshes, materials, cameras, and animations.
- **Extensibility.** glTF is fully extensible, enabling the addition of both general-purpose and vendor-specific extensions, including geometry and texture compression. Widely adopted extensions may be considered for integration into future versions of the glTF specification.

The following are outside the scope of glTF 2.0:

- **glTF is not a streaming format.** The binary data in glTF is inherently streamable, and the buffer design allows for fetching data incrementally, but there are no other streaming constructs in glTF 2.0.
- **glTF is not an authoring format.** glTF deliberately does not retain 3D authoring information, in order to preserve runtime efficiency, however glTF files may be ingested by 3D authoring tools for remixing.
- **glTF is not intended to be human-readable**, though by virtue of being represented in JSON, it is developer-friendly.

2.4. glTF Basics

A glTF asset is represented by:

- A JSON-formatted file (**.gltf**) containing a full scene description: node hierarchy, materials, cameras, as well as descriptor information for meshes, animations, and other constructs.
- Binary files (**.bin**) containing geometry, animation, and other buffer-based data.
- Image files (**.jpg**, **.png**) containing texture images.

Binary and image resources **MAY** also be embedded directly in JSON using [Data URI](#) or stored side-by-side with JSON in [GLB](#) container.

A valid glTF asset **MUST** specify its version.

2.5. Versioning

Any updates made to the glTF Specification in a minor version **MUST** be backward and forward compatible. Backward compatibility means that any client implementation that supports loading a glTF 2.x asset will also be able to load a glTF 2.0 asset. Forward compatibility means that a client implementation that only supports glTF 2.0 can load glTF 2.x assets while gracefully ignoring any new features it does not understand.

A minor version update **MAY** introduce new features but **MUST NOT** change any previously existing behavior. Existing functionality **MAY** be deprecated in a minor version update, but it **MUST NOT** be removed.

Major version updates **MAY** be incompatible with previous versions.

2.6. File Extensions and Media Types

- [JSON](#) glTF files **SHOULD** use **.gltf** extension and **model/gltf+json** Media Type.
- glTF files stored in [GLB](#) container **SHOULD** use **.glb** extension and **model/gltf-binary** Media Type.
- Files representing binary buffers **SHOULD** use either:

- **.bin** file extension with **application/octet-stream** Media Type;
- **.bin**, **.glbin**, or **.glbuf** file extensions with **application/glTF-buffer** Media Type.
- **PNG** images **SHOULD** use **.png** file extension with **image/png** Media Type;
 - PNG images **SHOULD NOT** contain animations, non-square pixel ratios, or embedded ICC profiles. Such features, if present, **MUST** be ignored by client implementations.
- **JPEG** images **SHOULD** use **.jpeg** or **.jpg** file extensions with **image/jpeg** Media Type
 - JPEG images **MUST** be compatible with **JPEG File Interchange Format**.
 - JPEG images **SHOULD NOT** contain embedded ICC profiles. If present, embedded ICC profiles **MUST** be ignored by client implementations.
 - **Exchangeable image file format (Exif)** chunks **MAY** be ignored by client implementations.



Implementation Note

Certain Exif chunks, e.g., “Orientation”, may severely impact an asset’s portability.

2.7. JSON Encoding

Although glTF Specification does not define any subset of the **JSON** format, implementations **SHOULD** be aware of its peculiar properties that could affect asset interoperability.

1. glTF JSON data **SHOULD** be written with UTF-8 encoding without BOM. This requirement is not applied when a glTF implementation does not control string encoding. glTF implementations **SHOULD** adhere to **RFC 8259**, Section 8.1. with regards to treating BOM presence.
2. ASCII characters stored in glTF JSON **SHOULD** be written without JSON escaping.



Example

"buffer" instead of **"\u0062\u0075\u0066\u0066\u0065\u0072"**.

3. Non-ASCII characters stored in glTF JSON **MAY** be escaped.

Example

These two examples represent the same glTF JSON data.

```
{
  "asset": {
    "version": "2.0"
  },
  "nodes": [
    {
      "name": "ky6"
    },
    {
      "name": "𐀀𐀀𐀀"
    }
  ]
}
```

```
{
  "asset": {
    "version": "2.0"
  },
  "nodes": [
    {
      "name": "\u0043\u0043\u0043"
    },
    {
      "name": "\u007a\u0062\u009a\u0064"
    }
  ]
}
```

4. Property names (keys) within JSON objects **SHOULD** be unique. glTF client implementations **SHOULD** override lexically preceding values for the same key.
5. Some of glTF properties are defined as integers in the schema. Such values **MAY** be stored as decimals with a zero fractional part or by using exponent notation. Regardless of encoding, such properties **MUST NOT** contain any non-zero fractional value.

Example

100, 100.0, and 1e2 represent the same value. See [RFC 8259](#), Section 6 for more details.

6. Non-integer numbers **SHOULD** be written in a way that preserves original values when these numbers are read back, i.e., they **SHOULD NOT** be altered by JSON serialization / deserialization roundtrip.



Implementation Note

This is typically achieved with algorithms like Grisu2 used by common JSON libraries.

2.8. URIs

glTF assets use [URIs](#) or [IRIs](#) to reference buffers and image resources. Assets **MAY** contain at least these two URI types:

- **Data URIs** that embed binary resources in the glTF JSON as defined by the [RFC 2397](#). The Data URI's **mediatype** field **MUST** match the encoded content.



Implementation Note

Base64 encoding used in Data URI increases the payload's byte length by 33%.

- **Relative paths** — **path-noscheme** or **ipath-noscheme** as defined by [RFC 3986](#), Section 4.2 or [RFC 3987](#), Section 2.2 — without scheme, authority, or parameters. Reserved characters (as defined by [RFC 3986](#), Section 2.2. and [RFC 3987](#), Section 2.2.) **MUST** be percent-encoded.

