



# 3D TILES SPECIFICATION

---

COMMUNITY STANDARD

APPROVED

**Version:** 1.1

**Submission Date:** 2022-10-07

**Approval Date:** 2022-12-17

**Publication Date:** 2023-01-12

**Editor:** Patrick Cozzi, Sean Lilley

**Notice:** This document is an OGC Member approved international standard. This document is available on a royalty free, non-discriminatory basis. Recipients of this document are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

## License Agreement

Use of this document is subject to the license agreement at <https://www.ogc.org/license>

## Copyright notice

Copyright © 2023 Open Geospatial Consortium  
To obtain additional rights of use, visit <https://www.ogc.org/legal>

## Note

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. The Open Geospatial Consortium shall not be held responsible for identifying any or all such patent rights.

Recipients of this document are requested to submit, with their comments, notification of any relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the standard set forth in this document, and to provide supporting documentation.

# CONTENTS

---

I.	ABSTRACT .....	xii
II.	KEYWORDS .....	xii
III.	PREFACE .....	xiii
IV.	SECURITY CONSIDERATIONS .....	xiv
V.	SUBMITTING ORGANIZATIONS .....	xv
VI.	SUBMITTERS .....	xv
VII.	FUTURE WORK .....	xv
1.	SCOPE .....	2
2.	CONFORMANCE .....	4
3.	NORMATIVE REFERENCES .....	6
4.	TERMS AND DEFINITIONS .....	8
5.	CONVENTIONS .....	15
6.	3D TILES FORMAT SPECIFICATION .....	17
6.1.	Introduction .....	17
6.2.	File Extensions and Media Types .....	19
6.3.	JSON encoding .....	19
6.4.	URLs .....	20
6.5.	Units .....	20
6.6.	Coordinate reference system (CRS) .....	20
6.7.	Concepts .....	21
6.8.	Tile format specifications .....	57
6.9.	Declarative styling specification .....	58
7.	TILE FORMATS .....	60
7.1.	Legacy Tile Formats .....	60
7.2.	gITF .....	60
7.3.	Batch Table .....	66
7.4.	Feature Table .....	71
7.5.	Batched 3D Model .....	73

7.6. Instanced 3D Model .....	78
7.7. Point Cloud .....	88
7.8. Composite .....	98
<b>8. IMPLICIT TILING .....</b>	<b>102</b>
8.1. Overview .....	102
8.2. Implicit Root Tile .....	103
8.3. Subdivision Scheme .....	104
8.4. Tile Coordinates .....	107
8.5. Template URLs .....	108
8.6. Subtrees .....	109
8.7. Subtree JSON Format .....	116
8.8. Subtree Binary Format .....	122
<b>9. 3D METADATA SPECIFICATION .....</b>	<b>125</b>
9.1. Overview .....	125
9.2. Concepts .....	125
9.3. Schemas .....	126
9.4. Storage Formats .....	136
<b>10. STYLING .....</b>	<b>146</b>
10.1. Overview .....	146
10.2. Concepts .....	147
10.3. Expressions .....	150
10.4. Point Cloud .....	170
10.5. File extension and media type .....	172
<b>ANNEX A (NORMATIVE) PROPERTIES REFERENCE .....</b>	<b>174</b>
A.1. Asset .....	174
A.2. Availability .....	175
A.3. Batched 3D Model Feature Table .....	177
A.4. Batch Table .....	179
A.5. Bounding Volume .....	182
A.6. Buffer .....	184
A.7. Buffer View .....	186
A.8. Class .....	188
A.9. Class Property .....	190
A.10. Content .....	197
A.11. Definitions .....	199
A.12. Enum .....	201
A.13. Enum value .....	203
A.14. Extension .....	205
A.15. Extras .....	205
A.16. Feature Table .....	206
A.17. Group Metadata .....	211
A.18. Instanced 3D Model Feature Table .....	213
A.19. Metadata Entity .....	218

A.20. Point Cloud Feature Table .....	220
A.21. Properties .....	226
A.22. Property Table .....	227
A.23. Property Table Property .....	229
A.24. Root Property .....	235
A.25. Schema .....	235
A.26. Statistics .....	238
A.27. Class Statistics .....	240
A.28. Property Statistics .....	241
A.29. Style .....	245
A.30. Boolean Expression .....	247
A.31. Color Expression .....	248
A.32. Conditions .....	248
A.33. Condition .....	249
A.34. Expression .....	249
A.35. Meta .....	250
A.36. Subtree .....	251
A.37. Subtrees .....	255
A.38. Template URI .....	256
A.39. Tile .....	257
A.40. Implicit tiling .....	263
A.41. Tileset .....	265
 ANNEX B (INFORMATIVE) JSON SCHEMA REFERENCE .....	271
B.1. JSON Schema for Asset .....	271
B.2. JSON Schema for Availability .....	271
B.3. JSON Schema for Batched 3D Model Feature Table .....	272
B.4. JSON Schema for Batch Table .....	273
B.5. JSON Schema for Bounding Volume .....	275
B.6. JSON Schema for Buffer .....	276
B.7. JSON Schema for Buffer View .....	276
B.8. JSON Schema for Class .....	277
B.9. JSON Schema for Class Property .....	277
B.10. JSON Schema for Content .....	281
B.11. JSON Schema for Definitions .....	281
B.12. JSON Schema for Enum .....	283
B.13. JSON Schema for Enum value .....	284
B.14. JSON Schema for Extension .....	285
B.15. JSON Schema for Extras .....	285
B.16. JSON Schema for Feature Table .....	285
B.17. JSON Schema for Group Metadata .....	288
B.18. JSON Schema for Instanced 3D Model Feature Table .....	288
B.19. JSON Schema for Metadata Entity .....	291
B.20. JSON Schema for Point Cloud Feature Table .....	291
B.21. JSON Schema for Properties .....	293
B.22. JSON Schema for Property Table .....	294
B.23. JSON Schema for Property Table Property .....	295

B.24. JSON Schema for Root Property .....	297
B.25. JSON Schema for Schema .....	297
B.26. JSON Schema for Statistics .....	298
B.27. JSON Schema for Class Statistics .....	299
B.28. JSON Schema for Property Statistics .....	299
B.29. JSON Schema for Style .....	301
B.30. JSON Schema for Boolean Expression .....	302
B.31. JSON Schema for Color Expression .....	302
B.32. JSON Schema for Conditions .....	302
B.33. JSON Schema for Condition .....	303
B.34. JSON Schema for Expression .....	303
B.35. JSON Schema for Meta .....	303
B.36. JSON Schema for Subtree .....	304
B.37. JSON Schema for Subtrees .....	305
B.38. JSON Schema for Template URI .....	306
B.39. JSON Schema for Tile .....	306
B.40. JSON Schema for Implicit tiling .....	308
B.41. JSON Schema for Tileset .....	309
 ANNEX C (INFORMATIVE) MIGRATION FROM LEGACY TILE FORMATS .....	313
C.1. Batched 3D Model (b3dm) .....	313
C.2. Instanced 3D Model (i3dm) .....	313
C.3. Point Cloud (pnts) .....	314
C.4. Composite (cmpt) .....	315
 ANNEX D (INFORMATIVE) AVAILABILITY INDEXING .....	317
D.1. Converting from Tile Coordinates to Morton Index .....	317
D.2. Availability Bitstream Lengths .....	318
D.3. Accessing Availability Bits .....	319
D.4. Global and Local Tile Coordinates .....	319
D.5. Finding Parent and Child Tiles .....	322
 ANNEX E (INFORMATIVE) 3D METADATA REFERENCE IMPLEMENTATION .....	325
E.1. Property Table Implementation .....	325
E.2. Schema Implementation .....	328
 ANNEX F (INFORMATIVE) 3D METADATA SEMANTIC REFERENCE .....	333
F.1. Overview .....	333
F.2. General .....	333
F.3. 3D Tiles .....	334
 ANNEX G (INFORMATIVE) CONTRIBUTORS .....	341
 ANNEX H (INFORMATIVE) REVISION HISTORY .....	344

# LIST OF TABLES

---

Table 1 – Legacy tile formats and common uses .....	17
Table 2 – A tile and a refined tile using replacement refinement .....	22
Table 3 – A tile and a refined tile using additive refinement .....	23
Table 4 – Different bounding volume types for a tile .....	23
Table 5 – Statistics for metadata .....	54
Table 6 – Legacy tile formats and common uses .....	58
Table 7 – Legacy tile formats and common uses .....	60
Table 8 – Different types of models based on glTF .....	61
Table 9 – Sizes of component types .....	69
Table 10 – Numbers of components for elements .....	69
Table 11 – Header fields for Batched 3D Models .....	75
Table 12 – Global semantics for Batched 3D models .....	76
Table 13 – Header fields for Instanced 3D Models .....	80
Table 14 – Semantics for instances of an Instanced 3D Model .....	81
Table 15 – Global semantics for Instanced 3D Models .....	82
Table 16 – Header fields for Point Clouds .....	89
Table 17 – Semantics for instances of a Point Cloud .....	90
Table 18 – Global semantics for Point Clouds .....	91
Table 19 – Header fields for Composite tiles .....	99
Table 20 – Common header fields for tile formats .....	100
Table 21 – Properties of implicit tiling object .....	104
Table 22 – Subdivision for bounding boxes .....	105
Table 23 – Subdivision for bounding regions .....	106
Table 24 – Subdivision rules for implicit tiling .....	106
Table 25 – Tile coordinates for bounding regions .....	107
Table 26 – Tile coordinates for bounding regions .....	108
Table 27 – Header fields for subtree files .....	122
Names and values of an example enum .....	128
Properties and types of a building .....	130
Properties and types of a tree .....	130
Table 28 – Types of metadata properties .....	131
Table 29 – Component types of metadata properties .....	132
Table 30 – Conversion of component types .....	133
Properties and types of a tree .....	136
Table 31 – Types for scalar metadata values .....	138
Table 32 – Offset types for strings and arrays .....	140
Names and values of an example enum .....	142
Properties of an example class .....	142
Table 33 – Vector type variables .....	160

Table A.1 – Asset Properties .....	174
Table A.2 – Availability Properties .....	175
Table A.3 – Batched 3D Model Feature Table Properties .....	177
Table A.4 – Batch Table Properties .....	179
Table A.5 – BinaryBodyReference Properties .....	180
Table A.6 – Bounding Volume Properties .....	182
Table A.7 – Buffer Properties .....	184
Table A.8 – Buffer View Properties .....	186
Table A.9 – Class Properties .....	188
Table A.10 – Class Property Properties .....	190
Table A.11 – Content Properties .....	197
Table A.12 – Enum Properties .....	201
Table A.13 – Enum value Properties .....	203
Table A.14 – Feature Table Properties .....	206
Table A.15 – BinaryBodyOffset Properties .....	207
Table A.16 – BinaryBodyReference Properties .....	208
Table A.17 – Group Metadata Properties .....	211
Table A.18 – Instanced 3D Model Feature Table Properties .....	213
Table A.19 – Metadata Entity Properties .....	219
Table A.20 – Point Cloud Feature Table Properties .....	220
Table A.21 – Properties Properties .....	226
Table A.22 – Property Table Properties .....	227
Table A.23 – Property Table Property Properties .....	230
Table A.24 – Root Property Properties .....	235
Table A.25 – Schema Properties .....	236
Table A.26 – Statistics Properties .....	238
Table A.27 – Class Statistics Properties .....	240
Table A.28 – Property Statistics Properties .....	241
Table A.29 – Style Properties .....	245
Table A.30 – Conditions Properties .....	248
Table A.31 – Meta Properties .....	250
Table A.32 – Subtree Properties .....	251
Table A.33 – Subtrees Properties .....	255
Table A.34 – Tile Properties .....	257
Table A.35 – Implicit tiling Properties .....	263
Table A.36 – Tileset Properties .....	265
Table D.1 – Availability bitstream lengths .....	318
Table D.2 – Availability offsets .....	319
Table F.1 – General metadata semantics .....	334
Table F.2 – Tileset metadata semantics .....	335
Table F.3 – Tile metadata semantics .....	336

## LIST OF FIGURES

---

Figure 1 – A tree of tiles .....	18
Figure 2 – A bounding region .....	24
Figure 3 – A bounding box .....	25
Figure 4 – A bounding sphere .....	26
Figure 5 – Bounding volumes for the root tile of a tileset. Building data from CyberCity3D. Imagery data from Bing Maps .....	27
Figure 6 – Structure of an example tileset with tiles that contain glTF content .....	31
Figure 7 – Elements of a tile JSON object .....	33
Figure 8 – An example of a tile that defines multiple contents .....	35
Figure 9 – Illustration of rendering options based on content groups .....	36
Figure 10 – A tileset that refers to other tilesets .....	38
Figure 11 – The chain of transforms for a tileset that refers to another tileset .....	39
Figure 12 – Bounding sphere for a terrain tile. ....	40
Figure 13 – Bounding spheres for the four child tiles. The children's content is completely inside the parent's bounding volume, but the children's bounding volumes are not since they are not tightly fit. ....	41
Figure 14 – Classic quadtree subdivision .....	42
Figure 15 – Quadtree with tight bounding volumes around each child .....	43
Figure 16 – Building data from CyberCity3D. Imagery data from Bing Maps .....	43
Figure 17 – Quadtree with non-uniform and overlapping tiles .....	44
Figure 18 – Building data from CyberCity3D. Imagery data from Bing Maps .....	45
Figure 19 – Example k-d tree. Note the non-uniform subdivision .....	46
Figure 20 – Traditional octree subdivision .....	47
Figure 21 – Non-uniform octree subdivision for a point cloud using additive refinement. Point Cloud of the Church of St Marie at Chappes, France by Prof. Peter Allen, Columbia University Robotics Lab. Scanning by Alejandro Troccoli and Matei Ciocarlie. ....	47
Figure 22 – Building data from CyberCity3D. Imagery data from Bing Maps .....	48
Figure 23 – Quadtree with tile coordinates .....	49
Figure 24 – Illustration of the different granularity levels for applying metadata .....	51
Figure 25 – Illustration of how to use metadata statistics for rendering an analytics .....	54
Figure 26 – 40 billion point cloud from the City of Montreal with ASPRS classification codes (CC-BY 4.0) .....	62
Figure 27 – A point cloud with two property tables, one storing metadata for groups of points and the other storing metadata for individual points .....	62
Figure 28 – Instanced tree models in Philadelphia from OpenTreeMap .....	63

Figure 29 – Instanced tree models where trees are assigned to groups with a per-instance feature ID attribute. One feature table stores per-group metadata and the other stores per-tree metadata .....	64
Figure 30 – Street level photogrammetry of San Francisco Ferry Building from Aerometrex. Left: per-texel colors showing the feature classification, e.g., roof, sky roof, windows, window frames, and AC units . Right: classification is used to determine rendering material properties, e.g., make the windows translucent .....	64
Figure 31 – An example of metadata associated with glTF models .....	65
Figure 32 – Data layout of a Batch Table .....	67
Figure 33 – An example showing how to access the binary body, based on the information from the JSON header .....	69
Figure 34 – Data layout of a Feature Table .....	71
Figure 35 – An example showing how to access the binary body, based on the information from the JSON header .....	73
Figure 36 – Data layout of a Batched 3D Model .....	74
Figure 37 – Data layout for Instanced 3D Models .....	79
Figure 38 – A box in the standard basis .....	83
Figure 39 – A box transformed into a rotated basis .....	83
Figure 40 – Illustration of the quantization that is used for the POSITION_QUANTIZED semantic .....	85
Figure 41 – Data layout for a Point Cloud .....	89
Figure 42 – Illustration of the quantization that is used for the POSITION_QUANTIZED semantic .....	93
Figure 43 – Data layout of a Composite tile .....	99
Figure 44 – A point cloud organized into a sparse octree. Data source: Trimble .....	103
Figure 45 – Subdivision in a quadtree .....	105
Figure 46 – Subdivision in an octree .....	105
Figure 47 – Coordinates of an octree node with a bounding box, and its child nodes .....	107
Figure 48 – Coordinates of an octree node with a bounding region, and its child nodes .....	108
Figure 49 – Examples of template URIs to identify the content for implicit tiles .....	109
Figure 50 – The structure of a subtree for implicit tiling .....	109
Figure 51 – A tree of subtrees representing an implicit tileset .....	110
Figure 52 – The computation of indices for accessing an availability bitstream, based on the coordinates of implicit tiles .....	111
Figure 53 – Illustration of a tile availability bitstream. Tiles that are available are represented with a 1 in the bitstream. ....	112
Figure 54 – Illustration of a content availability bitstream. Tiles that have associated content are represented with a 1 in the bitstream. ....	113
Figure 55 – Illustration of a child subtree availability bitstream. Tiles that are the roots of available subtrees are represented by a 1 in the bitstream. ....	114
Figure 56 – Illustration of storing the actual heights of individual tiles using the TILE_MAXIMUM_HEIGHT semantic .....	115
Figure 57 – Data layout for the subtree binary format .....	122

Figure 58 – Illustration of metadata that can be stored in a table .....	137
Figure 60 – Data layout for the buffers storing string metadata .....	139
Figure 61 – Data layout for the buffers storing string variable-length arrays .....	141
Figure 62 – Data layout for the buffers storing string variable-length arrays of strings .....	141
Figure 63 – Rendering of buildings with different colors, based on the height of the buildings .....	147
Figure C.1 – Batched 3D Models in 3D Tiles 1.0, and the corresponding representation in 3D Tiles 1.1 .....	313
Figure C.2 – Instanced 3D Models in 3D Tiles 1.0, and the corresponding representation in 3D Tiles 1.1 .....	314
Figure C.3 – Point Clouds in 3D Tiles 1.0, and the corresponding representation in 3D Tiles 1.1 .....	314
Figure D.1 – An example for the computation of the Morton index based on the tile coordinates on three levels of a quadtree .....	318
Figure D.2 – Illustration of how to compute the global Morton index of a tile, from the global Morton index of the root of the containing subtree, and the local Morton index of the tile in this subtree .....	320
Figure D.3 – Illustration of how to compute the global level of a tile, from the global level of the root of the containing subtree, and the local level of the tile in this subtree .....	321
Figure D.4 – Illustration of the computation of the global tile coordinates, from the global coordinates of the containing subtree, and the local coordinates of the tile in this subtree .....	322
Figure D.5 – Illustration of the computation of the coordinates of parent- and child tiles .....	323

## ABSTRACT

---



This document describes the specification for 3D Tiles, an open standard for streaming massive heterogeneous 3D geospatial datasets.

## KEYWORDS

---

The following are keywords to be used by search engines and document catalogues.

terrain, geospatial, gis, point cloud, spatial data, vector data, photogrammetry, gltf, 3d models, 3d tiles, metadata, implicit tiling

Bringing techniques from graphics research, the movie industry, and the game industry to 3D geospatial, 3D Tiles defines a spatial data structure and a set of tile formats designed for 3D, and optimized for streaming and rendering.

## SECURITY CONSIDERATIONS

---

No security considerations have been made for this document.

V

## SUBMITTING ORGANIZATIONS

---

The following organizations submitted this Document to the Open Geospatial Consortium (OGC):

- Cesium GS Inc.

VI

## SUBMITTERS

---

NAME	AFFILIATION	OGC MEMBER
Patrick Cozzi	Cesium GS, Inc.	Yes
Sean Lilley	Cesium GS, Inc.	Yes

VII

## FUTURE WORK

---

The 3D Tiles community anticipates that revisions to this Community Standard will be required to prescribe content appropriate to meet new use cases. These use cases may arise from either (or both) the external user and developer community or from OGC review and comments. Further, future revisions will be driven by any submitted change requests that document community uses cases and requirements.

1

# SCOPE

---

## SCOPE

---

3D Tiles is designed for streaming and rendering massive 3D geospatial content such as Photogrammetry, 3D Buildings, BIM/CAD, Instanced Features, and Point Clouds. It defines a hierarchical data structure and a set of tile formats which deliver renderable content. 3D Tiles does not define explicit rules for visualization of the content; a client may visualize 3D Tiles data however it sees fit.

2

# CONFORMANCE

---

## CONFORMANCE

---

Annex A of this document describes the Objects and Properties required to implement 3D Tiles. Conformance is relative to these elements and as partly expressed via the associated 3D Tiles JSON Schema.

All figures, examples, notes, and background information are informative.

3

# NORMATIVE REFERENCES

---

## NORMATIVE REFERENCES

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

EPSG: 4978, 2020. [https://epsg.org/crs\\_4978/WGS-84.html](https://epsg.org/crs_4978/WGS-84.html)

EPSG: 4979, 2020. [https://epsg.org/crs\\_4979/WGS-84.html](https://epsg.org/crs_4979/WGS-84.html)

L. Masinter: IETF RFC 2397, *The “data” URL scheme*. RFC Publisher (1998). <https://www.rfc-editor.org/info/rfc2397>.

F. Yergeau: IETF RFC 3629, *UTF-8, a transformation format of ISO 10646*. RFC Publisher (2003). <https://www.rfc-editor.org/info/rfc3629>.

T. Berners-Lee, R. Fielding, L. Masinter: IETF RFC 3986, *Uniform Resource Identifier (URI): Generic Syntax*. RFC Publisher (2005). <https://www.rfc-editor.org/info/rfc3986>.

T. Bray (ed.): IETF RFC 8259, *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC Publisher (2017). <https://www.rfc-editor.org/info/rfc8259>.

Khronos Group: glTF 2.0, 2021. <https://www.khronos.org/registry/glTF/specs/2.0/glTF-2.0.html>

Roger Lott: OGC 18-010r7, *Geographic information – Well-known text representation of coordinate reference systems*. Open Geospatial Consortium (2019). <https://docs.ogc.org/is/18-010r7/18-010r7.html>.