Paths with non-ASCII characters **MAY** be written as-is, with JSON string escaping, or with percent-encoding; all these options are valid. For example, the following three paths point to the same resource:

```
{
  "images": [
    {
      "uri": "grande_sphère.png"
    },
    {
      "uri": "grande_sph\u00E8re.png"
    },
    {
      "uri": "grande_sph%C3%A8re.png"
    }
  ]
}
```

Client implementations **MAY** optionally support additional URI components. For example [http://](#) or [file://](#) schemes, authorities, hostnames, absolute paths, and query or fragment parameters. Assets containing these additional URI components would be less portable.

*Implementation Note*

This allows the application to decide the best approach for delivery: if different assets share many of the same geometries, animations, or textures, separate files may be preferred to reduce the total amount of data requested. With separate files, applications may progressively load data and do not need to load data for parts of a model that are not visible. If an application cares more about single-file deployment, embedding data may be preferred even though it increases the overall size due to base64 encoding and does not support progressive or on-demand loading. Alternatively, an asset could use the GLB container to store JSON and binary data in one file without base64 encoding. See [GLB File Format Specification](#) for details.

URIs **SHOULD** undergo syntax-based normalization as defined by [RFC 3986](#), Section 6.2.2, [RFC 3987](#), Section 5.3.2, and applicable schema rules (e.g., [RFC 7230](#), Section 2.7.3 for HTTP) on export and/or import.

*Implementation Note*

While the specification does not explicitly disallow non-normalized URIs, their use may be unsupported or lead to unwanted side-effects — such as security warnings or cache misses — on some platforms.

Chapter 3. Concepts

3.1. General

The figure below shows relations between top-level arrays in a glTF asset. See the [Properties Reference](#).

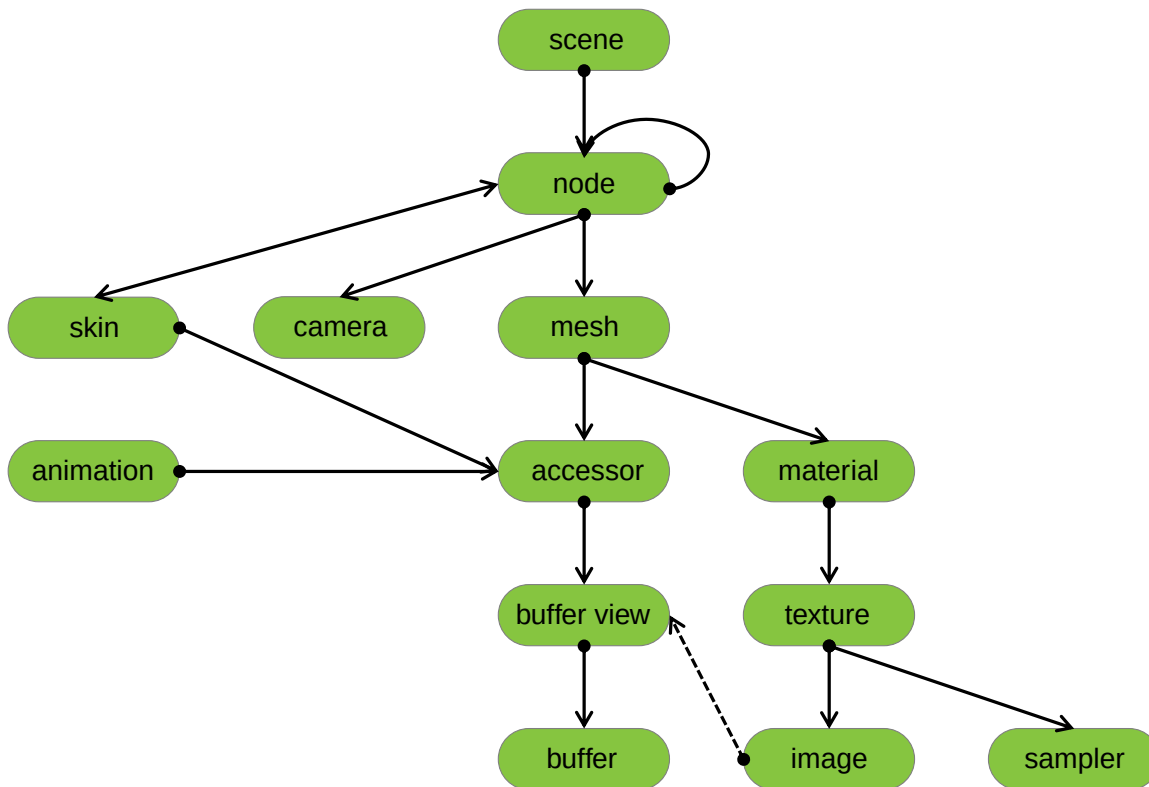


Figure 1. glTF Object Hierarchy

3.2. Asset

Each glTF asset **MUST** have an **asset** property. The **asset** object **MUST** contain a **version** property that specifies the target glTF version of the asset. Additionally, an optional **minVersion** property **MAY** be used to specify the minimum glTF version support required to load the asset. The **minVersion** property allows asset creators to specify a minimum version that a client implementation **MUST** support in order to load the asset. This is very similar to the **extensionsRequired** concept described in [Section 3.12](#), where an asset **SHOULD NOT** be loaded if the client does not support the specified extension. Additional metadata **MAY** be stored in optional properties such as **generator** or **copyright**. For example,

```
{
  "asset": {
    "version": "2.0",
    "generator": "collada2gltf@f356b99aef8868f74877c7ca545f2cd206b9d3b7",
    "copyright": "2017 (c) Khronos Group"
  }
}
```

Implementation Note



Client implementations should first check whether a `minVersion` property is specified and ensure both major and minor versions can be supported. If no `minVersion` is specified, then clients should check the `version` property and ensure the major version is supported. Clients that load `GLB format` should also check for the `minVersion` and `version` properties in the JSON chunk as the version specified in the GLB header only refers to the GLB container version.

3.3. Indices and Names

Entities of a glTF asset are referenced by their indices in corresponding arrays, e.g., a `bufferView` refers to a `buffer` by specifying the buffer's index in `buffers` array. For example:

```
{
  "buffers": [
    {
      "byteLength": 1024,
      "uri": "path-to.bin"
    }
  ],
  "bufferViews": [
    {
      "buffer": 0,
      "byteLength": 512,
      "byteOffset": 0
    }
  ]
}
```

In this example, `buffers` and `bufferViews` arrays have only one element each. The `bufferView` refers to the buffer using the buffer's index: `"buffer": 0`.

Indices **MUST** be non-negative integer numbers. Indices **MUST** always point to existing elements.

Whereas indices are used for internal glTF references, optional *names* are used for application-specific uses such as display. Any top-level glTF object **MAY** have a `name` string property for this purpose. These property values are not guaranteed to be unique as they are intended to contain values created when the asset was authored.

For property names, glTF usually uses camel case, *likeThis*.

3.4. Coordinate System and Units

glTF uses a right-handed coordinate system. glTF defines +Y as up, +Z as forward, and -X as right; the front of a glTF asset faces +Z.

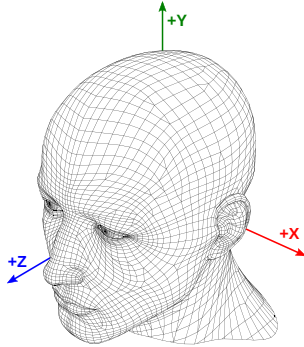


Figure 2. glTF Coordinate System Orientation

The units for all linear distances are meters.

All angles are in radians.

Positive rotation is counterclockwise.

Red, Green, and Blue primary colors use [Recommendation ITU-R BT.709](#) chromaticity coordinates.



Implementation Note

Chromaticity coordinates define the interpretation of each primary color channel of the color model. In the context of a typical display, color primaries describe the color of the red, green, and blue phosphors or filters. Unless a wide color gamut output is explicitly used, client implementations usually do not need to convert colors. Future specification versions or extensions may allow other color primaries (such as P3).

3.5. Scenes

3.5.1. Overview

glTF 2.0 assets **MAY** contain zero or more *scenes*, the set of visual objects to render. Scenes are defined in a **scenes** array. All nodes listed in **scene.nodes** array **MUST** be root nodes, i.e., they **MUST NOT** be listed in a **node.children** array of any node. The same root node **MAY** appear in multiple scenes.

An additional root-level property, **scene** (note singular), identifies which of the scenes in the array **SHOULD** be displayed at load time. When **scene** is undefined, client implementations **MAY** delay rendering until a particular scene is requested.

A glTF asset that does not contain any scenes **SHOULD** be treated as a library of individual entities

such as materials or meshes.

The following example defines a glTF asset with a single scene that contains a single node.

```
{
  "nodes": [
    {
      "name": "singleNode"
    }
  ],
  "scenes": [
    {
      "name": "singleScene",
      "nodes": [
        0
      ]
    }
  ],
  "scene": 0
}
```

3.5.2. Nodes and Hierarchy

glTF assets **MAY** define *nodes*, that is, the objects comprising the scene to render.

Nodes **MAY** have transform properties, as described later.

Nodes are organized in a parent-child hierarchy known informally as the *node hierarchy*. A node is called a *root node* when it doesn't have a parent.

The node hierarchy **MUST** be a set of disjoint strict trees. That is node hierarchy **MUST NOT** contain cycles and each node **MUST** have zero or one parent node.

The node hierarchy is defined using a node's *children* property, as in the following example:

```

{
  "nodes": [
    {
      "name": "Car",
      "children": [1, 2, 3, 4]
    },
    {
      "name": "wheel_1"
    },
    {
      "name": "wheel_2"
    },
    {
      "name": "wheel_3"
    },
    {
      "name": "wheel_4"
    }
  ]
}

```

The node named **Car** has four children. Each of those nodes could in turn have its own children, creating a hierarchy of nodes.

3.5.3. Transformations

Any node **MAY** define a local space transform either by supplying a **matrix** property, or any of **translation**, **rotation**, and **scale** properties (also known as *TRS properties*). **translation** and **scale** are 3D vectors in the local coordinate system. **rotation** is a unit quaternion value, XYZW, in the local coordinate system, where W is the scalar.

When **matrix** is defined, it **MUST** be decomposable to TRS properties.



Implementation Note

Transformation matrices cannot skew or shear.

When a node is targeted for animation (referenced by an **animation.channel.target**), only TRS properties **MAY** be present; **matrix** **MUST NOT** be present.

To compose the local transformation matrix, TRS properties **MUST** be converted to matrices and postmultiplied in the **T * R * S** order; first the scale is applied to the vertices, then the rotation, and then the translation.



Implementation Note

Non-invertible transforms (e.g., scaling one axis to zero) could lead to lighting and/or visibility artifacts.

*Implementation Note*

When the scale is zero on all three axes (by node transform or by animated scale), implementations are free to optimize away rendering of the node's mesh, and all of the node's children's meshes. This provides a mechanism to animate visibility. Skinned meshes must not use this optimization unless all of the joints in the skin are scaled to zero simultaneously.

The global transformation matrix of a node is the product of the global transformation matrix of its parent node and its own local transformation matrix. When the node has no parent node, its global transformation matrix is identical to its local transformation matrix.