W3C css-color-3, CSS Color Module Level 3. <https://www.w3.org/TR/css-color-3/>.

4

# TERMS AND DEFINITIONS

---

# TERMS AND DEFINITIONS

This document uses the terms defined in [OGC Policy Directive 49](#), which is based on the ISO/IEC Directives, Part 2, Rules for the structure and drafting of International Standards. In particular, the word “shall” (not “must”) is the verb form used to indicate a requirement to be strictly followed to conform to this document and OGC documents do not use the equivalent phrases in the ISO/IEC Directives, Part 2.

This document also uses terms defined in the OGC Standard for Modular specifications ([OGC 08-131r3](#)), also known as the ‘ModSpec’. The definitions of terms such as standard, specification, requirement, and conformance test are provided in the ModSpec.

For the purposes of this document, the following additional terms and definitions apply.

## 4.1. Availability

Data specifying which tiles/contents/child subtrees exist within a single subtree of an implicit tileset

## 4.2. Bitstream

A boolean array stored as a sequence of bits rather than bytes.

## 4.3. Bounding Volume

A closed volume completely containing the union of a set of geometric objects. See [Wikipedia: Bounding volume](#)

## 4.4. Child subtree

A subtree reachable from an available tile in the bottommost row of a subtree of an implicit tileset

## 4.5. Entity

---

An entity, in the context of 3D Tiles metadata, is an instance of a metadata class, populated with property values conforming to a metadata class definition

## 4.6. Feature

---

In 3D Tiles, an individual component of a tile, such as a 3D model in a Batched 3D Model or a point in a Point Cloud which contains position, appearance, and metadata properties.

## 4.7. Geometric Error

---

The difference, in meters, of a tile's simplified representation of its source geometry used to calculate the screen space error introduced if a tile's content is rendered and its children's are not.

## 4.8. glTF

---

An API-neutral runtime asset delivery format for 3D assets.

## 4.9. Hierarchical Level of Detail (HLOD)

---

Decreasing the complexity of a 3D representation according to metrics such as object importance or distance from the tile to the observation point or camera position. Generally, higher levels of detail provide greater visual fidelity. See [Wikipedia: Level of detail](#)

## 4.10. Implicit tiling

---

A description of a tileset using recursive subdivision.

## 4.11. Implicit root tile

---

A tile that contains the `implicitTiling` property, and therefore denotes the root of an implicit tilesset.

## 4.12. Metadata

---

In the context of 3D Tiles, metadata is equivalent to the concept of attribution in other Open Geospatial Consortium standards. Metadata refers to non-spatial information that is assigned to entities at multiple granularities, including tilesets, tiles, and features.

## 4.13. Metadata class

---

The description of the structure of an entity that contains 3D Tiles metadata, consisting of multiple metadata properties

## 4.14. Metadata property

---

An element of a metadata class that defines a name and a type for the corresponding element of a metadata entity.

## 4.15. Octree

---

A 3D subdivision scheme that divides each bounding volume into 8 smaller bounding volumes along the midpoint of the x, y, and z axes.

## 4.16. Quadtree

---

A 2D subdivision scheme that divides each bounding volume into 4 smaller bounding volumes along the midpoint of the x and y axes.

## 4.17. Schema

---

A set of metadata classes and enums that define the structure and type of metadata

## 4.18. Screen-Space Error (SSE)

---

The difference, in pixels, of a tile's simplified representation of its source geometry introduced if a tile's content is rendered and its children's are not.

## 4.19. Spatial Coherence

---

The union of all content of the child tiles is completely inside the parent tile's bounding volume

## 4.20. Style

---

A set of expressions to be evaluated which modify how each feature in a tileset is displayed

## 4.21. Subtree

---

A fixed-size section of the tree of an implicit tileset that contains availability information.

## 4.22. Subtree file

---

A binary file storing information about a specific subtree of an implicit tileset

## 4.23. Subdivision scheme

---

A recursive pattern of dividing a parent tile into smaller child tiles occupying the same area. This is done by uniformly dividing the bounding volume of the parent tile with an implicit tileset.

## 4.24. Template URI

---

A URI pattern containing tile coordinates for directly addressing tiles in an implicit tileset.

## 4.25. Tile

---

In 3D Tiles, a subset of a tileset containing a reference to renderable content and the metadata, such as the content's bounding volume, which is used by a client to determine if the content is rendered.

## 4.26. Tile Content

---

A binary blob containing information necessary to render a tile which is an instance of a specific tile format (Batched 3D Model, Instanced 3D Model, Point Clouds, Composite, or glTF).

## 4.27. Tile Format

---

The structure or layout of tile content data, (Batched 3D Model, Instanced 3D Model, Point Clouds, Composite, or glTF).

## 4.28. Tileset

---

In 3D Tiles, a collection of 3D Tiles tile instances organized into a spatial data structure and additional metadata, such that the aggregation of these tiles represent some 3D content at various levels of detail.

5

# CONVENTIONS

---

## CONVENTIONS

---

No conventions are specified in this document.

6

# 3D TILES FORMAT SPECIFICATION

---

# 3D TILES FORMAT SPECIFICATION

## 6.1. Introduction

In 3D Tiles, a *tileset* is a set of *tiles* organized in a spatial data structure, the *tree*. A tileset is described by at least one tileset JSON file containing tileset metadata and a tree of tile objects, each of which may reference renderable content.

[glTF 2.0](#) is the primary tile format for 3D Tiles. glTF is an open specification designed for the efficient transmission and loading of 3D content. A glTF asset includes geometry and texture information for a single tile, and may be extended to include metadata, model instancing, and compression. glTF may be used for a wide variety of 3D content including:

- Heterogeneous 3D models. E.g. textured terrain and surfaces, 3D building exteriors and interiors, massive models
- 3D model instances. E.g. trees, windmills, bolts
- Massive point clouds

See [glTF Tile Format](#) for more details.

Tiles may also reference the legacy 3D Tiles 1.0 formats listed below. These formats were deprecated in 3D Tiles 1.1 and may be removed in a future version of 3D Tiles.

**Table 1 – Legacy tile formats and common uses**

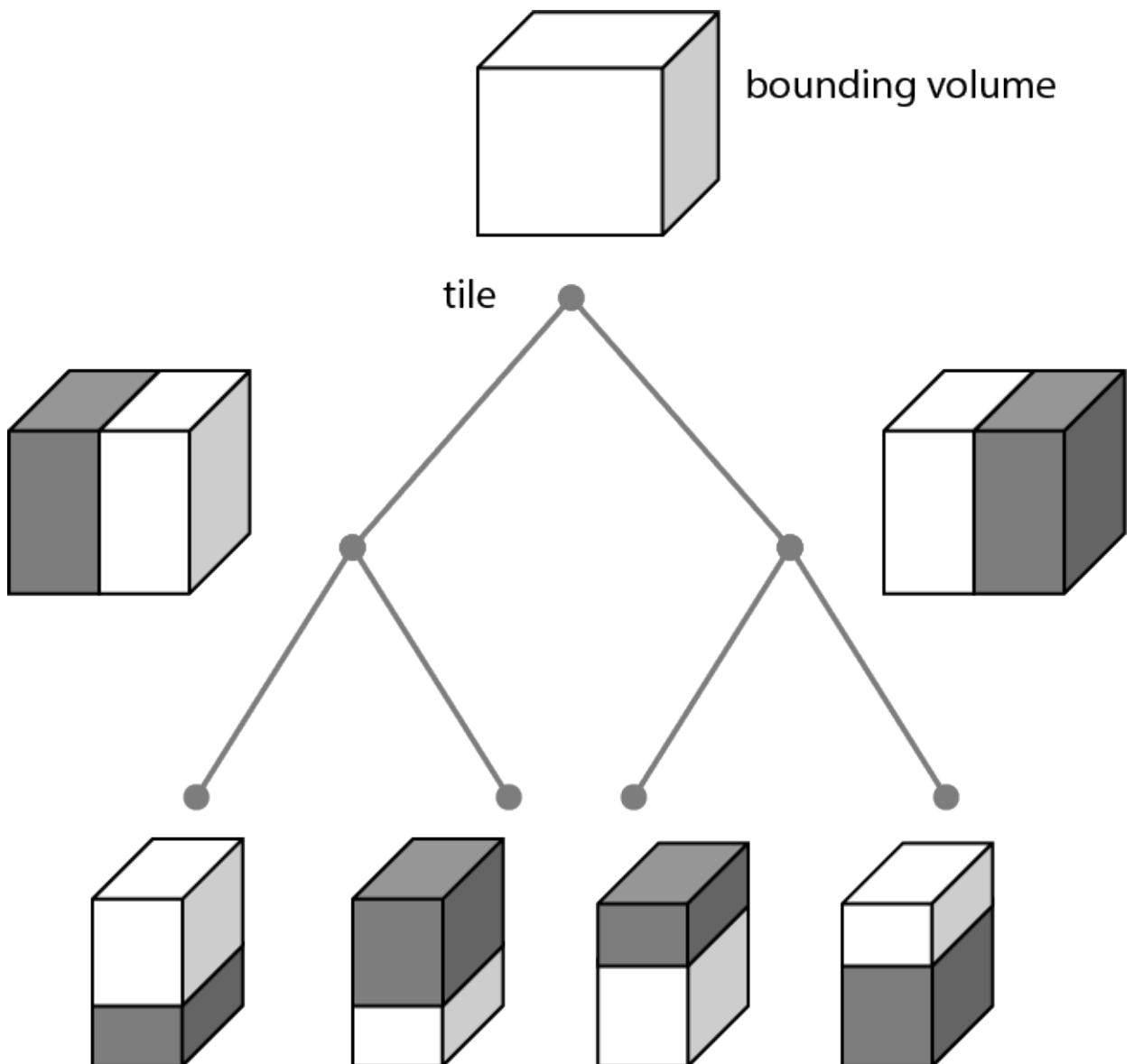
LEGACY FORMAT	USES
Batched 3D Model (b3dm)	Heterogeneous 3D models
Instanced 3D Model (i3dm)	3D model instances
Point Cloud (pnts)	Massive number of points
Composite (cmpt)	Concatenate tiles of different formats into one tile

A tile's *content* is an individual instance of a tile format. A tile may have multiple contents.

The content references a set of *features*, such as 3D models representing buildings or trees, or points in a point cloud. Each feature has position and appearance properties and additional

application-specific properties. A client may choose to select features at runtime and retrieve their properties for visualization or analysis.

Tiles are organized in a tree which incorporates the concept of Hierarchical Level of Detail (HLOD) for optimal rendering of spatial data. Each tile has a *bounding volume*, an object defining a spatial extent completely enclosing its content. The tree has spatial coherence; the content for child tiles are completely inside the parent's bounding volume.



**Figure 1 – A tree of tiles**

A tileset may use a 2D spatial tiling scheme similar to raster and vector tiling schemes (like a Web Map Tile Service (WMTS) or XYZ scheme) that serve predefined tiles at several levels of detail (or zoom levels). However since the content of a tileset is often non-uniform or may not easily be organized in only two dimensions, the tree can be any spatial data structure with spatial coherence, including k-d trees, quadtrees, octrees, and grids. Implicit tiling defines a concise representation of quadtrees and octrees.

Application-specific *metadata* may be provided at multiple granularities within a tileset. Metadata may be associated with high-level entities like tilesets, tiles, contents, or features, or with individual vertices and texels. Metadata conforms to a well-defined type system described by the 3D Metadata Specification, which may be extended with application- or domain-specific semantics.

Optionally a 3D Tiles Style, or *style*, may be applied to a tileset. A style defines expressions to be evaluated which modify how each feature is displayed.

## 6.2. File Extensions and Media Types

---

3D Tiles uses the following file extensions and Media Types.

- Tileset files should use the `.json` extension and the [application/json](#) Media Type.
- Tile content files should use the file extensions and Media Type specific to their tile format specification.
- Metadata schema files should use the `.json` extension and the [application/json](#) Media Type.
- Tileset style files should use the `.json` extension and the [application/json](#) Media Type.
- JSON subtree files should use the `.json` extension and the [application/json](#) Media Type.
- Binary subtree files should use the `.subtree` extension and the [application/octet-stream](#) Media Type.
- Files representing binary buffers should use the `.bin` extension and [application/octet-stream](#) Media Type.

Explicit file extensions are optional. Valid implementations may ignore it and identify a content's format by the `magic` field in its header.

## 6.3. JSON encoding

---

3D Tiles has the following restrictions on JSON formatting and encoding.

1. JSON shall use UTF-8 encoding without BOM.
2. All strings defined in this spec (properties names, enums) use only ASCII charset and shall be written as plain text, without JSON escaping.
3. Non-ASCII characters that appear as property *values* in JSON may be escaped.

4. Names (keys) within JSON objects shall be unique, i.e., duplicate keys aren't allowed.
5. Some 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, as defined in [RFC 8259, Section 6](#).

## 6.4. URLs

---

3D Tiles uses URIs to reference tile content. These URIs may point to [relative external references \(RFC3986\)](#) or be data URIs that embed resources in the JSON. Embedded resources use the "data" URL scheme ([RFC2397](#)).

When the URI is relative, its base is always relative to the referring tileset JSON file.

Client implementations are required to support relative external references and embedded resources. Optionally, client implementations may support other schemes (such as `http://`). All URIs shall be valid and resolvable.

## 6.5. Units

---

The unit for all linear distances is meters.

All angles are in radians.

## 6.6. Coordinate reference system (CRS)

---

3D Tiles uses a right-handed Cartesian coordinate system; that is, the cross product of x and y yields z. 3D Tiles defines the z axis as up for local Cartesian coordinate systems. A tileset's global coordinate system will often be in a [WGS 84](#) Earth-centered, Earth-fixed (ECEF) reference frame ([EPSG 4978](#)), but it doesn't have to be, e.g., a power plant may be defined fully in its local coordinate system for use with a modeling tool without a geospatial context.

The CRS of a tileset may be defined explicitly, as part of the tileset metadata. The metadata for the tileset can contain a property that has the `TILESET_CRS_GEOCENTRIC` semantic, which is a string that represents the EPSG Geodetic Parameter Dataset identifier.

An additional tile transform may be applied to transform a tile's local coordinate system to the parent tile's coordinate system.

The region bounding volume specifies bounds using a geographic coordinate system (latitude, longitude, height), specifically, [EPSG 4979](#). The reference ellipsoid is assumed to be the same as the reference ellipsoid of the tileset.

## 6.7. Concepts

---

### 6.7.1. Tiles

Tiles consist of metadata used to determine if a tile is rendered, a reference to the renderable content, and an array of any children tiles.

#### 6.7.1.1. Tile Content

A tile can be associated with renderable content. A tile can either have a single `tile.content` object, or multiple content objects, stored in a `tile.contents` array. The latter allows for flexible tileset structures: for example, a single tile may contain multiple representations of the same geometry data.

The `content.uri` of each content object refers to the tile's content in one of the tile formats that are defined in the Tile format specifications), or another tileset JSON to create a tileset of tilesets (see External tilesets).

The `content.group` property assigns the content to a group. Contents of different tiles or the contents of a single tile can be assigned to groups in order to categorize the content. Additionally, each group can be associated with Metadata.

Each content can be associated with a bounding volume. While `tile.boundingVolume` is a bounding volume that encloses *all* contents of the tile, each individual content `boundingVolume` is a tightly fit bounding volume enclosing just the respective content. More details about the role of tile- and content bounding volumes are given in the bounding volume section.

#### 6.7.1.2. Geometric error

Tiles are structured into a tree incorporating *Hierarchical Level of Detail* (HLOD) so that at runtime a client implementation will need to determine if a tile is sufficiently detailed for rendering and if the content of tiles should be successively refined by children tiles of higher resolution. An implementation will consider a maximum allowed *Screen-Space Error* (SSE), the error measured in pixels.

A tile's geometric error defines the selection metric for that tile. Its value is a nonnegative number that specifies the error, in meters, of the tile's simplified representation of its source geometry. Generally, the root tile will have the largest geometric error, and each successive level

of children will have a smaller geometric error than its parent, with leaf tiles having a geometric error of or close to 0.

In a client implementation, geometric error is used with other screen space metrics—e.g., distance from the tile to the camera, screen size, and resolution—to calculate the SSE introduced if this tile is rendered and its children are not. If the introduced SSE exceeds the maximum allowed, then the tile is refined and its children are considered for rendering.

The geometric error is formulated based on a metric like point density, mesh or texture decimation, or another factor specific to that tileset. In general, a higher geometric error means a tile will be refined more aggressively, and children tiles will be loaded and rendered sooner.

### 6.7.1.3. Refinement

Refinement determines the process by which a lower resolution parent tile renders when its higher resolution children are selected to be rendered. Permitted refinement types are replacement ("REPLACE") and additive ("ADD"). If the tile has replacement refinement, the children tiles are rendered in place of the parent, that is, the parent tile is no longer rendered. If the tile has additive refinement, the children are rendered in addition to the parent tile.

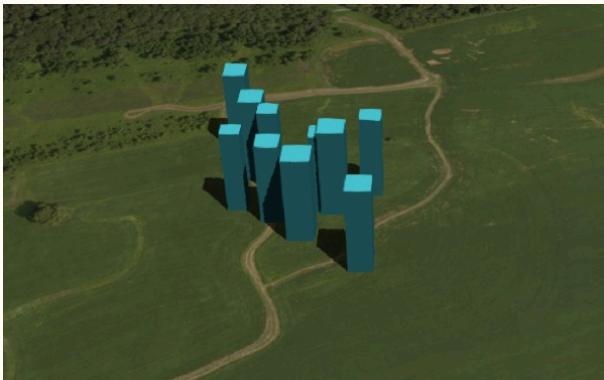
A tileset can use replacement refinement exclusively, additive refinement exclusively, or any combination of additive and replacement refinement.

A refinement type is required for the root tile of a tileset; it is optional for all other tiles. When omitted, a tile inherits the refinement type of its parent.

#### 6.7.1.3.1. Replacement

If a tile uses replacement refinement, when refined it renders its children in place of itself.

**Table 2 – A tile and a refined tile using replacement refinement**

PARENT TILE	REFINED
	

#### 6.7.1.3.2. Additive

If a tile uses additive refinement, when refined it renders itself and its children simultaneously.

**Table 3 – A tile and a refined tile using additive refinement**



#### 6.7.1.4. Bounding volumes

A bounding volume defines the spatial extent enclosing a tile or a tile's content. To support tight fitting volumes for a variety of datasets such as regularly divided terrain, cities not aligned with a line of latitude or longitude, or arbitrary point clouds, the bounding volume types include an oriented bounding box, a bounding sphere, and a geographic region defined by minimum and maximum latitudes, longitudes, and heights.

**Table 4 – Different bounding volume types for a tile**



#### 6.7.1.4.1. Region

The `boundingVolume.region` property is an array of six numbers that define the bounding geographic region with latitude, longitude, and height coordinates with the order [west, south, east, north, minimum height, maximum height]. Latitudes and longitudes are in the WGS 84 datum as defined in [EPSG 4979](#) and are in radians. Heights are in meters above (or below) the [WGS 84 ellipsoid](#).

**NOTE** The latitude and longitude values are given in *radians*, deviating from the EPSG 4979 definition, where they are given in *degrees*. The choice of using radians is due to internal computations usually taking place in radians — for example, when converting cartographic to Cartesian coordinates.



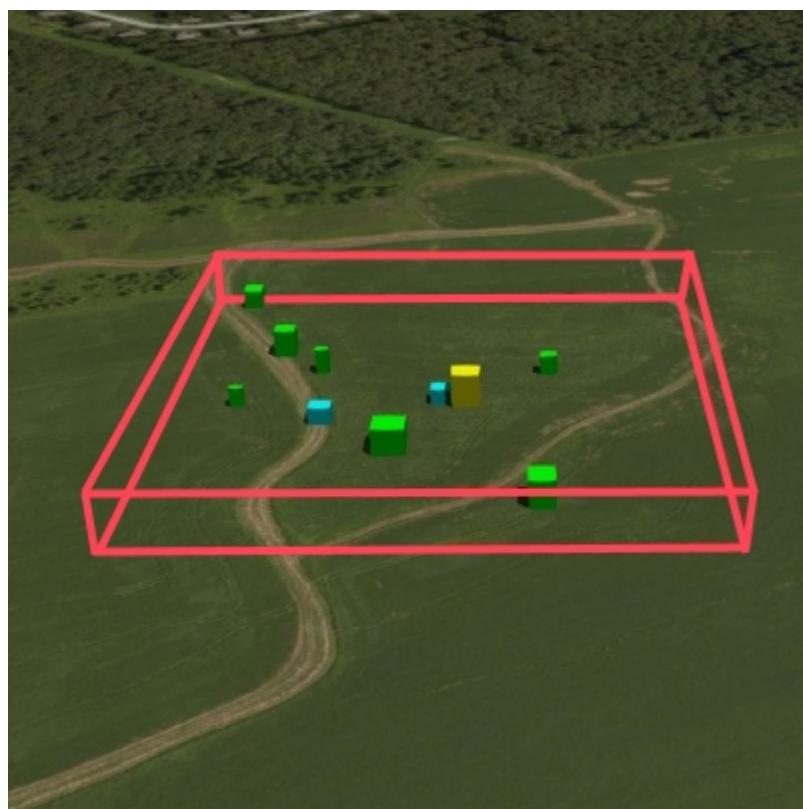
**Figure 2 – A bounding region**

```
"boundingVolume": {  
  "region": [  
    -1.3197004795898053,  
    0.6988582109,  
    -1.3196595204101946,  
    0.6988897891,  
    0,  
    20  
  ]  
}
```

#### 6.7.1.4.2. Box

The `boundingVolume.box` property is an array of 12 numbers that define an oriented bounding box in a right-handed 3-axis (x, y, z) Cartesian coordinate system where the z-axis is up. The first three elements define the x, y, and z values for the center of the box. The next three elements (with indices 3, 4, and 5) define the x-axis direction and half-length. The next three elements (indices 6, 7, and 8) define the y-axis direction and half-length. The last three elements (indices 9, 10, and 11) define the z-axis direction and half-length.