In the example below, a node named **Box** defines non-default rotation and translation.

```
{
  "nodes": [
    {
      "name": "Box",
      "rotation": [
        0,
        0,
        0,
        1
      ],
      "scale": [
        1,
        1,
        1
      ],
      "translation": [
        -17.7082,
        -11.4156,
        2.0922
      ]
    }
  ]
}
```

The next example defines the transform for a node with attached camera using the **matrix** property rather than using the individual TRS values:

```

{
  "nodes": [
    {
      "name": "node-camera",
      "camera": 1,
      "matrix": [
        -0.99975,
        -0.00679829,
        0.0213218,
        0,
        0.00167596,
        0.927325,
        0.374254,
        0,
        -0.0223165,
        0.374196,
        -0.927081,
        0,
        -0.0115543,
        0.194711,
        -0.478297,
        1
      ]
    }
  ]
}

```

3.6. Binary Data Storage

3.6.1. Buffers and Buffer Views

3.6.1.1. Overview

A **buffer** is arbitrary data stored as a binary blob. The buffer **MAY** contain any combination of geometry, animation, skins, and images.

Binary blobs allow efficient creation of GPU buffers and textures since they require no additional parsing, except perhaps decompression.

glTF assets **MAY** have any number of buffer resources. Buffers are defined in the asset's **buffers** array.

While there's no hard upper limit on buffer's size, glTF assets **SHOULD NOT** use buffers bigger than 2^{53} bytes because some JSON parsers may be unable to parse their **byteLength** correctly. Buffers stored as **GLB** binary chunk have an implicit limit of $2^{32}-1$ bytes.

All buffer data defined in this specification (i.e., geometry attributes, geometry indices, sparse accessor data, animation inputs and outputs, inverse bind matrices) **MUST** use little endian byte order.

The following example defines a buffer. The `byteLength` property specifies the size of the buffer file. The `uri` property is the URI to the buffer data.

```
{
  "buffers": [
    {
      "byteLength": 102040,
      "uri": "duck.bin"
    }
  ]
}
```

The byte length of the referenced resource **MUST** be greater than or equal to the `buffer.byteLength` property.

Buffer data **MAY** alternatively be embedded in the glTF file via `data:` URI with base64 encoding. When `data:` URI is used for buffer storage, its `mediatype` field **MUST** be set to `application/octet-stream` or `application/glTF-buffer`.

A *buffer view* represents a contiguous segment of data in a buffer, defined by a byte offset into the buffer specified in the `byteOffset` property and a total byte length specified by the `byteLength` property of the buffer view.

Buffer views used for images, vertex indices, vertex attributes, or inverse bind matrices **MUST** contain only one kind of data, i.e., the same buffer view **MUST NOT** be used both for vertex indices and vertex attributes.

When a buffer view is used by vertex indices or attribute accessors it **SHOULD** specify `bufferView.target` with a value of *element array buffer* or *array buffer* respectively.



Implementation Note

This allows client implementations to early designate each buffer view to a proper processing step, e.g, buffer views with vertex indices and attributes would be copied to the appropriate GPU buffers, while buffer views with image data would be passed to format-specific image decoders.

The `bufferView.target` value uses integer enums defined in the [Properties Reference](#).

The following example defines two buffer views: the first holds the indices for an indexed triangle set, and the second holds the vertex data for the triangle set.

```

{
  "bufferViews": [
    {
      "buffer": 0,
      "byteLength": 25272,
      "byteOffset": 0,
      "target": 34963
    },
    {
      "buffer": 0,
      "byteLength": 76768,
      "byteOffset": 25272,
      "byteStride": 32,
      "target": 34962
    }
  ]
}

```

When a buffer view is used for vertex attribute data, it **MAY** have a **byteStride** property. This property defines the stride in bytes between each vertex. Buffer views with other types of data **MUST NOT** not define **byteStride** (unless such layout is explicitly enabled by an extension).

Buffers and buffer views do not contain type information. They simply define the raw data for retrieval from the file. Objects within the glTF asset (meshes, skins, animations) access buffers or buffer views via **accessors**.

3.6.1.2. GLB-stored Buffer

The glTF asset **MAY** use the GLB file container to pack glTF JSON and one glTF buffer into one file. Data for such a buffer is provided via the GLB-stored **BIN** chunk.

A buffer with data provided by the GLB-stored **BIN** chunk, **MUST** be the first element of **buffers** array and it **MUST** have its **buffer.uri** property undefined. When such a buffer exists, a **BIN** chunk **MUST** be present.

Any glTF buffer with undefined **buffer.uri** property that is not the first element of **buffers** array does not refer to the GLB-stored **BIN** chunk, and the behavior of such buffers is left undefined to accommodate future extensions and specification versions.

The byte length of the **BIN** chunk **MAY** be up to 3 bytes bigger than JSON-defined **buffer.byteLength** value to satisfy GLB padding requirements.



Implementation Note

Not requiring strict equality of chunk's and buffer's lengths slightly simplifies glTF to GLB conversion: **buffer.byteLength** does not need to be updated after applying GLB padding.

In the following example, the first buffer object refers to GLB-stored data, while the second points to external resource:

```

{
  "buffers": [
    {
      "byteLength": 35884
    },
    {
      "byteLength": 504,
      "uri": "external.bin"
    }
  ]
}

```

See [GLB File Format Specification](#) for details on GLB File Format.

3.6.2. Accessors

3.6.2.1. Overview

All binary data for meshes, skins, and animations is stored in buffers and retrieved via accessors.

An *accessor* defines a method for retrieving data as typed arrays from within a buffer view. The accessor specifies a component type (e.g., *float*) and a data type (e.g., *VEC3* for 3D vectors), which when combined define the complete data type for each data element. The number of elements is specified using the *count* property. Elements could be, e.g., vertex indices, vertex attributes, animation keyframes, etc.

The *byteOffset* property specifies the location of the first data element within the referenced buffer view. If the accessor is used for vertex attributes (i.e., it is referenced by a mesh primitive or its morph targets), the locations of the subsequent data elements are controlled by the *bufferView.byteStride* property. If the accessor is used for any other kind of data (vertex indices, animation keyframes, etc.), its data elements are tightly packed.

All accessors are stored in the asset's *accessors* array.

The following example shows two accessors, the first is a scalar accessor for retrieving a primitive's indices, and the second is a 3-float-component vector accessor for retrieving the primitive's position data.

```

{
  "accessors": [
    {
      "bufferView": 0,
      "byteOffset": 0,
      "componentType": 5123,
      "count": 12636,
      "max": [
        4212
      ],
      "min": [
        0
      ],
      "type": "SCALAR"
    },
    {
      "bufferView": 1,
      "byteOffset": 0,
      "componentType": 5126,
      "count": 2399,
      "max": [
        0.961799,
        1.6397,
        0.539252
      ],
      "min": [
        -0.692985,
        0.0992937,
        -0.613282
      ],
      "type": "VEC3"
    }
  ]
}

```

3.6.2.2. Accessor Data Types

componentType	Data Type	Signed	Bits
5120	<i>signed byte</i>	Signed, two's complement	8
5121	<i>unsigned byte</i>	Unsigned	8
5122	<i>signed short</i>	Signed, two's complement	16
5123	<i>unsigned short</i>	Unsigned	16
5125	<i>unsigned int</i>	Unsigned	32
5126	<i>float</i>	Signed	32



Signed 32-bit integer components are not supported.

Floating-point data **MUST** use [IEEE-754](#) single precision format.

Values of **NaN**, **+Infinity**, and **-Infinity** **MUST NOT** be present.

type	Number of components
"SCALAR"	1
"VEC2"	2
"VEC3"	3
"VEC4"	4
"MAT2"	4
"MAT3"	9
"MAT4"	16

Element size, in bytes, is (size in bytes of the 'componentType') * (number of components defined by 'type').

For example:

```
{
  "accessors": [
    {
      "bufferView": 1,
      "byteOffset": 7032,
      "componentType": 5126,
      "count": 585,
      "type": "VEC3"
    }
  ]
}
```

In this accessor, the **componentType** is 5126 (*float*), so each component is four bytes. The **type** is "VEC3", so there are three components. The size of each element is 12 bytes (4 * 3). Thus, the accessor takes 7020 bytes ([7032 ... 14051] inclusive range of the buffer view).

3.6.2.3. Sparse Accessors

Sparse encoding of arrays is often more memory-efficient than dense encoding when describing incremental changes with respect to a reference array. This is often the case when encoding morph targets (it is, in general, more efficient to describe a few displaced vertices in a morph target than transmitting all morph target vertices).

Similar to a standard accessor, a sparse accessor initializes an array of typed elements from data stored in a **bufferView**. When **accessor.bufferView** is undefined, the sparse accessor is initialized as an array of zeros of size (size of the accessor element) * (**accessor.count**) bytes.

On top of that, a sparse accessor includes a **sparse** JSON object describing the elements that are different from their initialization values. The **sparse** object contains the following **REQUIRED** properties:

- **count**: number of displaced elements. This number **MUST NOT** be greater than the number of the base accessor elements.
- **indices**: object describing the location and the component type of indices of values to be replaced. The indices **MUST** form a strictly increasing sequence. The indices **MUST NOT** be greater than or equal to the number of the base accessor elements.
- **values**: object describing the location of displaced elements corresponding to the indices referred from the **indices**.

The following example shows an example of a sparse accessor with 10 elements that are different from the initialization array.

```
{
  "accessors": [
    {
      "bufferView": 0,
      "byteOffset": 0,
      "componentType": 5123,
      "count": 12636,
      "type": "VEC3",
      "sparse": {
        "count": 10,
        "indices": {
          "bufferView": 1,
          "byteOffset": 0,
          "componentType": 5123
        },
        "values": {
          "bufferView": 2,
          "byteOffset": 0
        }
      }
    }
  ]
}
```

3.6.2.4. Data Alignment

The offset of an **accessor** into a **bufferView** (i.e., **accessor.byteOffset**) and the offset of an **accessor** into a **buffer** (i.e., **accessor.byteOffset + bufferView.byteOffset**) **MUST** be a multiple of the size of the accessor's component type.

When **byteStride** of the referenced **bufferView** is not defined, it means that accessor elements are tightly packed, i.e., effective stride equals the size of the element. When **byteStride** is defined, it **MUST** be a multiple of the size of the accessor's component type.

When two or more vertex attribute accessors use the same `bufferView`, its `byteStride` **MUST** be defined.

Each accessor **MUST** fit its `bufferView`, i.e.,

$$\text{accessor.byteOffset} + \text{EFFECTIVE_BYTE_STRIDE} * (\text{accessor.count} - 1) + \text{SIZE_OF_COMPONENT} * \text{NUMBER_OF_COMPONENTS}$$

MUST be less than or equal to `bufferView.length`.

For performance and compatibility reasons, each element of a vertex attribute **MUST** be aligned to 4-byte boundaries inside a `bufferView` (i.e., `accessor.byteOffset` and `bufferView.byteStride` **MUST** be multiples of 4).

Accessors of matrix type have data stored in column-major order; start of each column **MUST** be aligned to 4-byte boundaries. Specifically, when `ROWS * SIZE_OF_COMPONENT` (where `ROWS` is the number of rows of the matrix) is not a multiple of 4, then $(\text{ROWS} * \text{SIZE_OF_COMPONENT}) \% 4$ padding bytes **MUST** be inserted at the end of each column.

Only the following three accessor configurations require padding.