**NOTE** The representation that is used for an oriented bounding box in 3D Tiles is versatile and compact: In addition the center position, the array contains the elements of a  $3 \times 3$  matrix. The columns of this matrix are the images of unit vectors under a transformation, and therefore uniquely and compactly define the scaling and orientation of the bounding box.



**Figure 3 – A bounding box**

```
"boundingVolume": {  
  "box": [  
    0, 0, 10,  
    100, 0, 0,  
    0, 100, 0,  
    0, 0, 10  
  ]  
}
```

#### 6.7.1.4.3. Sphere

The `boundingVolume.sphere` property is an array of four numbers that define a bounding sphere. The first three elements define the x, y, and z values for the center of the sphere in a right-handed 3-axis (x, y, z) Cartesian coordinate system where the z-axis is up. The last element (with index 3) defines the radius in meters.



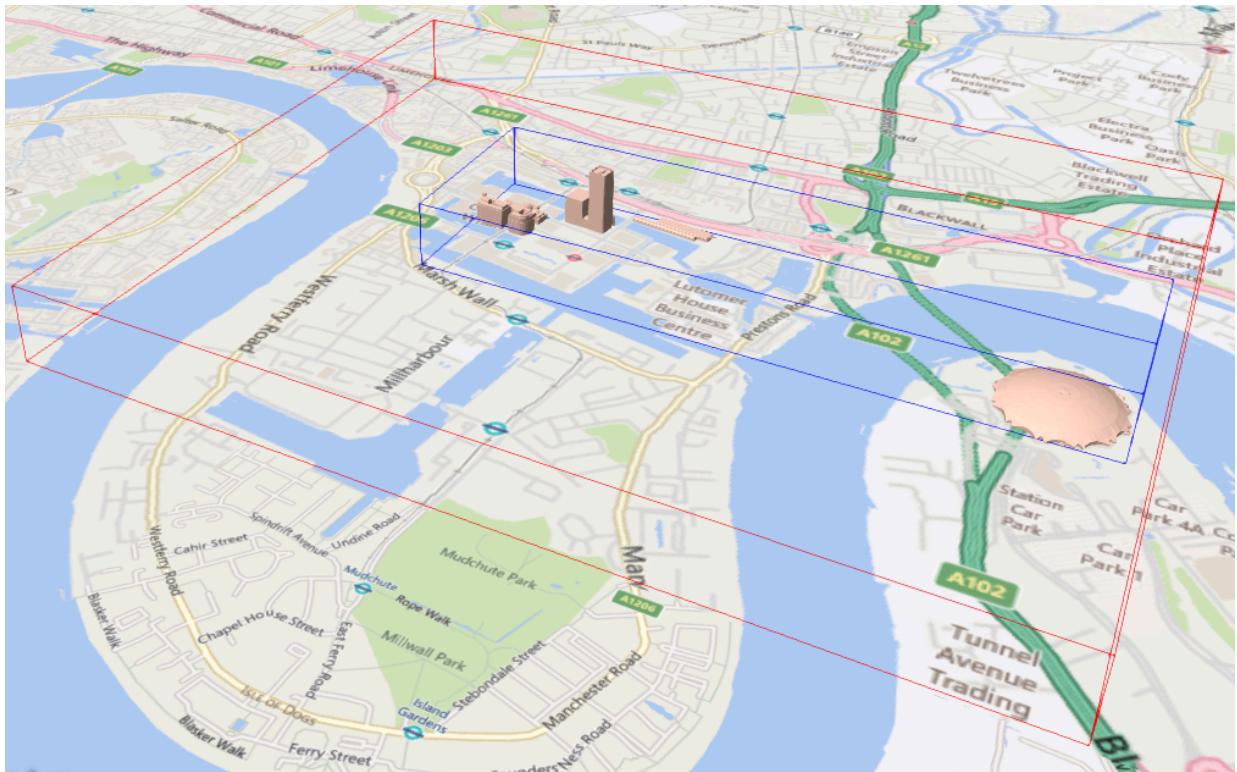
Figure 4 – A bounding sphere

```
"boundingVolume": {  
  "sphere": [  
    0,  
    0,  
    10,  
    141.4214  
  ]  
}
```

#### 6.7.1.4.4. Content Bounding Volume

The bounding volume can be given for each tile, via the `tile.boundingVolume` property. Additionally, it is possible to specify the bounding volume for each tile content individually. The `content.boundingVolume` may be a more tight-fitting bounding volume. This enables tight view frustum culling, excluding from rendering any content not in the volume of what is potentially in view. When it is not defined, the tile's bounding volume is still used for culling (see Grids).

The screenshot below shows the bounding volumes for the root tile for Canary Wharf. The tile.boundingVolume, shown in red, encloses the entire area of the tiles; content.boundingVolume shown in blue, encloses just the four features (models) in the root tile.



**Figure 5 – Bounding volumes for the root tile of a tilesset.**  
Building data from [CyberCity3D](#). Imagery data from [Bing Maps](#)

#### 6.7.1.4.5. Extensions

Other bounding volume types are supported through extensions.

- [3DTILES\\_bounding\\_volume\\_S2](#)

#### 6.7.1.5. Viewer request volume

A tile's viewerRequestVolume can be used for combining heterogeneous datasets, and can be combined with external tilesets.

The following example has a point cloud inside a building. The point cloud tile's boundingVolume is a sphere with a radius of 1.25. It also has a larger sphere with a radius of 15 for the viewerRequestVolume. Since the geometricError is zero, the point cloud tile's content is always rendered (and initially requested) when the viewer is inside the large sphere defined by viewerRequestVolume.

```
{  
  "children": [{
```

```

  "transform": [
    4.843178171884396, 1.2424271388626869, 0, 0,
    -0.7993325488216595, 3.1159251367235608, 3.8278032889280675, 0,
    0.9511533376784163, -3.7077466670407433, 3.2168186118075526, 0,
    1215001.7612985559, -4736269.697480114, 4081650.708604793, 1
  ],
  "boundingVolume": {
    "box": [
      0, 0, 6.701,
      3.738, 0, 0,
      0, 3.72, 0,
      0, 0, 13.402
    ]
  },
  "geometricError": 32,
  "content": {
    "uri": "building.glb"
  }
}, {
  "transform": [
    0.968635634376879, 0.24848542777253732, 0, 0,
    -0.15986650990768783, 0.6231850279035362, 0.7655606573007809, 0,
    0.19023066741520941, -0.7415493329385225, 0.6433637229384295, 0,
    1215002.037130238, -4736270.772726648, 4081651.6414821907, 1
  ],
  "viewerRequestVolume": {
    "sphere": [0, 0, 0, 15]
  },
  "boundingVolume": {
    "sphere": [0, 0, 0, 1.25]
  },
  "geometricError": 0,
  "content": {
    "uri": "points.glb"
  }
}
]
}

```

For more on request volumes, see the [sample tileset](#).

## 6.7.1.6. Transforms

### 6.7.1.6.1. Tile transforms

To support local coordinate systems—e.g., so a building tileset inside a city tileset can be defined in its own coordinate system, and a point cloud tileset inside the building could, again, be defined in its own coordinate system—each tile has an optional `transform` property.

The `transform` property is a  $4 \times 4$  affine transformation matrix, stored in column-major order, that transforms from the tile's local coordinate system to the parent tile's coordinate system—or the tileset's coordinate system in the case of the root tile.

**NOTE** The storage of the transform matrix in column-major order follows the conventions that are common in graphics programming APIs like OpenGL, meaning that the elements in the `transform` array directly correspond to the entries of a  $4 \times 4$  matrix in these systems.

The transform property applies to

- tile.content
  - Each feature's position.
  - Each feature's normal should be transformed by the top-left  $3 \times 3$  matrix of the inverse-transpose of transform to account for correct vector transforms when scale is used.
  - content.boundingVolume, except when content.boundingVolume.region is defined, which is explicitly in EPSG:4979 coordinates.
- tile.boundingVolume, except when tile.boundingVolume.region is defined, which is explicitly in EPSG:4979 coordinates.
- tile.viewerRequestVolume, except when tile.viewerRequestVolume.region is defined, which is explicitly in EPSG:4979 coordinates.

The transform property scales the geometricError by the largest scaling factor from the matrix.

When transform is not defined, it defaults to the identity matrix:

```
[  
  1.0, 0.0, 0.0, 0.0,  
  0.0, 1.0, 0.0, 0.0,  
  0.0, 0.0, 1.0, 0.0,  
  0.0, 0.0, 0.0, 1.0  
]
```

The transformation from each tile's local coordinate system to the tileset's global coordinate system is computed by a top-down traversal of the tileset and by post-multiplying a child's transform with its parent's transform like a traditional scene graph or node hierarchy in computer graphics.

### 6.7.1.6.2. glTF transforms

glTF defines its own node hierarchy and uses a y-up coordinate system. Any transforms specific to a tile format and the tile.transform property are applied after these transforms are resolved.

#### 6.7.1.6.2.1. glTF node hierarchy

First, glTF node hierarchy transforms are applied according to the glTF specification.

#### 6.7.1.6.2.2. y-up to z-up

Next, for consistency with the z-up coordinate system of 3D Tiles, gITFs shall be transformed from y-up to z-up at runtime. This is done by rotating the model about the x-axis by  $\pi/2$  radians. Equivalently, apply the following matrix transform (shown here as row-major):

```
[  
  1.0, 0.0, 0.0, 0.0,  
  0.0, 0.0, -1.0, 0.0,  
  0.0, 1.0, 0.0, 0.0,  
  0.0, 0.0, 0.0, 1.0  
]
```

More broadly the order of transformations is:

1. gITF node hierarchy transformations
2. gITF y-up to z-up transform
3. Tile transform

**NOTE** When working with source data that is inherently z-up, such as data in WGS 84 coordinates or in a local z-up coordinate system, a common workflow is:

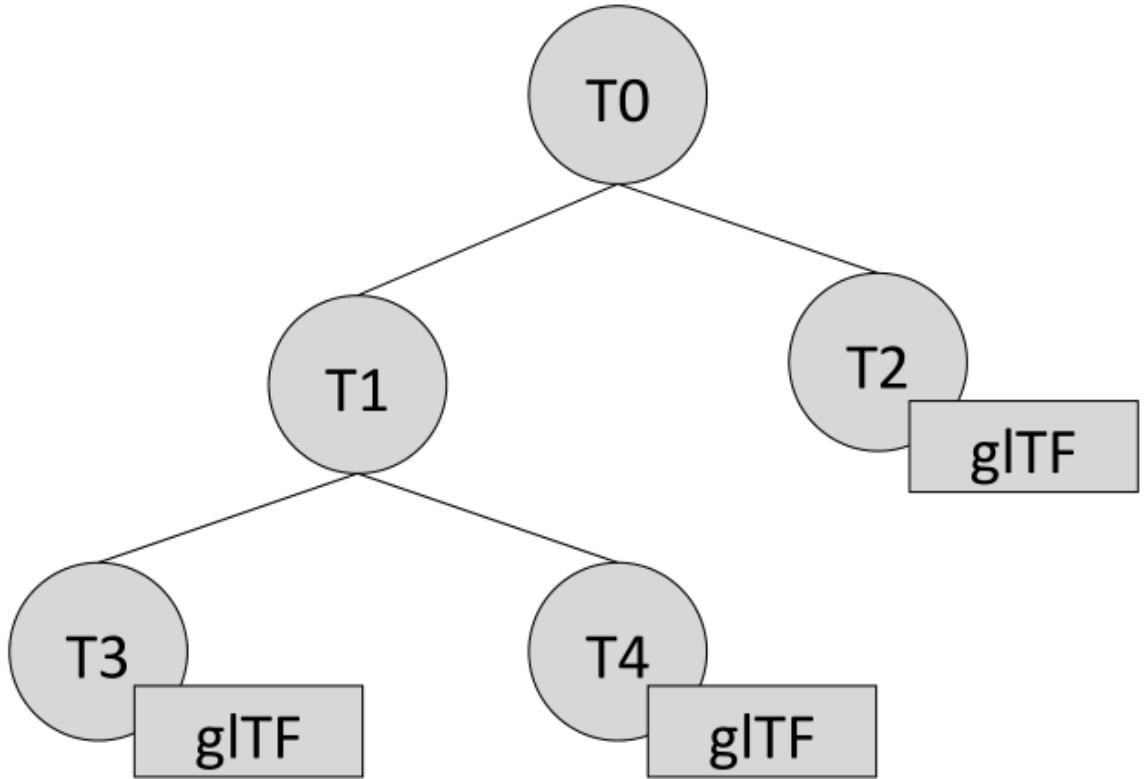
- Mesh data, including positions and normals, are not modified – they remain z-up.
- The root node matrix specifies a column-major z-up to y-up transform. This transforms the source data into a y-up coordinate system as required by gITF.
- At runtime the gITF is transformed back from y-up to z-up with the matrix above. Effectively the transforms cancel out.

Example gITF root node:

```
"nodes": [  
  {  
    "matrix": [1,0,0,0,0,0,-1,0,0,1,0,0,0,0,0,1],  
    "mesh": 0,  
    "name": "rootNode"  
  }  
]
```

#### 6.7.1.6.3. Example

For an example of the computed transforms (`transformToRoot` in the code above) for a tileset, consider:



**Figure 6 – Structure of an example tileset with tiles that contain glTF content**

The computed transform for each tile is:

- T0: [T0]
- T1: [T0][T1]
- T2: [T0][T2]
- T3: [T0][T1][T3]
- T4: [T0][T1][T4]

The full computed transforms, taking into account the glTF y-up to z-up transform and glTF Transforms are

- T0: [T0]
- T1: [T0][T1]
- T2: [T0][T2][glTF y-up to z-up][glTF transform]
- T3: [T0][T1][T3][glTF y-up to z-up][glTF transform]

- T4: [T0][T1][T4][glTF y-up to z-up][glTF transform]

#### 6.7.1.6.4. Implementation example

*This section is informative*

The following JavaScript code shows how to compute this using Cesium's [Matrix4](#) and [Matrix3](#) types.

```
function computeTransforms(tileset) {
  const root = tileset.root;
  const transformToRoot = defined(root.transform) ? Matrix4.fromArray(root.
transform) : Matrix4.IDENTITY;

  computeTransform(root, transformToRoot);
}

function computeTransform(tile, transformToRoot) {
  // Apply 4x4 transformToRoot to this tile's positions and bounding volumes

  let normalTransform = Matrix4.getRotation(transformToRoot, new Matrix4());
  normalTransform = Matrix3.inverseTranspose(normalTransform, normalTransform);
  // Apply 3x3 normalTransform to this tile's normals

  const children = tile.children;
  if (defined(children)) {
    const length = children.length;
    for (let i = 0; i < length; ++i) {
      const child = children[i];
      let childToRoot = defined(child.transform) ? Matrix4.fromArray(child.
transform) : Matrix4.clone(Matrix4.IDENTITY);
      childToRoot = Matrix4.multiplyTransformation(transformToRoot,
childToRoot, childToRoot);
      computeTransform(child, childToRoot);
    }
  }
}
```

#### 6.7.1.7. Tile JSON

A tile JSON object consists of the following properties.

# tile

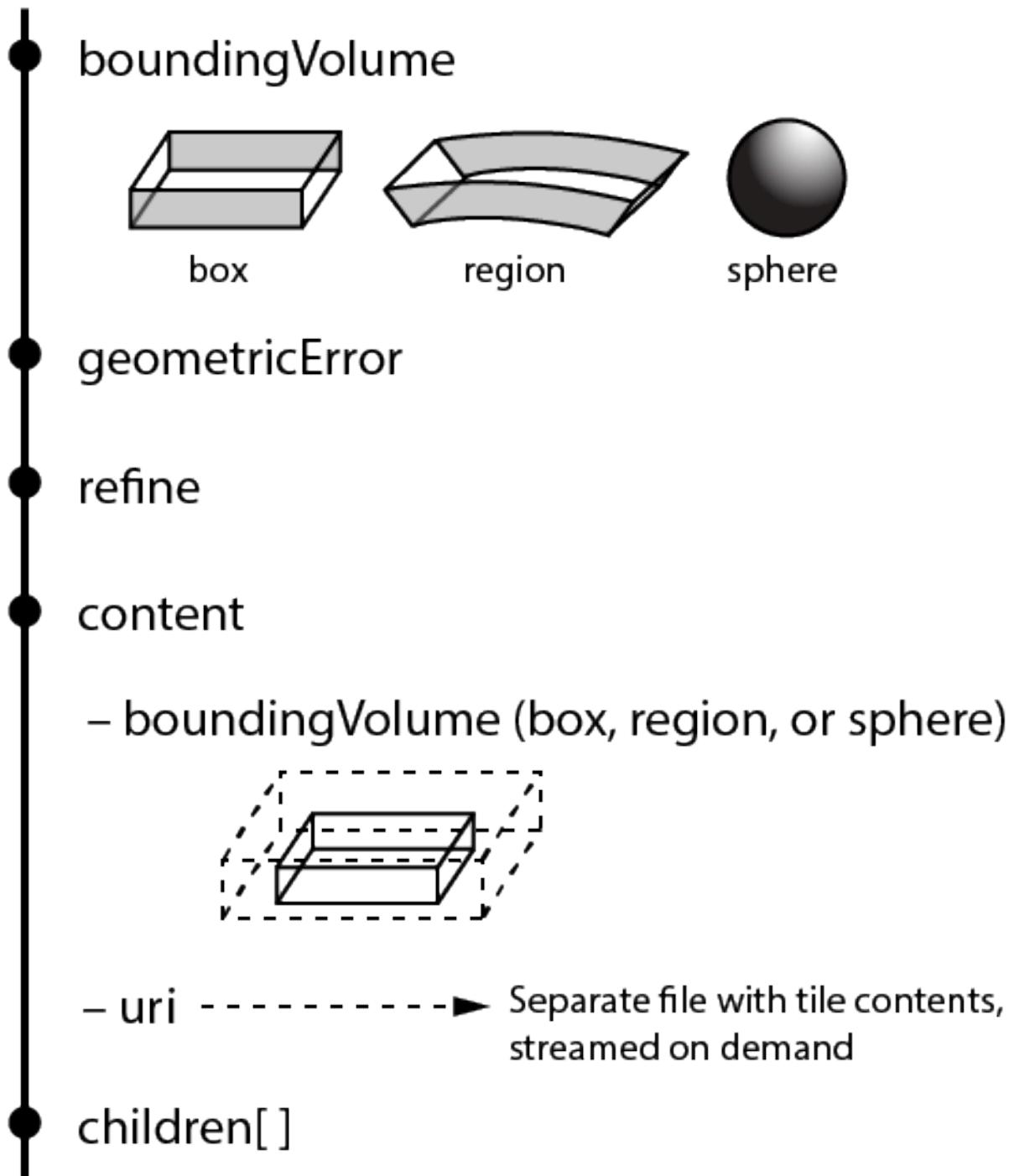


Figure 7 – Elements of a tile JSON object

The following example shows one non-leaf tile.

```
{  
  "boundingVolume": {  
    "region": [
```

```

        -1.2419052957251926,
        0.7395016240301894,
        -1.2415404171917719,
        0.7396563300150859,
        0,
        20.4
    ],
},
"geometricError": 43.88464075650763,
"refine" : "ADD",
"content": {
    "boundingVolume": {
        "region": [
            -1.2418882438584018,
            0.7395016240301894,
            -1.2415422846940714,
            0.7396461198389616,
            0,
            19.4
        ]
    },
    "uri": "2/0/0.glb"
},
"children": [...]
}

```

The `boundingVolume` defines a volume enclosing the tile, and is used to determine which tiles to render at runtime. The above example uses a `region` volume, but other bounding volumes, such as `box` or `sphere`, may be used.

The `geometricError` property is a nonnegative number that defines the error, in meters, introduced if this tile is rendered and its children are not. At runtime, the geometric error is used to compute *Screen-Space Error* (SSE), the error measured in pixels. The SSE determines if a tile is sufficiently detailed for the current view or if its children should be considered, see Geometric error.

The optional `viewerRequestVolume` property (not shown above) defines a volume, using the same schema as `boundingVolume`, that the viewer shall be inside of before the tile's content will be requested and before the tile will be refined based on `geometricError`. See the Viewer request volume section.

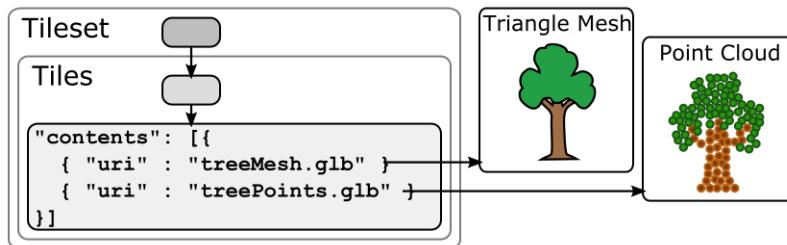
The `refine` property is a string that is either "REPLACE" for replacement refinement or "ADD" for additive refinement, see Refinement. It is required for the root tile of a tileset; it is optional for all other tiles. A tileset can use any combination of additive and replacement refinement. When the `refine` property is omitted, it is inherited from the parent tile.

The `content` property is an object that describes the tile content. A file extension is not required for `content.uri`. A content's tile format can be identified by the `magic` field in its header, or else as an external tileset if the content is JSON.

The `content.boundingVolume` property defines an optional bounding volume similar to the top-level `tile.boundingVolume` property. But unlike the top-level `boundingVolume` property, `content.boundingVolume` is a tightly fit bounding volume enclosing just the tile's content.

It is also possible to define multiple contents for a tile: The `contents` property (not shown above) is an array containing one or more contents. `contents` and `content` are mutually exclusive. When a tile has a single content it should use `content` for backwards

compatibility with engines that only support 3D Tiles 1.0. Multiple contents allow for different representations of the tile content – for example, one as a triangle mesh and one as a point cloud:

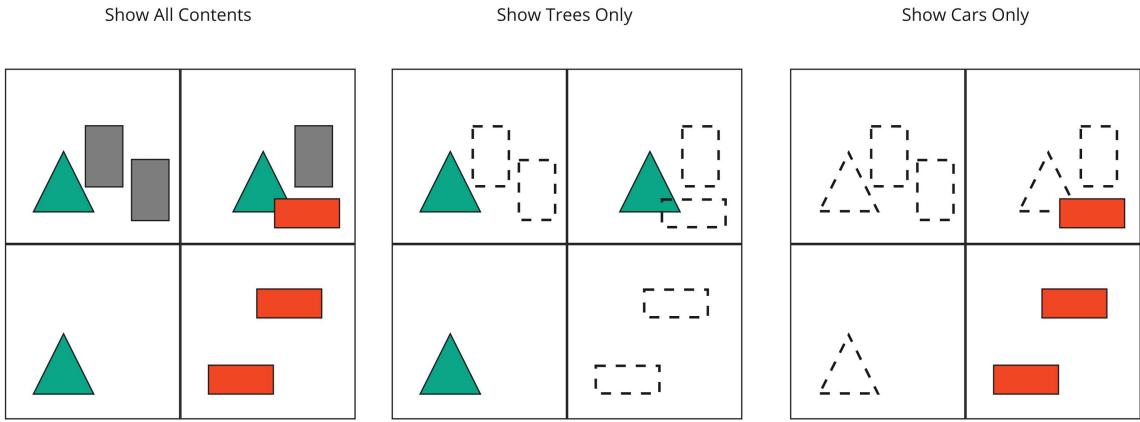


**Figure 8** – An example of a tile that defines multiple contents

Contents can also be arranged into groups, using the `content.group` property:

```
{
  "root": {
    "refine": "ADD",
    "geometricError": 0.0,
    "boundingVolume": {
      "region": [-1.707, 0.543, -1.706, 0.544, 203.895, 253.113]
    },
    "contents": [
      {
        "uri": "buildings.glb",
        "group": 0
      },
      {
        "uri": "trees.glb",
        "group": 1
      },
      {
        "uri": "cars.glb",
        "group": 2
      }
    ]
  }
}
```

These groups can be associated with group metadata: The value of the `content.group` property is an index into the array of groups that are defined in a top-level array of the tileset. Each element of this array is a metadata entity, as defined in the metadata section. This allows applications to perform styling or filtering based on the group that the content belongs to:



**Figure 9 — Illustration of rendering options based on content groups**

The optional `transform` property (not shown above) defines a  $4 \times 4$  affine transformation matrix that transforms the tile's content, `boundingVolume`, and `viewerRequestVolume` as described in the Tile transform section.

The optional `implicitTiling` property (not shown above) defines how the tile is subdivided and where to locate content resources. See Implicit Tiling.

The `children` property is an array of objects that define child tiles. Each child tile's content is fully enclosed by its parent tile's `boundingVolume` and, generally, a `geometricError` less than its parent tile's `geometricError`. For leaf tiles, the length of this array is zero, and `children` may not be defined. See the Tileset JSON section below.

The full JSON schema can be found in `tile.schema.json`.

## 6.7.2. Tileset JSON

3D Tiles uses one main tileset JSON file as the entry point to define a tileset. Both entry and external tileset JSON files are not required to follow a specific naming convention.

Here is a subset of the tileset JSON used for Canary Wharf:

```
{
  "asset" : {
    "version": "1.1",
    "tilesetVersion": "e575c6f1-a45b-420a-b172-6449fa6e0a59",
  },
  "properties": {
    "Height": {
      "minimum": 1,
      "maximum": 241.6
    }
  },
  "geometricError": 494.50961650991815,
  "root": {
    "boundingVolume": {
      "center": [
        -0.00010000000000000002,
        0.00010000000000000002,
        0.00010000000000000002
      ],
      "size": [
        0.00010000000000000002,
        0.00010000000000000002,
        0.00010000000000000002
      ]
    }
  }
}
```

```

    "region": [
      -0.0005682966577418737,
      0.8987233516605286,
      0.00011646582098558159,
      0.8990603398325034,
      0,
      241.6
    ],
  },
  "geometricError": 268.37878244706053,
  "refine": "ADD",
  "content": {
    "uri": "0/0/0.glb",
    "boundingVolume": {
      "region": [
        -0.0004001690908972599,
        0.8988700116775743,
        0.00010096729722787196,
        0.8989625664878067,
        0,
        241.6
      ]
    }
  },
  "children": [...]
}

```

The tileset JSON has four top-level properties: asset, properties, geometricError, and root.

asset is an object containing metadata about the entire tileset. The asset.version property is a string that defines the 3D Tiles version, which specifies the JSON schema for the tileset and the base set of tile formats. The tilesetVersion property is an optional string that defines an application-specific version of a tileset, e.g., for when an existing tileset is updated.

**NOTE**The tilesetVersion can be used as a query parameter when requesting content to avoid using outdated content from a cache.

properties is an object containing objects for each per-feature property in the tileset. This tileset JSON snippet is for 3D buildings, so each tile has building models, and each building model has a Height property (see Batch Table). The name of each object in properties matches the name of a per-feature property, and its value defines its minimum and maximum numeric values, which are useful, for example, for creating color ramps for styling.

geometricError is a nonnegative number that defines the error, in meters, that determines if the tileset is rendered. At runtime, the geometric error is used to compute *Screen-Space Error* (SSE), the error measured in pixels. If the SSE does not exceed a required minimum, the tileset should not be rendered, and none of its tiles should be considered for rendering, see Geometric error.

root is an object that defines the root tile using the tile JSON described in the above section. root.geometricError is not the same as the tileset's top-level geometricError. The tileset's geometricError is used at runtime to determine the SSE at which the tileset's root tile renders; root.geometricError is used at runtime to determine the SSE at which the root tile's children are rendered.

### 6.7.2.1. External tilesets

To create a tree of trees, a tile's content.uri can point to an external tileset (the uri of another tileset JSON file). This enables, for example, storing each city in a tileset and then having a global tileset of tilesets.

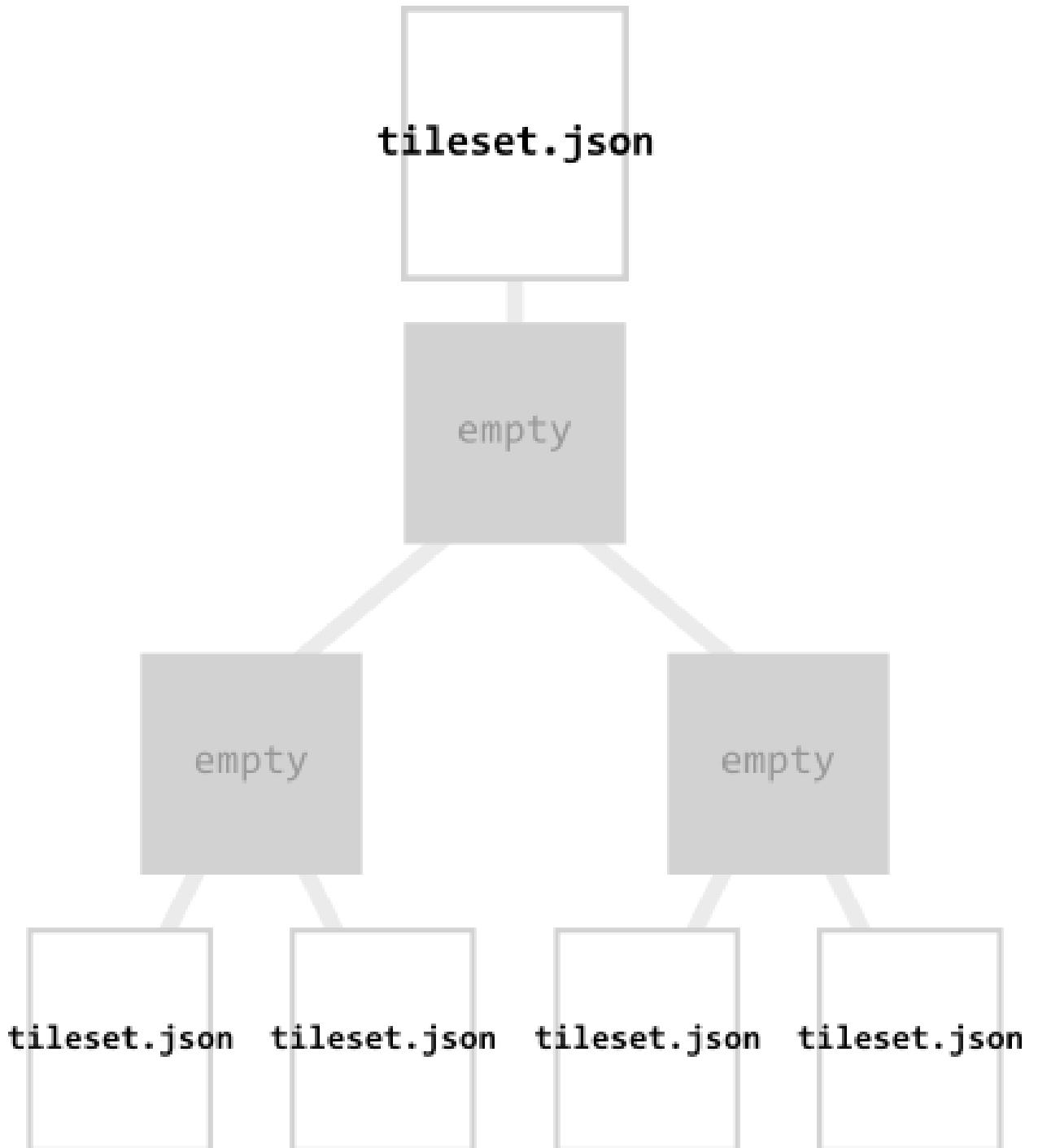
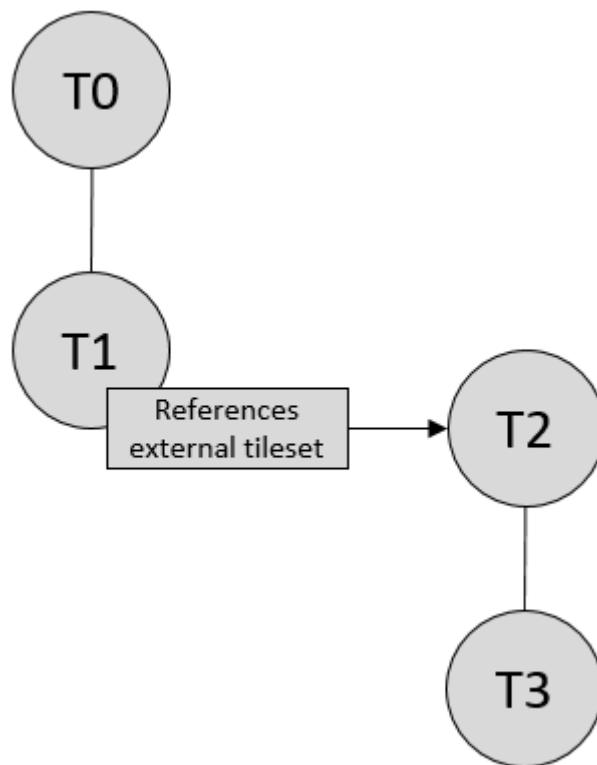


Figure 10 – A tileset that refers to other tilesets

When a tile points to an external tileset, the tile:

- Cannot have any children; `tile.children` shall be omitted
- Cannot be used to create cycles, for example, by pointing to the same tileset file containing the tile or by pointing to another tileset file that then points back to the initial file containing the tile.
- Will be transformed by both the tile's `transform` and root tile's `transform`. For example, in the following tileset referencing an external tileset, the computed transform for T3 is  $[T_0][T_1][T_2][T_3]$ .

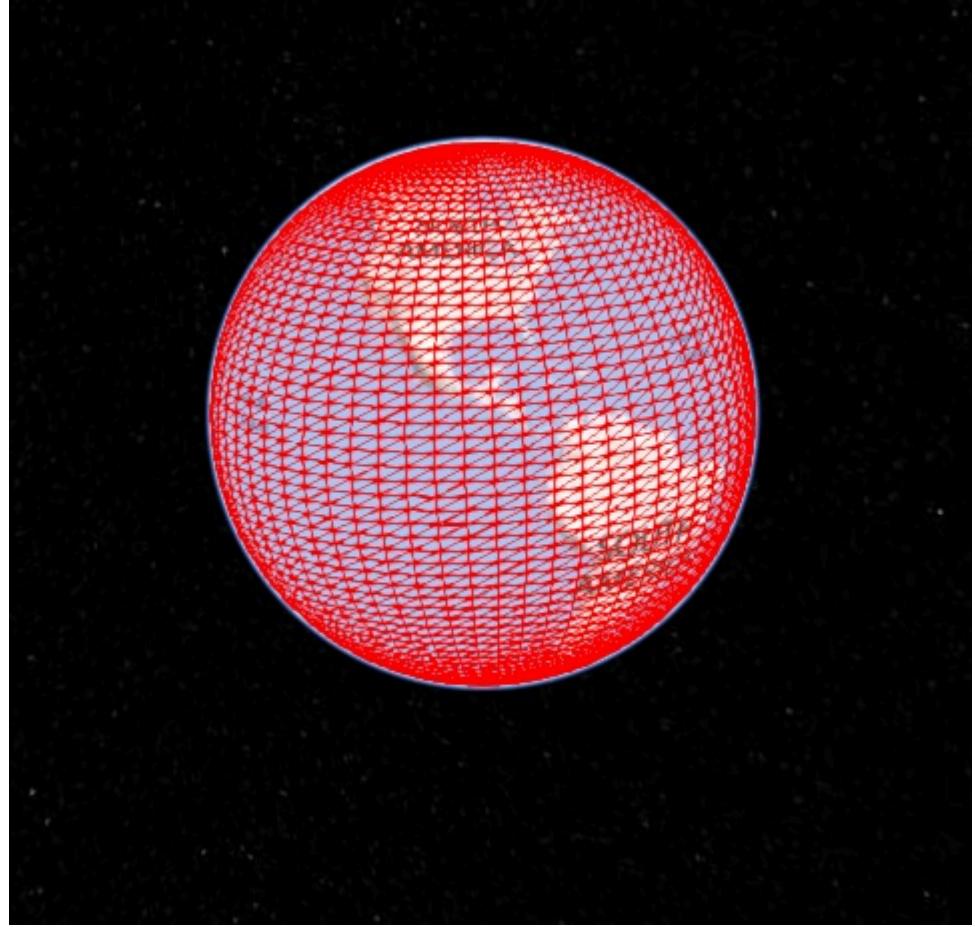


**Figure 11** – The chain of transforms for a tileset that refers to another tileset

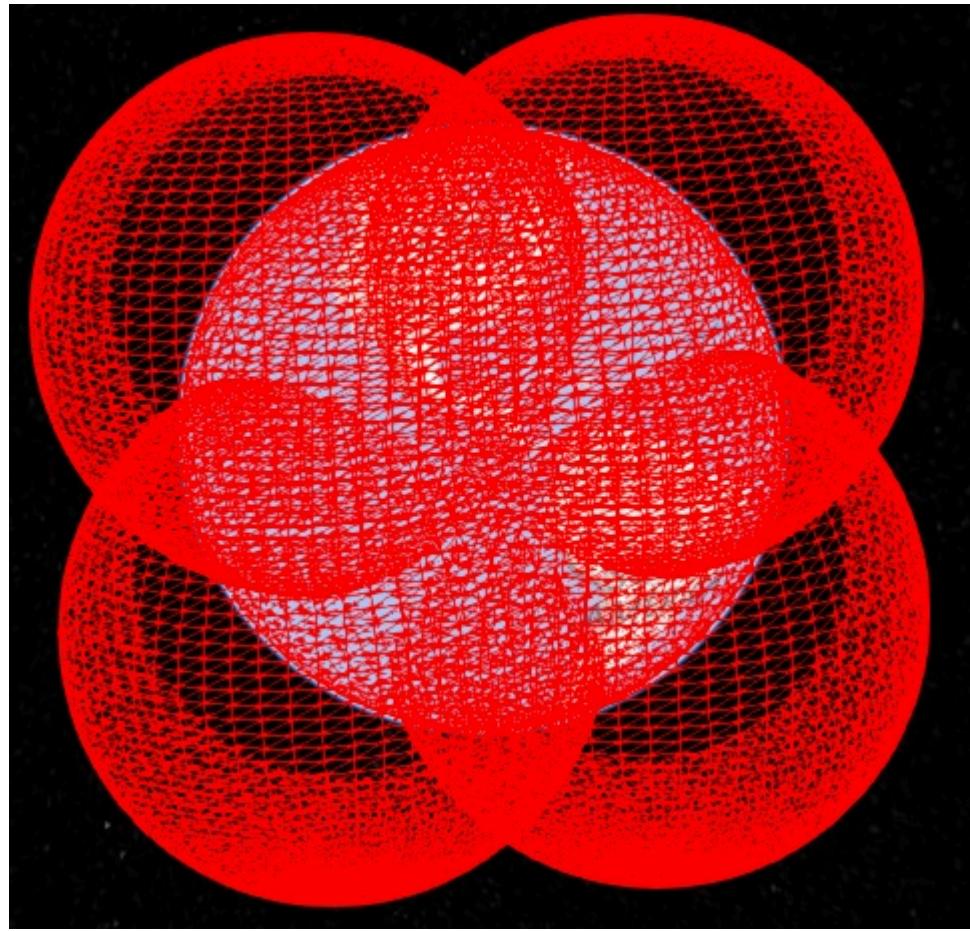
If an external tileset defines `asset.tilesetVersion`, this overrides the value from the parent tileset. If the external tileset does not define `asset.tilesetVersion`, the value is inherited from the parent tileset (if defined).

#### 6.7.2.2. Bounding volume spatial coherence

As described above, the tree has spatial coherence; each tile has a bounding volume completely enclosing its content, and the content for child tiles are completely inside the parent's bounding volume. This does not imply that a child's bounding volume is completely inside its parent's bounding volume. For example:



**Figure 12** – Bounding sphere for a terrain tile.



**Figure 13** – Bounding spheres for the four child tiles. The children’s content is completely inside the parent’s bounding volume, but the children’s bounding volumes are not since they are not tightly fit.

#### 6.7.2.3. Spatial data structures

3D Tiles incorporates the concept of Hierarchical Level of Detail (HLOD) for optimal rendering of spatial data. A tileset is composed of a tree, defined by root and, recursively, its children tiles, which can be organized by different types of spatial data structures.

A runtime engine is generic and will render any tree defined by a tileset. Any combination of tile formats and refinement approaches can be used, enabling flexibility in supporting heterogeneous datasets, see Refinement.

A tileset may use a 2D spatial tiling scheme similar to raster and vector tiling schemes (like a Web Map Tile Service (WMTS) or XYZ scheme) that serve predefined tiles at several levels of detail (or zoom levels). However since the content of a tileset is often non-uniform or may not easily be organized in only two dimensions, other spatial data structures may be more optimal.

Included below is a brief description of how 3D Tiles can represent various spatial data structures.

#### 6.7.2.3.1. Quadtrees

A quadtree is created when each tile has four uniformly subdivided children (e.g., using the center latitude and longitude), similar to typical 2D geospatial tiling schemes. Empty child tiles can be omitted.