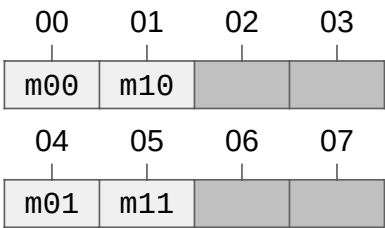


Figure 3. Matrix 2x2, 1-byte Components

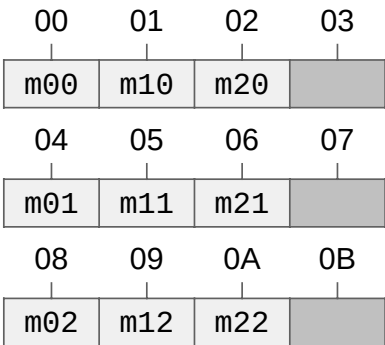


Figure 4. Matrix 3x3, 1-byte Components

00	01	02	03	04	05	06	07
m00	m00	m10	m10	m20	m20		
08	09	0A	0B	0C	0D	0E	0F
m01	m01	m11	m11	m21	m21		
10	11	12	13	14	15	16	17
m02	m02	m12	m12	m22	m22		

Figure 5. Matrix 3x3, 2-byte Components

Alignment requirements apply only to the start of each column, so trailing bytes **MAY** be omitted if there's no further data.



Implementation Note

Alignment requirements allow client implementations to more efficiently process binary buffers because creating aligned data views usually does not require extra copying.

Consider the following example:

```
{
  "bufferViews": [
    {
      "buffer": 0,
      "byteLength": 17136,
      "byteOffset": 620
    }
  ],
  "accessors": [
    {
      "bufferView": 0,
      "byteOffset": 4608,
      "componentType": 5123,
      "count": 42,
      "type": "VEC2"
    }
  ]
}
```

In this example, the accessor describes tightly-packed two-component unsigned short values.

The corresponding segment of the underlying buffer would start from byte 5228

```
start = accessor.byteOffset + accessor.bufferView.byteOffset
```

and continue until byte 5396 exclusive

```
end = 2 * 2 * accessor.count + start
```

The unsigned short view for the resulting buffer range could be created without copying: 84 scalar values starting from byte offset 5228.

When accessor values are not tightly-packed (i.e., `bufferView.byteStride` is greater than element's byte length), iteration over the created data view would need to take interleaved values into account (i.e., skip them).

3.6.2.5. Accessors Bounds

`accessor.min` and `accessor.max` properties are arrays that contain per-component minimum and maximum values, respectively. The length of these arrays **MUST** be equal to the number of accessor's components.

Values stored in glTF JSON **MUST** match actual minimum and maximum binary values stored in buffers. The `accessor.normalized` flag has no effect on these properties.

A sparse accessor `min` and `max` properties correspond, respectively, to the minimum and maximum component values once the sparse substitution is applied.

When neither `sparse` nor `bufferView` is defined, `min` and `max` properties **MAY** have any values. This is intended for use cases when binary data is supplied by external means (e.g., via extensions).

For floating-point components, JSON-stored minimum and maximum values represent single precision floats and **SHOULD** be rounded to single precision before usage to avoid any potential boundary mismatches.



ECMAScript Implementation Note

`Math.fround` function could be used to achieve that.

Animation input and vertex position attribute accessors **MUST** have `accessor.min` and `accessor.max` defined. For all other accessors, these properties are optional.

3.7. Geometry

3.7.1. Overview

Any node **MAY** contain one mesh, defined in its `mesh` property. The mesh **MAY** be skinned using information provided in a referenced `skin` object. The mesh **MAY** have morph targets.

3.7.2. Meshes

3.7.2.1. Overview

Meshes are defined as arrays of *primitives*. Primitives correspond to the data required for GPU draw calls. Primitives specify one or more `attributes`, corresponding to the vertex attributes used in the draw calls. Indexed primitives also define an `indices` property. Attributes and indices are

defined as references to accessors containing corresponding data. Each primitive **MAY** also specify a **material** and a **mode** that corresponds to the GPU topology type (e.g., triangle set).



Implementation Note

Splitting one mesh into several **primitives** can be useful to limit the number of indices per draw call or to assign different materials to different parts of the mesh.

If **material** is undefined, then a **default material** **MUST** be used.

The following example defines a mesh containing one indexed triangle primitive:

```
{
  "meshes": [
    {
      "primitives": [
        {
          "attributes": {
            "NORMAL": 23,
            "POSITION": 22,
            "TANGENT": 24,
            "TEXCOORD_0": 25
          },
          "indices": 21,
          "material": 3,
          "mode": 4
        }
      ]
    }
  ]
}
```

Each attribute is defined as a property of the **attributes** object. The name of the property corresponds to an enumerated value identifying the vertex attribute, such as **POSITION**. The value of the property is the index of an accessor that contains the data.

The specification defines the following attribute semantics: **POSITION**, **NORMAL**, **TANGENT**, **TEXCOORD_n**, **COLOR_n**, **JOINTS_n**, and **WEIGHTS_n**.

Application-specific attribute semantics **MUST** start with an underscore, e.g., **_TEMPERATURE**. Application-specific attribute semantics **MUST NOT** use *unsigned int* component type.

Valid accessor type and component type for each attribute semantic property are defined below.

Name	Accessor Type(s)	Component Type(s)	Description
POSITION	VEC3	<i>float</i>	Unitless XYZ vertex positions
NORMAL	VEC3	<i>float</i>	Normalized XYZ vertex normals

Name	Accessor Type(s)	Component Type(s)	Description
TANGENT	VEC4	<i>float</i>	XYZW vertex tangents where the XYZ portion is normalized, and the W component is a sign value (-1 or +1) indicating handedness of the tangent basis
TEXCOORD_n	VEC2	<i>float</i> <i>unsigned byte</i> normalized <i>unsigned short</i> normalized	ST texture coordinates
COLOR_n	VEC3 VEC4	<i>float</i> <i>unsigned byte</i> normalized <i>unsigned short</i> normalized	RGB or RGBA vertex color linear multiplier
JOINTS_n	VEC4	<i>unsigned byte</i> <i>unsigned short</i>	See Skinned Mesh Attributes
WEIGHTS_n	VEC4	<i>float</i> <i>unsigned byte</i> normalized <i>unsigned short</i> normalized	See Skinned Mesh Attributes

POSITION accessor **MUST** have its **min** and **max** properties defined.

The W component of each **TANGENT** accessor element **MUST** be set to **1.0** or **-1.0**.

When a **COLOR_n** attribute uses an accessor of "**VEC3**" type, its alpha component **MUST** be assumed to have a value of **1.0**.

All components of each **COLOR_0** accessor element **MUST** be clamped to **[0.0, 1.0]** range.

TEXCOORD_n, **COLOR_n**, **JOINTS_n**, and **WEIGHTS_n** attribute semantic property names **MUST** be of the form **[semantic]_[set_index]**, e.g., **TEXCOORD_0**, **TEXCOORD_1**, **COLOR_0**. All indices for indexed attribute semantics **MUST** start with 0 and be consecutive positive integers: **TEXCOORD_0**, **TEXCOORD_1**, etc. Indices **MUST NOT** use leading zeroes to pad the number of digits (e.g., **TEXCOORD_01** is not allowed).

Client implementations **SHOULD** support at least two texture coordinate sets, one vertex color, and one joints/weights set.

All attribute accessors for a given primitive **MUST** have the same **count**. When **indices** property is not defined, attribute accessors' **count** indicates the number of vertices to render; when **indices** property is defined, it indicates the upper (exclusive) bound on the index values in the **indices** accessor, i.e., all index values **MUST** be less than attribute accessors' **count**.

indices accessor **MUST NOT** contain the maximum possible value for the component type used (i.e., 255 for unsigned bytes, 65535 for unsigned shorts, 4294967295 for unsigned ints).



Implementation Note

The maximum values trigger primitive restart in some graphics APIs and would require client implementations to rebuild the index buffer.

When **indices** property is not defined, the number of vertex indices to render is defined by **count** of attribute accessors (with the implied values from range **[0..count)**); when **indices** property is defined, the number of vertex indices to render is defined by **count** of accessor referred to by **indices**. In either case, the number of vertex indices **MUST** be valid for the topology type used:

- For *points*, it **MUST** be non-zero.
- For *line loops* and *line strips*, it **MUST** be 2 or greater.
- For *triangle strips* and *triangle fans*, it **MUST** be 3 or greater.
- For *lines*, it **MUST** be divisible by 2 and non-zero.
- For *triangles*, it **MUST** be divisible by 3 and non-zero

Topology types are defined as follows.

- **Points**

Each vertex defines a single point primitive, according to the equation:

$$p_i = \{v_i\}$$

- **Line Strips**

One line primitive is defined by each vertex and the following vertex, according to the equation:

$$p_i = \{v_i, v_{i+1}\}$$

- **Line Loops**

Loops are the same as line strips except that a final segment is added from the final specified vertex to the first vertex.

- **Lines**

Each consecutive pair of vertices defines a single line primitive, according to the equation:

$$p_i = \{v_{2i}, v_{2i+1}\}$$

- **Triangles**

Each consecutive set of three vertices defines a single triangle primitive, according to the equation:

$$p_i = \{v_{3i}, v_{3i+1}, v_{3i+2}\}$$

- **Triangle Strips**

One triangle primitive is defined by each vertex and the two vertices that follow it, according to the equation:

the equation:

$$p_i = \{v_i, v_{i+(1+i\%2)}, v_{i+(2-i\%2)}\}$$

• Triangle Fans

Triangle primitives are defined around a shared common vertex, according to the equation:

$$p_i = \{v_{i+1}, v_{i+2}, v_0\}$$

Mesh geometry **SHOULD NOT** contain degenerate lines or triangles, i.e., lines or triangles that use the same vertex more than once per topology primitive.

When positions are not specified, client implementations **SHOULD** skip primitive's rendering unless its positions are provided by other means (e.g., by an extension). This applies to both indexed and non-indexed geometry.

When tangents are not specified, client implementations **SHOULD** calculate tangents using default [MikkTSpace](#) algorithms with the specified vertex positions, normals, and texture coordinates associated with the normal texture.

When normals are not specified, client implementations **MUST** calculate flat normals and the provided tangents (if present) **MUST** be ignored.

Vertices of the same triangle **SHOULD** have the same `tangent.w` value. When vertices of the same triangle have different `tangent.w` values, its tangent space is considered undefined.

The bitangent vectors **MUST** be computed by taking the cross product of the normal and tangent XYZ vectors and multiplying it against the W component of the tangent: `bitangent = cross(normal.xyz, tangent.xyz) * tangent.w`.

Extensions **MAY** add additional attribute names, accessor types, and/or component types.

3.7.2.2. Morph Targets

Morph targets are defined by extending the Mesh concept.

A morph target is a morphable Mesh where the primitives' attributes are obtained by adding the original attributes to a weighted sum of the target's attributes.

For instance, the morph target vertices `POSITION` for the primitive at index `i` are computed in this way:

```
primitives[i].attributes.POSITION +
weights[0] * primitives[i].targets[0].POSITION +
weights[1] * primitives[i].targets[1].POSITION +
weights[2] * primitives[i].targets[2].POSITION + ...
```

Morph targets are specified via the **targets** property defined in the Mesh **primitives**. Each target in the **targets** array is a plain JSON object mapping a primitive attribute to an accessor containing morph target displacement data (deltas).

For each morph target attribute, an original attribute **MUST** be present in the mesh primitive.

Attributes present in the base mesh primitive but not included in a given morph target **MUST** retain their original values for the morph target.



Implementation Note

This allows skipping zero-filled accessors and implies that different morph targets may contain different sets of attributes.

Client implementations **SHOULD** support at least three attributes—**POSITION**, **NORMAL**, and **TANGENT**—for morphing. Client implementations **MAY** optionally support morphed **TEXCOORD_n** and/or **COLOR_n** attributes.

If morph targets contain application-specific semantics, their names **MUST** be prefixed with an underscore (e.g., **_TEMPERATURE**) like the associated attribute semantics.