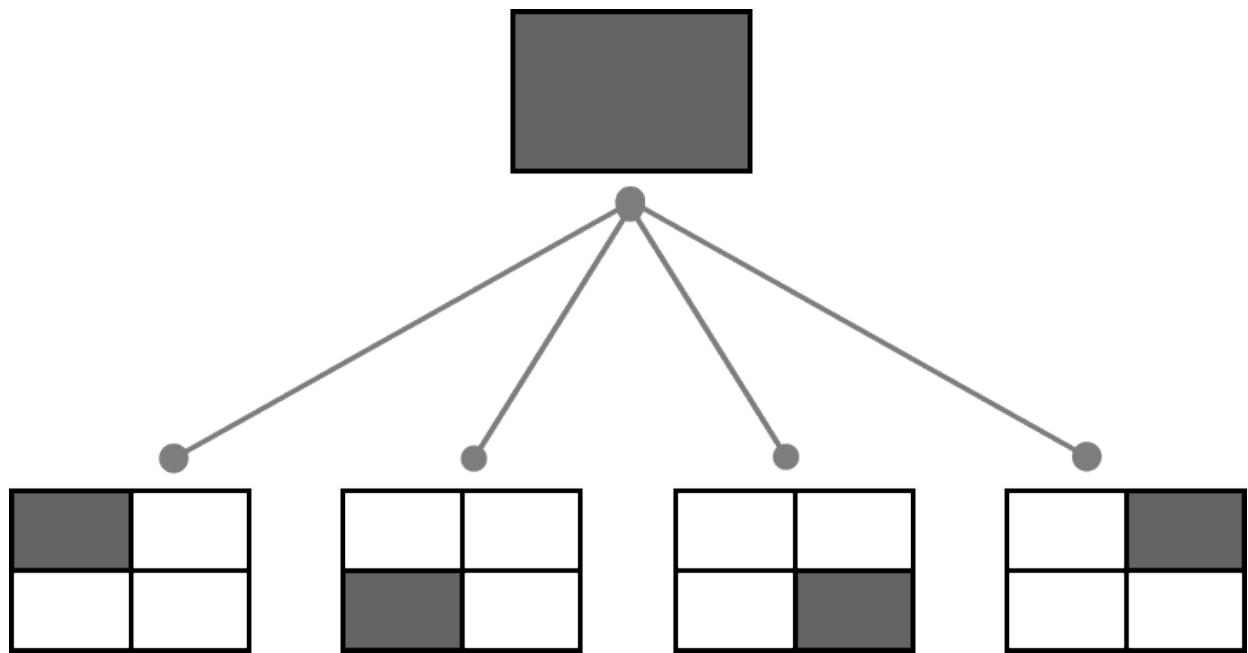
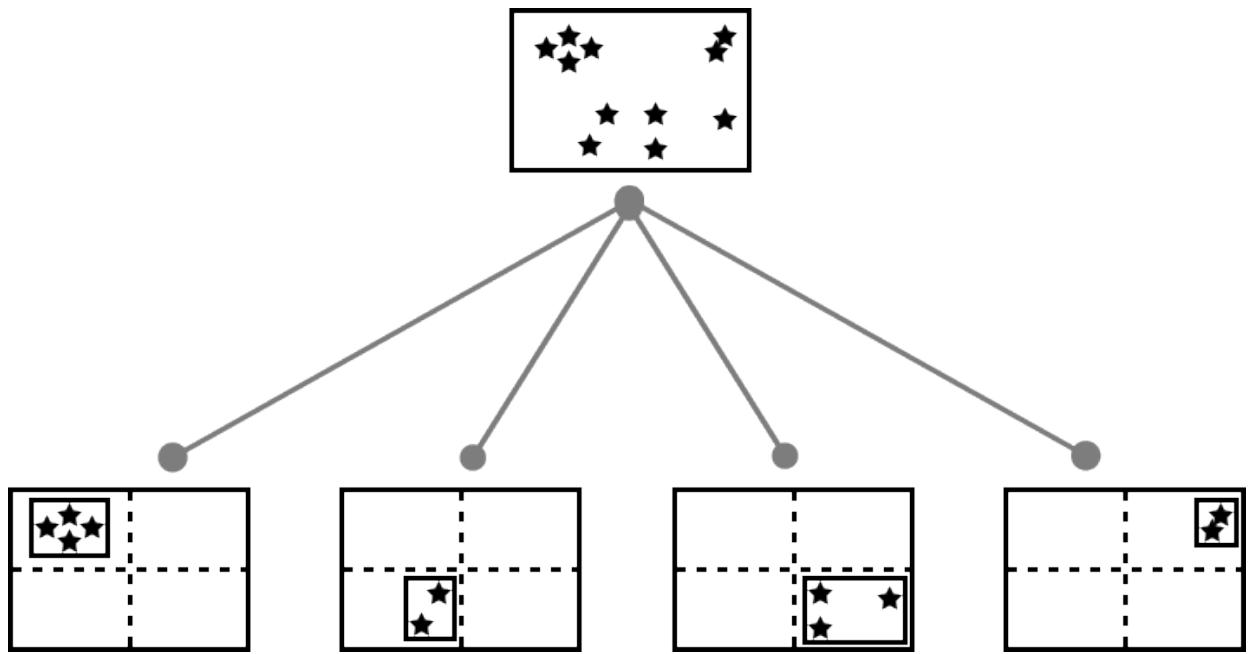


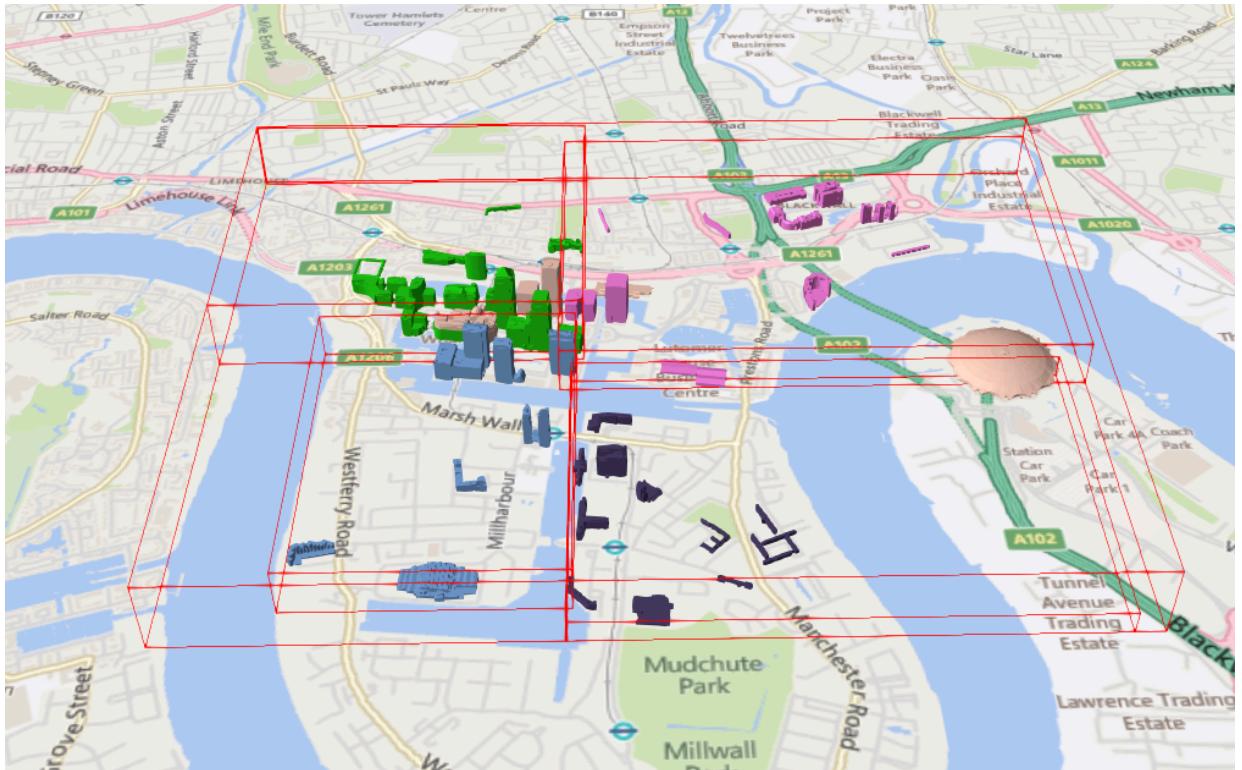
Figure 14 – Classic quadtree subdivision

3D Tiles enable quadtree variations such as non-uniform subdivision and tight bounding volumes (as opposed to bounding, for example, the full 25% of the parent tile, which is wasteful for sparse datasets).



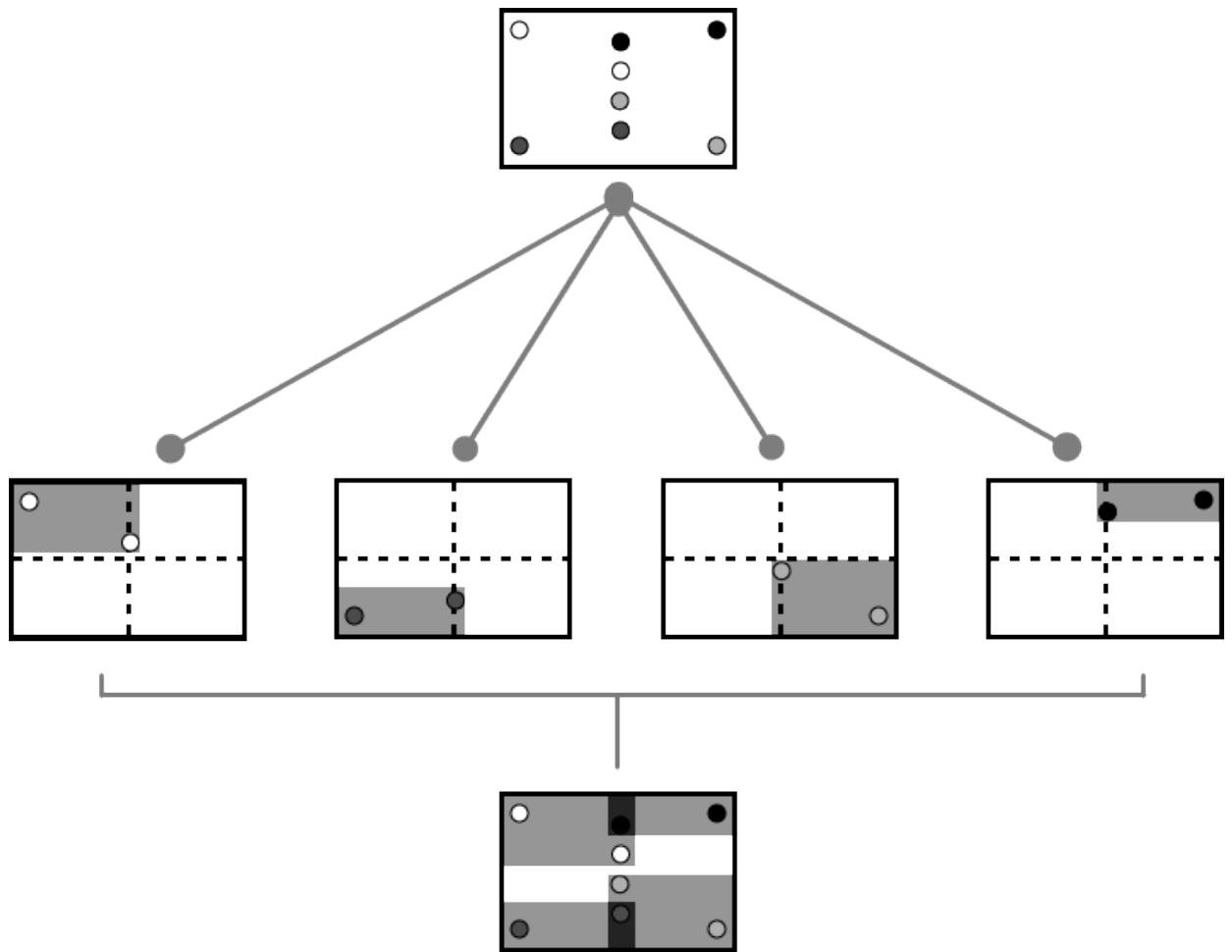
**Figure 15 – Quadtree with tight bounding volumes around each child**

For example, here is the root tile and its children for Canary Wharf. Note the bottom left, where the bounding volume does not include the water on the left where no buildings will appear:



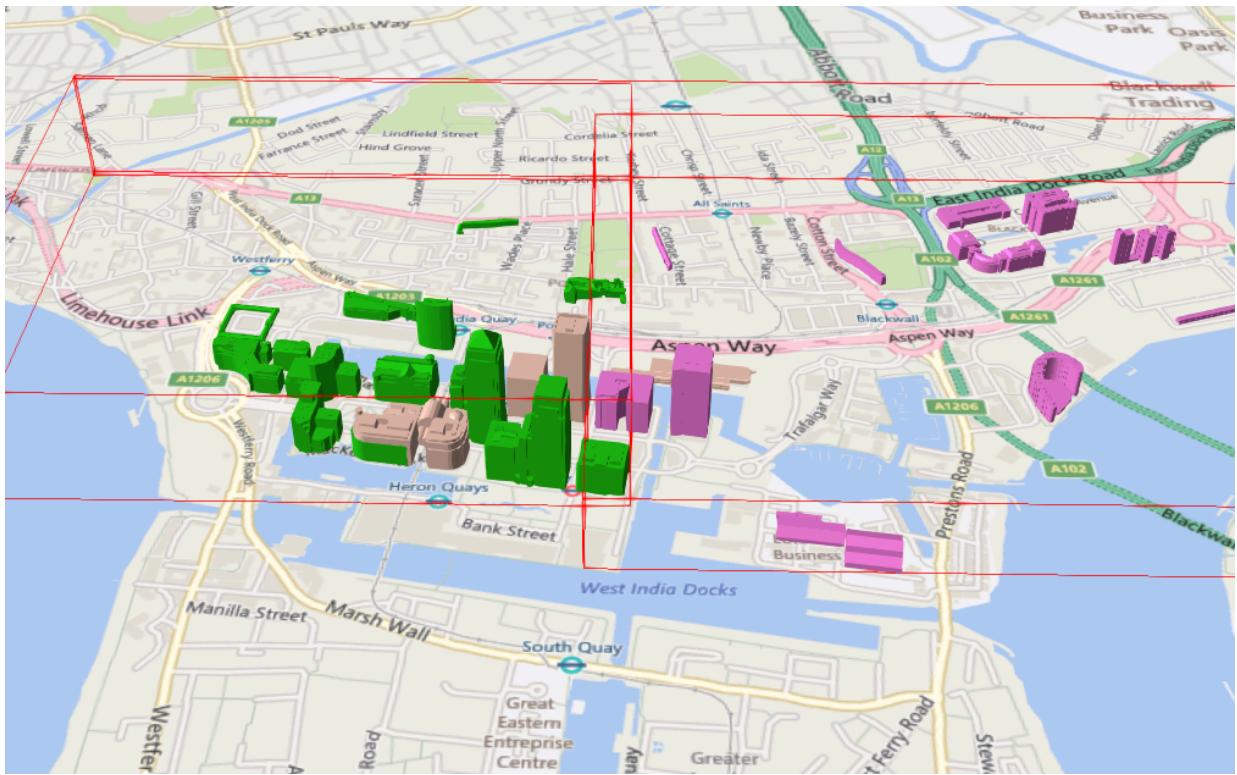
**Figure 16 – Building data from [CyberCity3D](#). Imagery data from [Bing Maps](#)**

3D Tiles also enable other quadtree variations such as [loose quadtrees](#), where child tiles overlap but spatial coherence is still preserved, i.e., a parent tile completely encloses all of its children. This approach can be useful to avoid splitting features, such as 3D models, across tiles.



**Figure 17 – Quadtree with non-uniform and overlapping tiles**

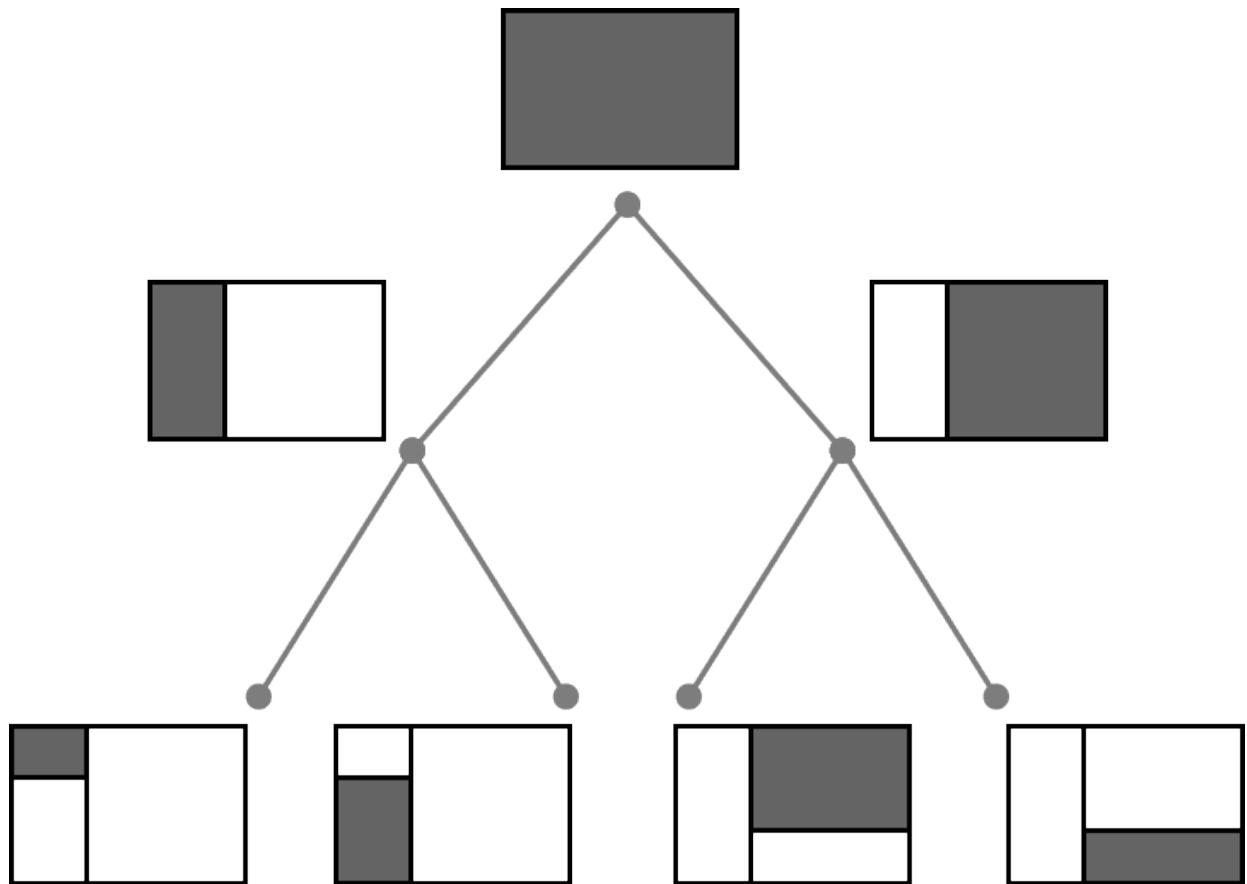
Below, the green buildings are in the left child and the purple buildings are in the right child. Note that the tiles overlap so the two green and one purple building in the center are not split.



**Figure 18** – Building data from [CyberCity3D](#). Imagery data from [Bing Maps](#)

#### 6.7.2.3.2. K-d trees

A k-d tree is created when each tile has two children separated by a *splitting plane* parallel to the x, y, or z axis (or latitude, longitude, height). The split axis is often round-robin rotated as levels increase down the tree, and the splitting plane may be selected using the median split, surface area heuristics, or other approaches.



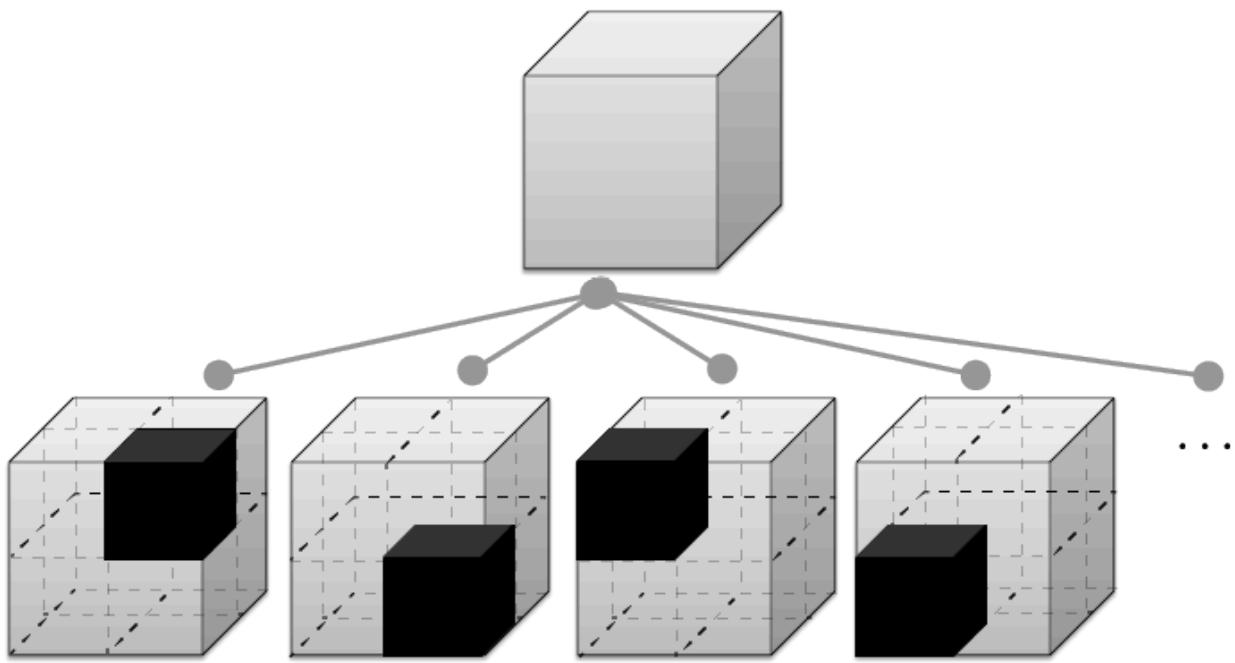
**Figure 19** – Example k-d tree. Note the non-uniform subdivision

Note that a k-d tree does not have uniform subdivision like typical 2D geospatial tiling schemes and, therefore, can create a more balanced tree for sparse and non-uniformly distributed datasets.

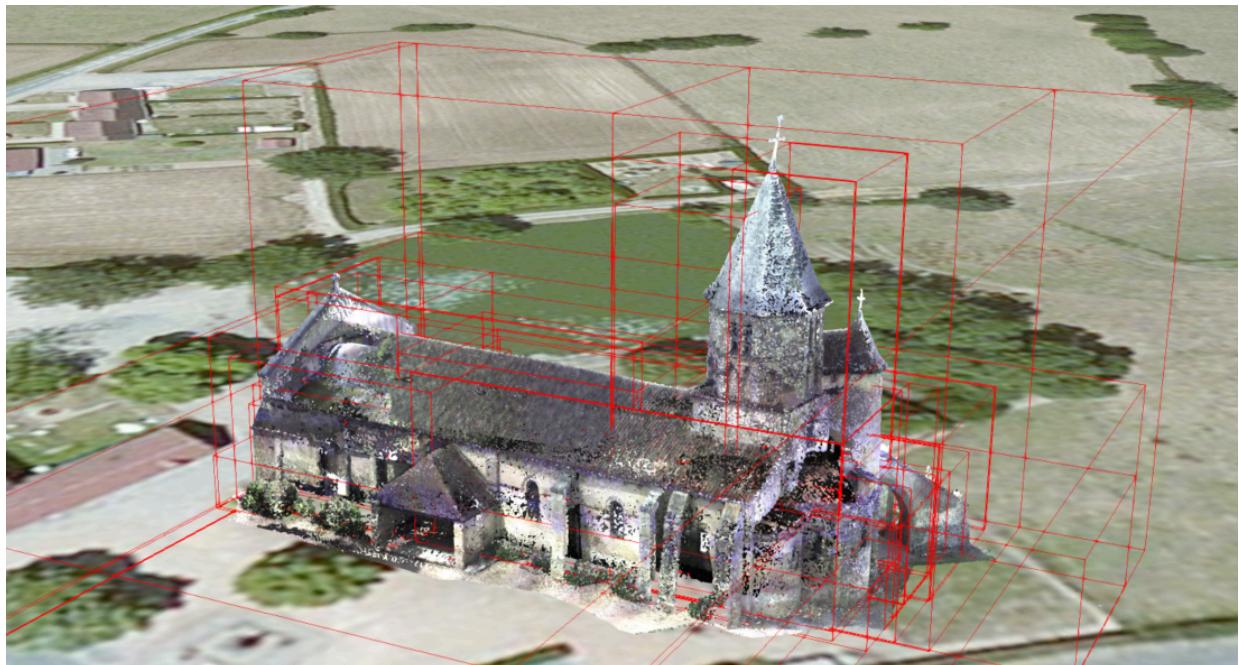
3D Tiles enables variations on k-d trees such as multi-way k-d trees where, at each leaf of the tree, there are multiple splits along an axis. Instead of having two children per tile, there are n children.

#### 6.7.2.3.3. Octrees

An octree extends a quadtree by using three orthogonal splitting planes to subdivide a tile into eight children. Like quadtrees, 3D Tiles allows variations to octrees such as non-uniform subdivision, tight bounding volumes, and overlapping children.



**Figure 20 – Traditional octree subdivision**



**Figure 21 – Non-uniform octree subdivision for a point cloud using additive refinement.**

Point Cloud of the [Church of St Marie at Chappes, France](#) by Prof. Peter Allen,  
Columbia University Robotics Lab. Scanning by Alejandro Troccoli and Matei Ciocarlie.

#### 6.7.2.3.4. Grids

3D Tiles enables uniform, non-uniform, and overlapping grids by supporting an arbitrary number of child tiles. For example, here is a top-down view of a non-uniform overlapping grid of Cambridge:

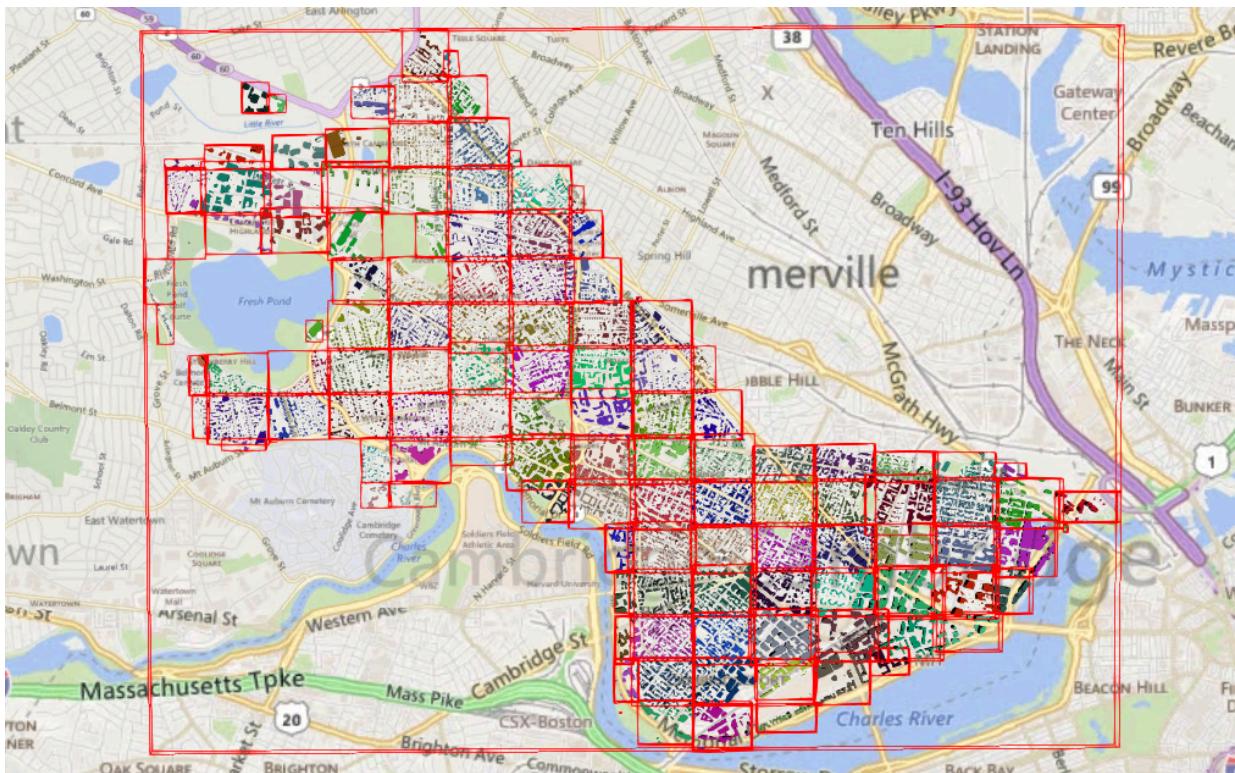
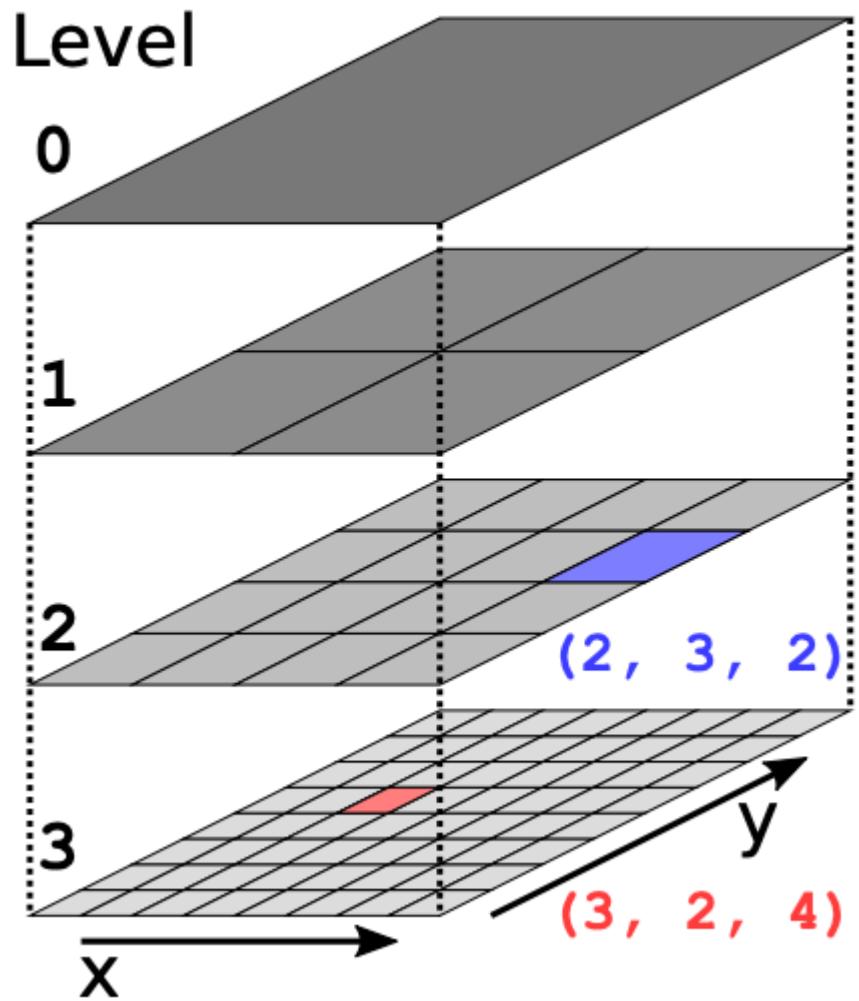


Figure 22 – Building data from [CyberCity3D](#). Imagery data from [Bing Maps](#)

3D Tiles takes advantage of empty tiles: those tiles that have a bounding volume, but no content. Since a tile's content property does not need to be defined, empty non-leaf tiles can be used to accelerate non-uniform grids with hierarchical culling. This essentially creates a quadtree or octree without hierarchical levels of detail (HLOD).

#### 6.7.2.4. Implicit Tiling

The bounding volume hierarchy may be defined *explicitly* – as shown previously – which enables a wide variety of spatial data structures. Certain common data structures such as quadtrees and octrees may be defined *implicitly* without providing bounding volumes for every tile. This regular pattern allows for random access of tiles based on their tile coordinates which enables accelerated spatial queries, new traversal algorithms, and efficient updates of tile content, among other use cases.



**Figure 23 – Quadtree with tile coordinates**

In order to support sparse datasets, availability data determines which tiles exist. To support massive datasets, availability is partitioned into fixed-size subtrees. Subtrees may store metadata for available tiles and contents.

An `implicitTiling` object may be added to any tile in the tileset JSON. The object defines how the tile is subdivided and where to locate content resources. It may be added to multiple tiles to create more complex subdivision schemes.

The following example shows a quadtree defined on the root tile, with template URLs pointing to content and subtree files.

```
{
  "root": {
    "boundingVolume": {
      "region": [-1.318, 0.697, -1.319, 0.698, 0, 20]
    },
    "refine": "REPLACE",
    "geometricError": 5000,
  }
}
```

```

    "content": {
      "uri": "content/{level}/{x}/{y}.glb"
    },
    "implicitTiling": {
      "subdivisionScheme": "QUADTREE",
      "availableLevels": 21,
      "subtreeLevels": 7,
      "subtrees": [
        "uri": "subtrees/{level}/{x}/{y}.json"
      ]
    }
  }
}

```

See [Implicit Tiling](#) for more details about the `implicitTiling` object structure and the subtree file format.

### 6.7.3. Metadata

Application-specific *metadata* may be provided at multiple granularities within a tileset. Metadata may be associated with high-level entities like tilesets, tiles, contents, or features, or with individual vertices and texels. Metadata conforms to a well-defined type system described by the 3D Metadata Specification, which may be extended with application- or domain-specific semantics.

Metadata enables additional use cases and functionality for the format:

- **Inspection:** Applications displaying a tileset within a user interface (UI) may allow users to click or hover over specific tiles or tile contents, showing informative metadata about a selected entity in the UI.
- **Collections:** Tile content groups may be used to define collections (similar to map layers), such that each collection may be shown, hidden, or visually styled with effects synchronized across many tiles.
- **Structured Data:** Metadata supports both embedded and externally-referenced schemas, such that tileset authors may define new data models for common domains (e.g. for AEC or scientific datasets) or fully customized, application-specific data (e.g. for a particular video game).
- **Optimization:** Per-content metadata may include properties with performance-related semantics, enabling engines to optimize traversal and streaming algorithms significantly.

The metadata can be associated with elements of a tileset at various levels of granularity:

- **Tileset** – The tileset as a whole may be associated with global metadata. Common examples might include year of collection, author details, or other general context for the tileset contents.
- **Tile** – Tiles may be individually associated with more specific metadata. This may be the timestamp when a tile was last updated or the maximum height of the tile, or spatial hints to optimize traversal algorithms.

- **Groups** – Tile contents may be organized into groups. Each group definition represents a metadata entity that can be assigned to the tile contents by specifying the index within this list as the group property of the content. This is useful for working with collections of contents as layers, e.g. to manage visibility or visual styling.
- **Content** – Tile contents may be individually associated with more specific metadata, such as a list of attribution strings.
- **Features** gITF 2.0 assets with feature metadata can be included as tile contents. The [EXT\\_structural\\_metadata](#) extension allows associating metadata with vertices or texels.

The figure below shows the relationship between these entities, and examples of metadata that may be associated with these entities:

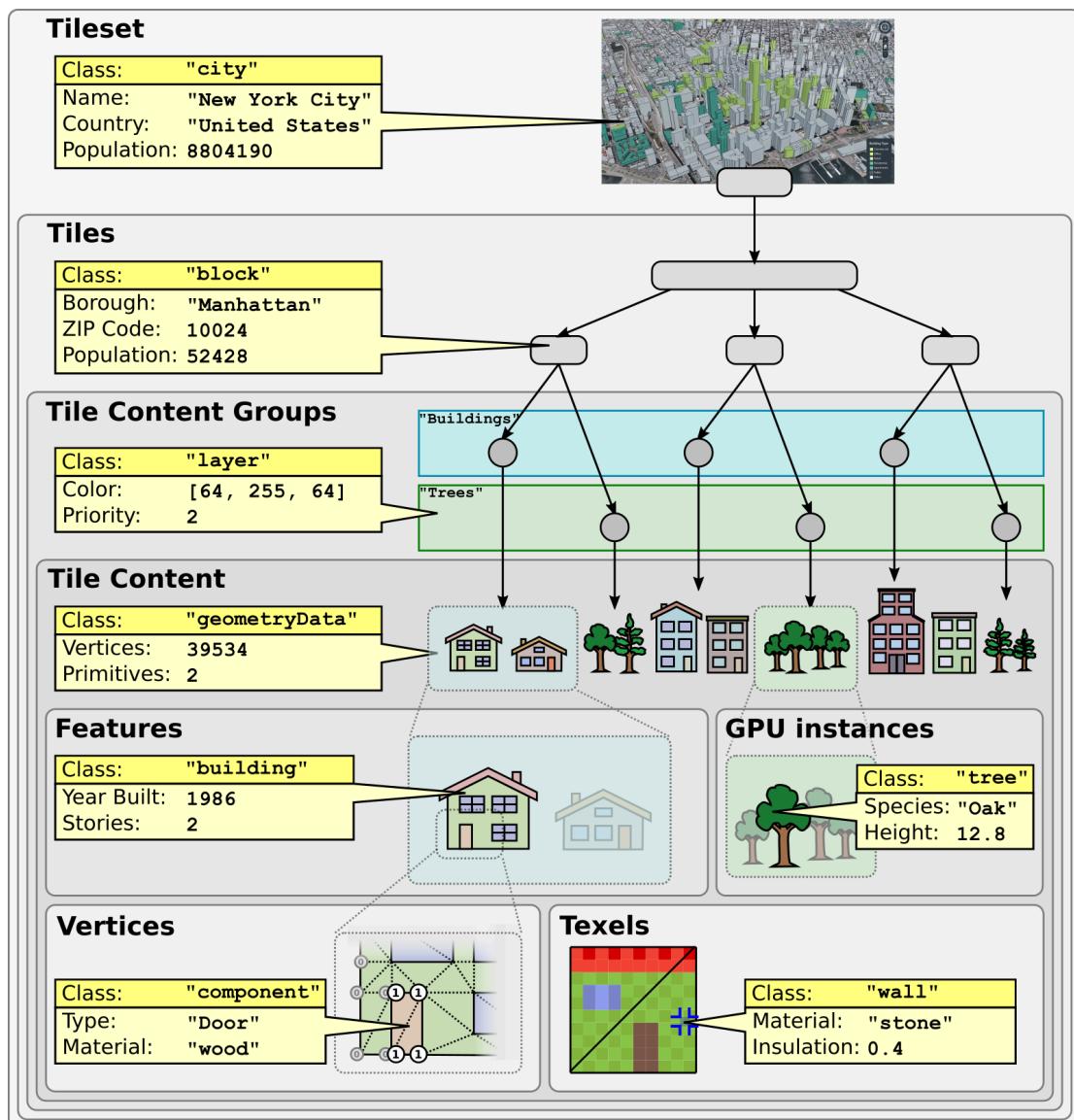


Figure 24 – Illustration of the different granularity levels for applying metadata

Although they are defined independently, the metadata structure in 3D Tiles and in the glTF EXT\_structural\_metadata extension both conform to the 3D Metadata Specification and build upon the Reference Implementation of the 3D Metadata Specification. Concepts and terminology used here refer to the 3D Metadata Specification, which should be considered a normative reference for definitions and requirements. This document provides inline definitions of terms where appropriate.

### 6.7.3.1. Metadata Schema

The Metadata schema defines the structure of the metadata. It contains a definition of the metadata classes, which are templates for the metadata instances, and define the set of properties that each metadata instance has. The metadata schema is stored within a tileset in the form of a JSON representation according to the Metadata Schema Reference Implementation. This reference implementation includes the definition of the JSON schema for the metadata schema.

Schemas may be embedded in tilesets with the `schema` property, or referenced externally by the `schemaUri` property. Multiple tilesets and glTF contents may refer to the same schema to avoid duplication. Any class that is defined in the schema of an external tileset shall also be defined in the top-level tileset schema.

**NOTE 1** Schema with a building class having three properties, “height”, “owners”, and “buildingType”. The “buildingType” property refers to the `buildingType` enum as its data type, also defined in the schema. Later examples show how entities declare their class and supply values for their properties.

```
{
  "schema": {
    "classes": {
      "building": {
        "properties": {
          "height": {
            "type": "SCALAR",
            "componentType": "FLOAT32"
          },
          "owners": {
            "type": "STRING",
            "array": true,
            "description": "Names of owners."
          },
          "buildingType": {
            "type": "ENUM",
            "enumType": "buildingType"
          }
        }
      }
    },
    "enums": {
      "buildingType": {
        "values": [
          {"value": 0, "name": "Residential"},
          {"value": 1, "name": "Commercial"},
          {"value": 2, "name": "Other"}
        ]
      }
    }
  }
}
```

**NOTE 2**External schema referenced by a URI.

```
{  
    "schemaUri": "https://example.com/metadata/buildings/1.0/schema.json"  
}
```

### 6.7.3.2. Assigning Metadata

While classes within a schema define the data types and meanings of properties, properties do not take on particular values until a metadata is assigned (i.e. the class is “instantiated”) as a particular metadata entity within the 3D Tiles hierarchy.

The common structure of metadata entities that appear in a tileset is defined in `metadataEntity.schema.json`. Each metadata entity contains the name of the class that it is an instance of, as well as a dictionary of property values that correspond to the properties of that class. Each property value assigned shall be defined by a class property with the same property ID, with values matching the data type of the class property. An entity may provide values for only a subset of the properties of its class, but class properties marked required: `true` shall not be omitted.

**NOTE**A metadata entity for the building class presented earlier. Such an entity could be assigned to a tileset, a tile, or tile content, by storing it as their respective `metadata` property.

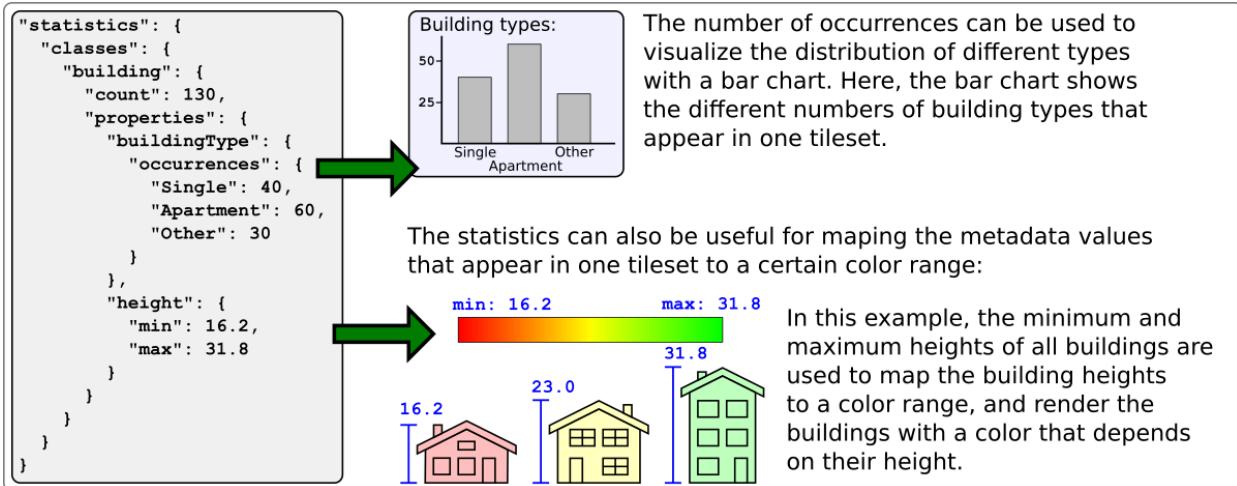
```
"metadata": {  
    "class": "building",  
    "properties": {  
        "height": 16.8,  
        "owners": [ "John Doe", "Jane Doe" ],  
        "buildingType": "Residential"  
    }  
}
```

Most property values are encoded as JSON within the entity. One notable exception is metadata assigned to implicit tiles and contents, stored in a more compact binary form. See Implicit Tiling.