All primitives **MUST** have the same number of morph targets in the same order.

Accessor type and component type for each morphed attribute semantic property **MUST** follow the table below. Note that the W component for handedness is omitted when targeting **TANGENT** data since handedness cannot be displaced.

Name	Accessor Type(s)	Component Type(s)	Description
POSITION	VEC3	<i>float</i>	XYZ vertex position displacements
NORMAL	VEC3	<i>float</i>	XYZ vertex normal displacements
TANGENT	VEC3	<i>float</i>	XYZ vertex tangent displacements
TEXCOORD_n	VEC2	<i>float</i> <i>signed byte</i> normalized <i>signed short</i> normalized <i>unsigned byte</i> normalized <i>unsigned short</i> normalized	ST texture coordinate displacements

Name	Accessor Type(s)	Component Type(s)	Description
COLOR_n	VEC3 VEC4	<i>float</i> <i>signed byte normalized</i> <i>signed short normalized</i> <i>unsigned byte normalized</i> <i>unsigned short normalized</i>	RGB or RGBA color deltas

POSITION accessor **MUST** have its **min** and **max** properties defined.

Displacements for **POSITION**, **NORMAL**, and **TANGENT** attributes **MUST** be applied before any transformation matrices affecting the mesh vertices such as skinning or node transforms.

When the base mesh primitive does not specify tangents, client implementations **SHOULD** calculate tangents for each morph target using default **MikkTSpace** algorithms with the updated vertex positions, normals, and texture coordinates associated with the normal texture.

When the base mesh primitive does not specify normals, client implementations **MUST** calculate flat normals for each morph target; the provided tangents and their displacements (if present) **MUST** be ignored.

When **COLOR_n** deltas use an accessor of "**VEC3**" type, their alpha components **MUST** be assumed to have a value of **0.0**.

After applying color deltas, all components of each **COLOR_0** morphed accessor element **MUST** be clamped to **[0.0, 1.0]** range.

All morph target accessors **MUST** have the same **count** as the accessors of the original primitive.

A mesh with morph targets **MAY** also define an optional **mesh.weights** property that stores the default targets' weights. These weights **MUST** be used when **node.weights** is undefined. When **mesh.weights** is undefined, the default targets' weights are zeros.

The following example extends the Mesh defined in the previous example to a morphable one by adding two morph targets:

```

{
  "primitives": [
    {
      "attributes": {
        "NORMAL": 23,
        "POSITION": 22,
        "TANGENT": 24,
        "TEXCOORD_0": 25
      },
      "indices": 21,
      "material": 3,
      "targets": [
        {
          "NORMAL": 33,
          "POSITION": 32,
          "TANGENT": 34
        },
        {
          "NORMAL": 43,
          "POSITION": 42,
          "TANGENT": 44
        }
      ]
    }
  ],
  "weights": [0, 0.5]
}

```

The number of morph targets is not limited. Client implementations **SHOULD** support at least eight morphed attributes. This means that they **SHOULD** support eight morph targets when each morph target has one attribute, four morph targets where each morph target has two attributes, or two morph targets where each morph target has three or four attributes.

For assets that contain a higher number of morphed attributes, client implementations **MAY** choose to only use the eight attributes of the morph targets with the highest weights.



Implementation Note

A significant number of authoring and client implementations associate names with morph targets. While the glTF 2.0 specification currently does not provide a way to specify names, most tools use an array of strings, `mesh.extras.targetNames`, for this purpose. The `targetNames` array and all primitive `targets` arrays must have the same length.

3.7.3. Skins

3.7.3.1. Overview

glTF 2.0 meshes support Linear Blend Skinning via skin objects, joint hierarchies, and designated

vertex attributes.

Skins are stored in the **skins** array of the asset. Each skin is defined by a **REQUIRED joints** property that lists the indices of nodes used as joints to pose the skin and an **OPTIONAL inverseBindMatrices** property that points to an accessor with inverse bind matrices data used to bring coordinates being skinned into the same space as each joint.

The order of joints is defined by the **skin.joints** array and it **MUST** match the order of **inverseBindMatrices** accessor elements (when the latter is present). The **skeleton** property (if present) points to the node that is the common root of a joints hierarchy or to a direct or indirect parent node of the common root.



Implementation Note

Although the **skeleton** property is not needed for computing skinning transforms, it may be used to provide a specific “pivot point” for the skinned geometry.

An accessor referenced by **inverseBindMatrices** **MUST** have floating-point components of “MAT4” type. The number of elements of the accessor referenced by **inverseBindMatrices** **MUST** be greater than or equal to the number of **joints** elements. The fourth row of each matrix **MUST** be set to **[0.0, 0.0, 0.0, 1.0]**.



Implementation Note

The matrix defining how to pose the skin’s geometry for use with the joints (also known as “Bind Shape Matrix”) should be premultiplied to mesh data or to Inverse Bind Matrices.

3.7.3.2. Joint Hierarchy

The joint hierarchy used for controlling skinned mesh pose is simply the node hierarchy, with each node designated as a *joint* by a reference from the **skin.joints** array. Each skin’s joints **MUST** have a common parent node (direct or indirect) called *common root*, which may or may not be a joint node itself. When a skin is referenced by a node within a scene, the common root **MUST** belong to the same scene.



Implementation Note

A node object does not specify whether it is a joint. Client implementations may need to traverse the **skins** array first, marking each joint node.

A joint node **MAY** have other nodes attached to it, even a complete node sub graph with meshes.



Implementation Note

It’s common to have an entire geometry attached to a joint node without having it being skinned (e.g., a sword attached to a hand). Note that the node transform is the local transform of the node relative to the joint, like any other node in the glTF node hierarchy as described in the [Transformations](#) section.

Only the joint transforms are applied to the skinned mesh; the transform of the skinned mesh node