### 6.7.3.3. Metadata Statistics

Statistics provide aggregate information about the distribution of property values, summarized over all instances of a metadata class within a tileset. For example, statistics may include the minimum/maximum values of a numeric property, or the number of occurrences for specific enum values.

These summary statistics allow applications to analyze or display metadata, e.g. with the declarative styling language, without first having to process the complete dataset to identify bounds for color ramps and histograms. Statistics are provided on a per-class basis, so that applications can provide styling or context based on the tileset as a whole, while only needing to download and process a subset of its tiles.



**Figure 25 – Illustration of how to use metadata statistics for rendering an analytics**

The statistics are stored in the top-level `statistics` object of the tileset. The structure of this `statistics` object is defined in `statistics.schema.json`. The statistics are defined for each metadata class, including the following elements:

- `count` is the number of entities of a class occurring within the tileset
- `properties` contains summary statistics about properties of a class occurring within the tileset

Properties may include the following built-in statistics:

**Table 5 – Statistics for metadata**

NAME	DESCRIPTION	TYPE
<code>min</code>	The minimum property value	Scalars, vector, matrices
<code>max</code>	The maximum property value	...
<code>mean</code>	The arithmetic mean of the property values	...
<code>median</code>	The median of the property values	...
<code>standardDeviation</code>	The standard deviation of the property values	...
<code>variance</code>	The variance of the property values	...
<code>sum</code>	The sum of the property values	...
<code>occurrences</code>	Frequencies of value occurrences	Object in which keys are property values (for enums, the

NAME	DESCRIPTION	TYPE
		enum name), and values are the number of occurrences of that property value

Tileset authors may define their own additional statistics, like `_mode` in the example below. Application-specific statistics should use an underscore prefix (`_*`) and lowerCamelCase for consistency and to avoid conflicting with future built-in statistics.

**NOTE** Definition of a “building” class, with three properties. Summary statistics provide a minimum, maximum, and (application-specific) “`_mode`” for the numerical “height” property. The enum “`buildingType`” property is summarized by the number of distinct enum value occurrences.

```
{
  "schema": {
    "classes": {
      "building": {
        "properties": {
          "height": {
            "type": "SCALAR",
            "componentType": "FLOAT32"
          },
          "owners": {
            "type": "STRING",
            "array": true
          },
          "buildingType": {
            "type": "ENUM",
            "enumType": "buildingType"
          }
        }
      }
    },
    "enums": {
      "buildingType": {
        "valueType": "UINT16",
        "values": [
          {"name": "Residential", "value": 0},
          {"name": "Commercial", "value": 1},
          {"name": "Hospital", "value": 2},
          {"name": "Other", "value": 3}
        ]
      }
    }
  },
  "statistics": {
    "classes": {
      "building": {
        "count": 100000,
        "properties": {
          "height": {
            "min": 3.9,
            "max": 341.7,
            "_mode": 5.0
          },
          "buildingType": {
            "occurrences": {
              "Residential": 50000,
              "Commercial": 40950,
              "Hospital": 50
            }
          }
        }
      }
    }
  }
}
```

```
        }
    }
```

## 6.7.4. Specifying extensions and application specific extras

3D Tiles defines extensions to allow the base specification to have extensibility for new features.

### 6.7.4.1. Extensions

Extensions allow the base specification to be extended with new features. The optional extensions dictionary property may be added to any 3D Tiles JSON object, which contains the name of the extensions and the extension specific objects. The following example shows a tile object with a hypothetical vendor extension which specifies a separate collision volume.

```
{
  "transform": [
    4.843178171884396, 1.2424271388626869, 0, 0,
    -0.7993325488216595, 3.1159251367235608, 3.8278032889280675, 0,
    0.9511533376784163, -3.7077466670407433, 3.2168186118075526, 0,
    1215001.7612985559, -4736269.697480114, 4081650.708604793, 1
  ],
  "boundingVolume": {
    "box": [
      0, 0, 6.701,
      3.738, 0, 0,
      0, 3.72, 0,
      0, 0, 13.402
    ]
  },
  "geometricError": 32,
  "content": {
    "uri": "building.glb"
  },
  "extensions": {
    "VENDOR_collision_volume": {
      "box": [
        0, 0, 6.8,
        3.8, 0, 0,
        0, 3.8, 0,
        0, 0, 13.5
      ]
    }
  }
}
```

All extensions used in a tileset or any descendant external tilesets shall be listed in the entry tileset JSON in the top-level extensionsUsed array property, e.g.,

```
{
  "extensionsUsed": [
    "VENDOR_collision_volume"
  ]
}
```

All extensions required to load and render a tileset or any descendant external tilesets shall also be listed in the entry tileset JSON in the top-level extensionsRequired array property, such that extensionsRequired is a subset of extensionsUsed. All values in extensionsRequired shall also exist in extensionsUsed.

#### 6.7.4.2. Extras

The extras property allows application specific metadata to be added to any 3D Tiles JSON object. The following example shows a tile object with an additional application specific name property.

```
{  
  "transform": [  
    4.843178171884396, 1.2424271388626869, 0, 0,  
    -0.7993325488216595, 3.1159251367235608, 3.8278032889280675, 0,  
    0.9511533376784163, -3.7077466670407433, 3.2168186118075526, 0,  
    1215001.7612985559, -4736269.697480114, 4081650.708604793, 1  
,  
  "boundingVolume": {  
    "box": [  
      0, 0, 6.701,  
      3.738, 0, 0,  
      0, 3.72, 0,  
      0, 0, 13.402  
    ]  
  },  
  "geometricError": 32,  
  "content": {  
    "uri": "building.glb"  
  },  
  "extras": {  
    "name": "Empire State Building"  
  }  
}
```

The full JSON schema can be found in `tileset.schema.json`.

## 6.8. Tile format specifications

---

Each tile's content.uri property is a uri to a file containing information for rendering the tile's 3D content. The content is an instance of one of the formats listed below.

[glTF 2.0](#) is the primary tile format for 3D Tiles. glTF is an open specification designed for the efficient transmission and loading of 3D content. A glTF asset includes geometry and texture information for a single tile, and may be extended to include metadata, model instancing, and compression. glTF may be used for a wide variety of 3D content including:

- Heterogeneous 3D models. E.g. textured terrain and surfaces, 3D building exteriors and interiors, massive models
- 3D model instances. E.g. trees, windmills, bolts

- Massive point clouds

See glTF Tile Format for more details.

Tiles may also reference the legacy 3D Tiles 1.0 formats listed below. These formats were deprecated in 3D Tiles 1.1 and may be removed in a future version of 3D Tiles.

**Table 6 — Legacy tile formats and common uses**

LEGACY FORMAT	USES
Batched 3D Model (b3dm)	Heterogeneous 3D models
Instanced 3D Model (i3dm)	3D model instances
Point Cloud (pnts)	Massive number of points
Composite (cmpt)	Concatenate tiles of different formats into one tile

## 6.9. Declarative styling specification

---

3D Tiles includes concise declarative styling defined with JSON and expressions written in a small subset of JavaScript augmented for styling.

Styles define how a feature is displayed, for example `show` and `color` (RGB and translucency), using an expression based on a feature's properties.

The following example colors features with a height above 90 as red and the others as white.

```
{
  "color" : "(${Height} > 90) ? color('red') : color('white')"
}
```

For complete details, see the Declarative Styling specification.

7

# TILE FORMATS

---

# TILE FORMATS

---

The tile content in a 3D Tiles tileset represents the renderable content of a tile. It is referred to with the `content.uri` of the tile JSON.

The primary tile format in 3D Tiles 1.1 is the glTF Tile Format. It is built on [glTF 2.0](#) and allows modeling many different use cases and different forms of renderable content in 3D Tiles.

## 7.1. Legacy Tile Formats

---

The following tile formats have been part of 3D Tiles 1.0, and have been superseded by the glTF Tile Format.

**Table 7 – Legacy tile formats and common uses**

LEGACY FORMAT	USES
Batched 3D Model (b3dm)	Heterogeneous 3D models
Instanced 3D Model (i3dm)	3D model instances
Point Cloud (pnts)	Massive number of points
Composite (cmp t)	Concatenate tiles of different formats into one tile

See the migration guide for further information about how these use cases can be modeled based on the glTF tile format.

## 7.2. glTF

---

### 7.2.1. Overview

[glTF 2.0](#) is the primary tile format for 3D Tiles. glTF is an open specification designed for the efficient transmission and loading of 3D content. A glTF asset includes geometry and texture information for a single tile, and may be extended to include metadata, model instancing, and compression. glTF may be used for a wide variety of 3D content including:

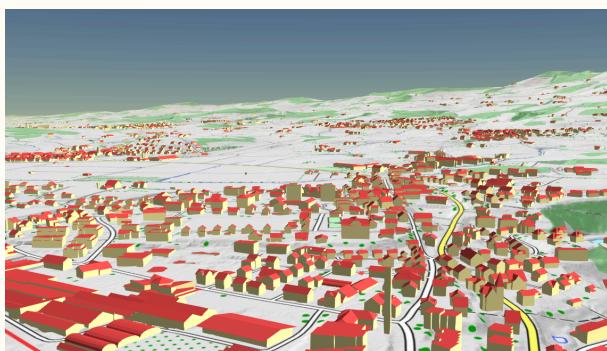
- Heterogeneous 3D models. E.g. textured terrain and surfaces, 3D building exteriors and interiors, massive models
- Massive point clouds
- 3D model instances. E.g. trees, windmills, bolts

## 7.2.2. Use Cases

### 7.2.2.1. 3D Models

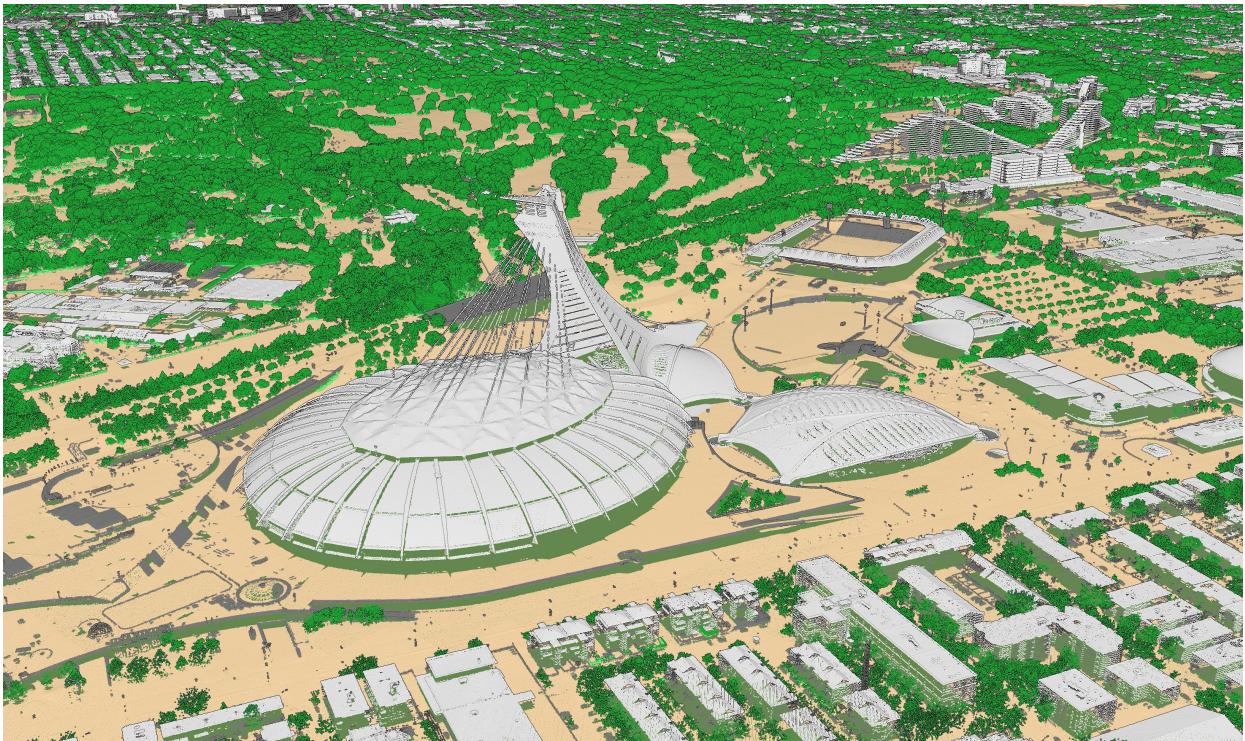
glTF provides flexibility for a wide range of mesh and material configurations, and is well suited for photogrammetry as well as stylized 3D models.

**Table 8** – Different types of models based on glTF

PHOTOGRAMMETRY	3D BUILDINGS
 <p>San Francisco photogrammetry model from Aerometrex in O3DE</p>	 <p>3D buildings from swisstopo in CesiumJS</p>

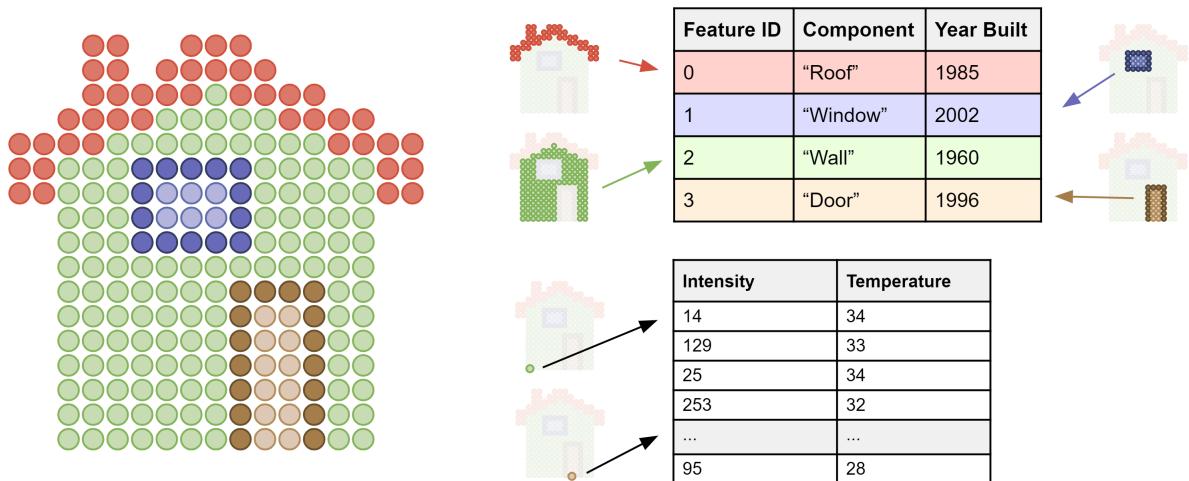
### 7.2.2.2. Point Clouds

glTF supports point clouds with the `0 (POINTS)` primitive mode. Points can have positions, colors, normals, and custom attributes as specified in the `primitive.attributes` field.



**Figure 26** – 40 billion point cloud from the City of Montreal with ASPRS classification codes ([CC-BY 4.0](#))

When using the `EXT_mesh_features` extension points can be assigned feature IDs and these features can have associated metadata.



**Figure 27** – A point cloud with two property tables, one storing metadata for groups of points and the other storing metadata for individual points

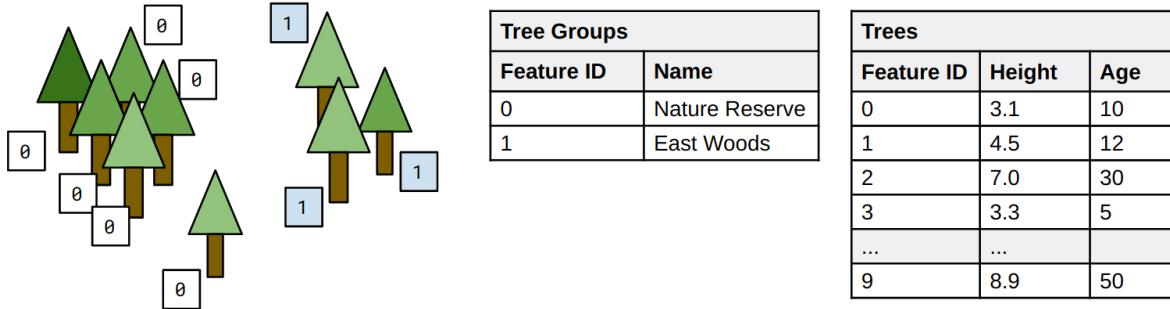
### 7.2.2.3. Instancing

glTF can leverage GPU instancing with the `EXT_mesh_gpu_instancing` extension. Instances can be given unique translations, rotations, scales, and other per-instance attributes.



**Figure 28 – Instanced tree models in Philadelphia from OpenTreeMap**

When using the `EXT_instance_features` extension instances can be assigned feature IDs and these features can have associated metadata.



**Figure 29** – Instanced tree models where trees are assigned to groups with a per-instance feature ID attribute. One feature table stores per-group metadata and the other stores per-tree metadata

### 7.2.3. Feature Identification

`EXT_mesh_features` provides a means of assigning identifiers to geometry and subcomponents of geometry within a glTF 2.0 asset. Feature IDs can be assigned to vertices or texels. `EXT_instance_features` allows feature IDs to be assigned to individual instances.



**Figure 30** – Street level photogrammetry of San Francisco Ferry Building from Aerometrex. Left: per-texel colors showing the feature classification, e.g., roof, sky roof, windows, window frames, and AC units . Right: classification is used to determine rendering material properties, e.g., make the windows translucent

## 7.2.4. Metadata

EXT\_structural\_metadata stores metadata at per-vertex, per-texel, and per-feature granularity and uses the type system defined in the 3D Metadata Specification. This metadata can be used for visualization and analysis.

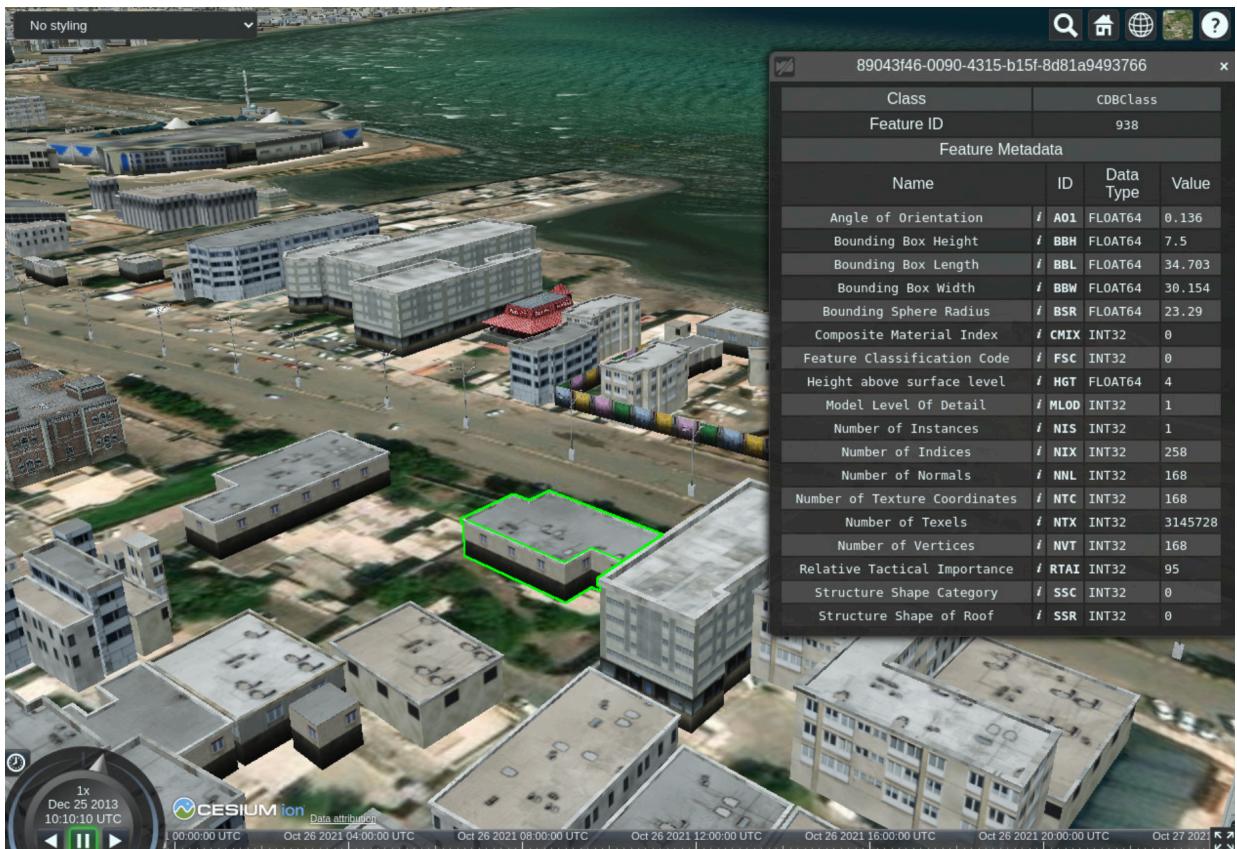


Figure 31 – An example of metadata associated with glTF models

## 7.2.5. Compression

glTF has several extensions for geometry and texture compression. These extensions can help reduce file sizes and GPU memory usage.

### 7.2.5.1. Geometry Compression

- KHR\_mesh\_quantization
- EXT\_meshopt\_compression
- KHR\_draco\_mesh\_compression

### 7.2.5.2. Texture Compression

- [KHR\\_texture\\_basisu](#)

## 7.2.6. File Extensions and Media Types

See [glTF File Extensions and Media Types](#).

An explicit file extension is optional. Valid implementations may ignore it and identify a content's format through other means, such as the `magic` field in the binary glTF header or the presence of an `asset` field in JSON glTF.

## 7.3. Batch Table

---

### WARNING

*Batch Table was deprecated in 3D Tiles 1.1. See [glTF migration guide](#).*

### 7.3.1. Overview

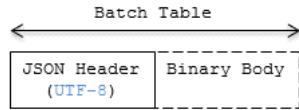
A *Batch Table* is a component of a tile's binary body and contains per-feature application-specific properties in a tile. These properties are queried at runtime for declarative styling and any application-specific use cases such as populating a UI or issuing a REST API request. Some example Batch Table properties are building heights, geographic coordinates, and database primary keys.

A Batch Table is used by the following tile formats:

- Batched 3D Model (b3dm)
- Instanced 3D Model (i3dm)
- Point Cloud (pnts)

### 7.3.2. Layout

A Batch Table is composed of two parts: a JSON header and an optional binary body in little endian. The JSON describes the properties, whose values either can be defined directly in the JSON as an array, or can refer to sections in the binary body. It is more efficient to store long numeric arrays in the binary body. The following figure shows the Batch Table layout:



**Figure 32 – Data layout of a Batch Table**

When a tile format includes a Batch Table, the Batch Table immediately follows the tile's Feature Table. The header will also contain batchTableJSONByteLength and batchTableBinaryByteLength uint32 fields, which can be used to extract each respective part of the Batch Table.

### 7.3.2.1. Padding

The JSON header shall end on an 8-byte boundary within the containing tile binary. The JSON header shall be padded with trailing Space characters (0x20) to satisfy this requirement.

The binary body shall start and end on an 8-byte boundary within the containing tile binary. The binary body shall be padded with additional bytes, of any value, to satisfy this requirement.

Binary properties shall start at a byte offset that is a multiple of the size in bytes of the property's componentType. For example, a property with the component type FLOAT has 4 bytes per element, and therefore shall start at an offset that is a multiple of 4. Preceding binary properties shall be padded with additional bytes, of any value, to satisfy this requirement.

### 7.3.2.2. JSON header

Batch Table values can be represented in the JSON header in two different ways:

1. An array of values, e.g., "name" : [ 'name1', 'name2', 'name3' ] or "height" : [10.0, 20.0, 15.0].
  - Array elements can be any valid JSON data type, including objects and arrays. Elements may be null.
  - The length of each array is equal to batchLength, which is specified in each tile format. This is the number of features in the tile. For example, batchLength may be the number of models in a b3dm tile, the number of instances in a i3dm tile, or the number of points (or number of objects) in a pnts tile.
2. A reference to data in the binary body, denoted by an object with byteOffset, componentType, and type properties, e.g., "height" : { "byteOffset" : 24, "componentType" : "FLOAT", "type" : "SCALAR" }.
  - byteOffset specifies a zero-based offset relative to the start of the binary body. The value of byteOffset shall be a multiple of the size in bytes of the

property's componentType, e.g., a property with the component type FLOAT shall have a byteOffset value that is a multiple of 4.

- componentType is the datatype of components in the attribute. Allowed values are "BYTE", "UNSIGNED\_BYTE", "SHORT", "UNSIGNED\_SHORT", "INT", "UNSIGNED\_INT", "FLOAT", and "DOUBLE".
- type specifies if the property is a scalar or vector. Allowed values are "SCALAR", "VEC2", "VEC3", and "VEC4".

The Batch Table JSON is a UTF-8 string containing JSON.

**NOTE**In JavaScript, the Batch Table JSON can be extracted from an ArrayBuffer using the TextDecoder JavaScript API and transformed to a JavaScript object with `JSON.parse`.

A batchId is used to access elements in each array and extract the corresponding properties. For example, the following Batch Table has properties for a batch of two features:

```
{  
  "id" : ["unique id", "another unique id"],  
  "displayName" : ["Building name", "Another building name"],  
  "yearBuilt" : [1999, 2015],  
  "address" : [{"street" : "Main Street", "houseNumber" : "1"}, {"street" : "Main Street", "houseNumber" : "2"}]  
}
```

The properties for the feature with batchId = 0 are

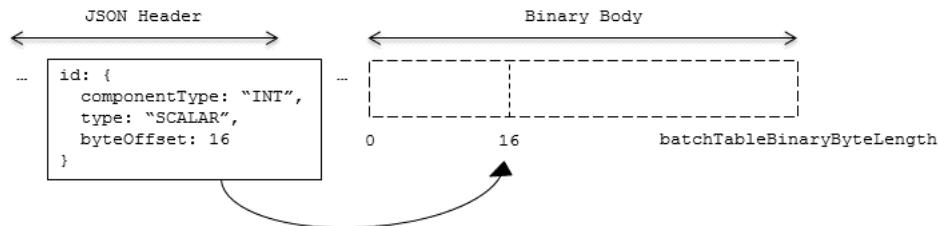
```
id[0] = 'unique id';  
displayName[0] = 'Building name';  
yearBuilt[0] = 1999;  
address[0] = {street : 'Main Street', houseNumber : '1'};
```

The properties for batchId = 1 are

```
id[1] = 'another unique id';  
displayName[1] = 'Another building name';  
yearBuilt[1] = 2015;  
address[1] = {street : 'Main Street', houseNumber : '2'};
```

### 7.3.2.3. Binary body

When the JSON header includes a reference to the binary section, the provided byteOffset is used to index into the data, as shown in the following figure:



**Figure 33** – An example showing how to access the binary body, based on the information from the JSON header

Values can be retrieved using the number of features, batchLength; the desired batch id, batchSize; and the componentType and type defined in the JSON header.

The following tables can be used to compute the byte size of a property.

**Table 9** – Sizes of component types

componentType	SIZE IN BYTES
"BYTE"	1
"UNSIGNED_BYTE"	1
"SHORT"	2
"UNSIGNED_SHORT"	2
"INT"	4
"UNSIGNED_INT"	4
"FLOAT"	4
"DOUBLE"	8

**Table 10** – Numbers of components for elements

type	NUMBER OF COMPONENTS
"SCALAR"	1
"VEC2"	2

type	NUMBER OF COMPONENTS
"VEC3"	3
"VEC4"	4

### 7.3.3. Extensions

The following extensions can be applied to a Batch Table.

- `3DTILES_batch_table_hierarchy`

### 7.3.4. Implementation example

*This section is informative*

The following examples access the "height" and "geographic" values respectively given the following Batch Table JSON with batchLength of 10:

```
{
  "height" : {
    "byteOffset" : 0,
    "componentType" : "FLOAT",
    "type" : "SCALAR"
  },
  "geographic" : {
    "byteOffset" : 40,
    "componentType" : "DOUBLE",
    "type" : "VEC3"
  }
}
```

To get the "height" values:

```
var height = batchTableJSON.height;
var byteOffset = height.byteOffset;
var componentType = height.componentType;
var type = height.type;

var heightArrayByteLength = batchLength * sizeInBytes(componentType) *
numberComponents(type); // 10 * 4 * 1
var heightArray = new Float32Array(batchTableBinary.buffer, byteOffset,
heightArrayByteLength);
var heightOfFeature = heightArray[batchId];
```

To get the "geographic" values:

```
var geographic = batchTableJSON.geographic;
var byteOffset = geographic.byteOffset;
var componentType = geographic.componentType;
var type = geographic.type;
var componentSizeInBytes = sizeInBytes(componentType)
var numberComponents = numberComponents(type);
```

```

var geographicArrayByteLength = batchLength * componentSizeInBytes *
numberOfComponents // 10 * 8 * 3
var geographicArray = new Float64Array(batchTableBinary.buffer, byteOffset,
geographicArrayByteLength);
var geographicOfFeature = positionArray.subarray(batchId * numberOfComponents,
batchId * numberOfComponents + numberOfComponents); // Using subarray creates
a view into the array, and not a new array.

```

Code for reading the Batch Table can be found in [Cesium3DTileBatchTable.js](#) in the CesiumJS implementation of 3D Tiles.

## 7.4. Feature Table

---

### WARNING

*Feature Table was deprecated in 3D Tiles 1.1. See glTF migration guide.*

### 7.4.1. Overview

A *Feature Table* is a component of a tile's binary body and describes position and appearance properties required to render each feature in a tile. The Batch Table, on the other hand, contains per-feature application-specific properties not necessarily used for rendering.

A Feature Table is used by tile formats like Batched 3D Model (b3dm) where each model is a feature, and Point Cloud (pnts) where each point is a feature.

Per-feature properties are defined using tile format-specific semantics defined in each tile format's specification. For example, for *Instanced 3D Model*, SCALE\_NON\_UNIFORM defines the non-uniform scale applied to each 3D model instance.

### 7.4.2. Layout

A Feature Table is composed of two parts: a JSON header and an optional binary body in little endian. The JSON property names are tile format-specific semantics, and their values can either be defined directly in the JSON, or refer to sections in the binary body. It is more efficient to store long numeric arrays in the binary body. The following figure shows the Feature Table layout:



**Figure 34 – Data layout of a Feature Table**

When a tile format includes a Feature Table, the Feature Table immediately follows the tile's header. The header will also contain featureTableJSONByteLength and

`featureTableBinaryByteLength` uint32 fields, which can be used to extract each respective part of the Feature Table.

#### 7.4.2.1. Padding

The JSON header shall end on an 8-byte boundary within the containing tile binary. The JSON header shall be padded with trailing Space characters (0x20) to satisfy this requirement.

The binary body shall start and end on an 8-byte boundary within the containing tile binary. The binary body shall be padded with additional bytes, of any value, to satisfy this requirement.

Binary properties shall start at a byte offset that is a multiple of the size in bytes of the property's implicit component type. For example, a property with the implicit component type FLOAT has 4 bytes per element, and therefore shall start at an offset that is a multiple of 4. Preceding binary properties shall be padded with additional bytes, of any value, to satisfy this requirement.

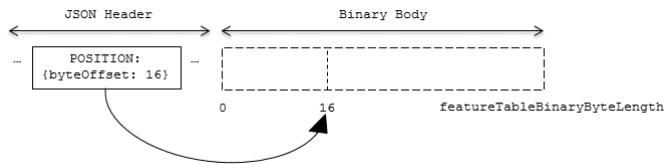
#### 7.4.2.2. JSON header

Feature Table values can be represented in the JSON header in two different ways:

1. A single value or object, e.g., "INSTANCES\_LENGTH" : 4.
  - This is used for global semantics like "INSTANCES\_LENGTH", which defines the number of model instances in an Instanced 3D Model tile.
2. A reference to data in the binary body, denoted by an object with a `byteOffset` property, e.g., "SCALE" : { "byteOffset" : 24}.
  - `byteOffset` specifies a zero-based offset relative to the start of the binary body. The value of `byteOffset` shall be a multiple of the size in bytes of the property's implicit component type, e.g., the "POSITION" property has the component type FLOAT (4 bytes), so the value of `byteOffset` shall be of a multiple of 4.
  - The semantic defines the allowed data type, e.g., when "POSITION" in Instanced 3D Model refers to the binary body, the component type is FLOAT and the number of components is 3.
  - Some semantics allow for overriding the implicit component type. These cases are specified in each tile format, e.g., "BATCH\_ID" : { "byteOffset" : 24, "componentType" : "UNSIGNED\_BYTE" }. The only valid properties in the JSON header are the defined semantics by the tile format and optional extras and extensions properties. Application-specific data should be stored in the Batch Table.

#### 7.4.2.3. Binary body

When the JSON header includes a reference to the binary, the provided byteOffset is used to index into the data. The following figure shows indexing into the Feature Table binary body:



**Figure 35** – An example showing how to access the binary body, based on the information from the JSON header

Values can be retrieved using the number of features, featuresLength; the desired feature id, featureId; and the data type (component type and number of components) for the feature semantic.

#### 7.4.3. Implementation example

*This section is informative*

The following example accesses the position property using the POSITION semantic, which has a float32[3] data type:

```
var byteOffset = featureTableJSON.POSITION.byteOffset;  
  
var positionArray = new Float32Array(featureTableBinary.buffer, byteOffset,  
featuresLength * 3); // There are three components for each POSITION feature.  
var position = positionArray.subarray(featureId * 3, featureId * 3 + 3); //  
Using subarray creates a view into the array, and not a new array.
```

Code for reading the Feature Table can be found in [Cesium3DTileFeatureTable.js](#) in the CesiumJS implementation of 3D Tiles.

### 7.5. Batched 3D Model

#### WARNING

*Batched 3D Model was deprecated in 3D Tiles 1.1. See [b3dm migration guide](#).*

### 7.5.1. Overview

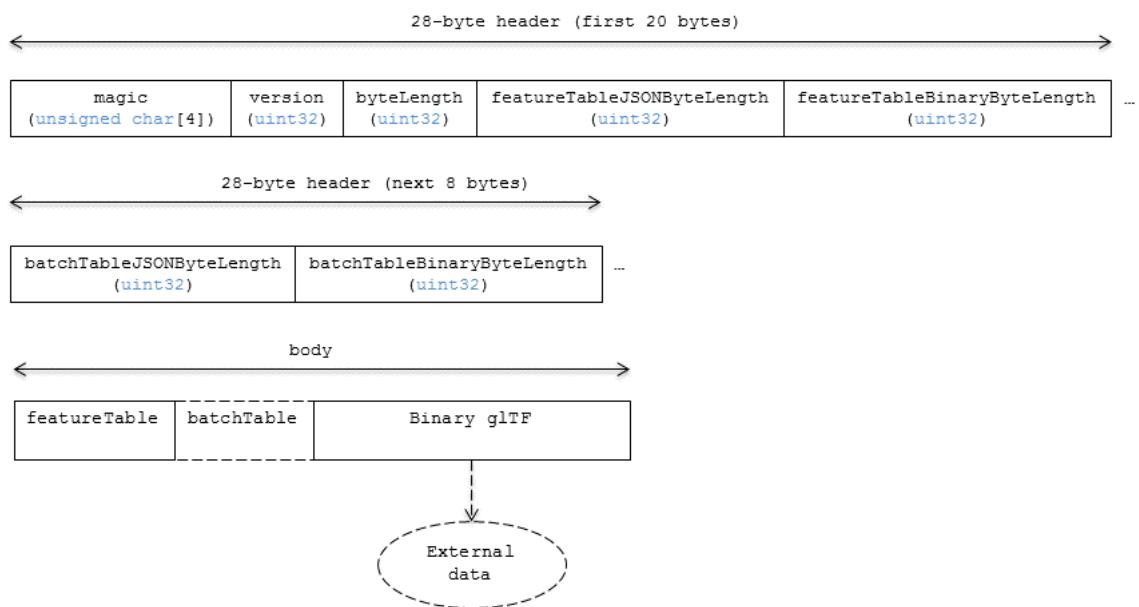
*Batched 3D Model* allows offline batching of heterogeneous 3D models, such as different buildings in a city, for efficient streaming to a web client for rendering and interaction. Efficiency comes from transferring multiple models in a single request and rendering them in the least number of WebGL draw calls necessary. Using the core 3D Tiles spec language, each model is a *feature*.

Per-model properties, such as IDs, enable individual models to be identified and updated at runtime, e.g., show/hide, highlight color, etc. Properties may be used, for example, to query a web service to access metadata, such as passing a building's ID to get its address. Or a property might be referenced on the fly for changing a model's appearance, e.g., changing highlight color based on a property value.

A Batched 3D Model tile is a binary blob in little endian.

### 7.5.2. Layout

A tile is composed of two sections: a header immediately followed by a body. The following figure shows the Batched 3D Model layout (dashes indicate optional fields):



**Figure 36 – Data layout of a Batched 3D Model**

### 7.5.2.1. Padding

A tile's byteLength shall be aligned to an 8-byte boundary. The contained Feature Table and Batch Table shall conform to their respective padding requirement.

The binary glTF shall start and end on an 8-byte boundary so that glTF's byte-alignment guarantees are met. This can be done by padding the Feature Table or Batch Table if they are present.

### 7.5.3. Header

The 28-byte header contains the following fields:

**Table 11 – Header fields for Batched 3D Models**

FIELD NAME	DATA TYPE	DESCRIPTION
magic	4-byte ANSI string	"b3dm". This can be used to identify the content as a Batched 3D Model tile.
version	uint32	The version of the Batched 3D Model format. It is currently 1.
byteLength	uint32	The length of the entire tile, including the header, in bytes.
featureTableJSONByteLength	uint32	The length of the Feature Table JSON section in bytes.
featureTableBinaryByteLength	uint32	The length of the Feature Table binary section in bytes.
batchTableJSONByteLength	uint32	The length of the Batch Table JSON section in bytes. Zero indicates there is no Batch Table.
batchTableBinaryByteLength	uint32	The length of the Batch Table binary section in bytes. If batchTableJSONByteLength is zero, this will also be zero.

The body section immediately follows the header section, and is composed of three fields: Feature Table, Batch Table, and Binary glTF.

### 7.5.4. Feature Table

Contains values for b3dm semantics.

More information is available in the Feature Table specification.

The full JSON schema can be found in `b3dm.featureTable.schema.json`.

### 7.5.4.1. Semantics

#### 7.5.4.1.1. Feature semantics

There are currently no per-feature semantics.

#### 7.5.4.1.2. Global semantics

These semantics define global properties for all features.

**Table 12** – Global semantics for Batched 3D models

SEMANTIC	DATA TYPE	DESCRIPTION	REQUIRED
BATCH_LENGTH	uint32	The number of distinguishable models, also called features, in the batch. If the Binary glTF does not have a <code>batchId</code> attribute, this field <i>shall</i> be 0.	Yes.
RTC_CENTER	float32[3]	A 3-component array of numbers defining the center position when positions are defined relative-to-center, (see Coordinate system).	No.

### 7.5.5. Batch Table

The *Batch Table* contains per-model application-specific properties, indexable by `batchId`, that can be used for declarative styling and application-specific use cases such as populating a UI or issuing a REST API request. In the binary glTF section, each vertex has a numeric `batchId` attribute in the integer range `[0, number of models in the batch - 1]`. The `batchId` indicates the model to which the vertex belongs. This allows models to be batched together and still be identifiable.

See the Batch Table reference for more information.

### 7.5.6. Binary glTF

Batched 3D Model embeds [glTF 2.0](#) containing model geometry and texture information.

The [binary glTF](#) immediately follows the Feature Table and Batch Table. It may embed all of its geometry, texture, and animations, or it may refer to external sources for some or all of these data.

As described above, each vertex has a batchId attribute indicating the model to which it belongs. For example, vertices for a batch with three models may look like this:

```
batchId: [0, 0, 0, ..., 1, 1, 1, ..., 2, 2, 2, ...]  
position: [xyz, xyz, xyz, ..., xyz, xyz, xyz, ..., xyz, xyz, xyz, ...]  
normal: [xyz, xyz, xyz, ..., xyz, xyz, xyz, ..., xyz, xyz, xyz, ...]
```

Vertices do not need to be ordered by batchId, so the following is also OK:

```
batchId: [0, 1, 2, ..., 2, 1, 0, ..., 1, 2, 0, ...]  
position: [xyz, xyz, xyz, ..., xyz, xyz, xyz, ..., xyz, xyz, xyz, ...]  
normal: [xyz, xyz, xyz, ..., xyz, xyz, xyz, ..., xyz, xyz, xyz, ...]
```

Note that a vertex can't belong to more than one model; in that case, the vertex needs to be duplicated so the batchIds can be assigned.

The batchId parameter is specified in a glTF [mesh primitive](#) by providing the `_BATCHID` attribute semantic, along with the index of the `batchId` accessor. For example,

```
"primitives": [  
    {  
        "attributes": {  
            "_BATCHID": 0  
        }  
    }  
]  
{  
    "accessors": [  
        {  
            "bufferView": 1,  
            "byteOffset": 0,  
            "componentType": 5126,  
            "count": 4860,  
            "max": [2],  
            "min": [0],  
            "type": "SCALAR"  
        }  
    ]  
}
```

The accessor.type shall be a value of "SCALAR". All other properties shall conform to the glTF schema, but have no additional requirements.

When a Batch Table is present or the `BATCH_LENGTH` property is greater than 0, the `_BATCHID` attribute is required; otherwise, it is not.

### 7.5.6.1. Coordinate system

By default embedded glTFS use a right handed coordinate system where the y-axis is up. For consistency with the z-up coordinate system of 3D Tiles, glTFS shall be transformed at runtime. See glTF transforms for more details.

Vertex positions may be defined relative-to-center for high-precision rendering, see [Precisions](#), [Precisions](#). If defined, RTC\_CENTER specifies the center position that all vertex positions are relative to after the coordinate system transform and glTF node hierarchy transforms have been applied. Specifically, when the RTC\_CENTER is defined in the feature table of a Batched 3D Model, the computation of the tile transform is done as follows:

1. glTF node hierarchy transformations
2. glTF y-up to z-up transform
3. The transform for the RTC\_CENTER, which is used to translate model vertices
4. Tile transform

### 7.5.7. File extension and media type

Batched 3D Model tiles use the .b3dm extension and application/octet-stream media type.

An explicit file extension is optional. Valid implementations may ignore it and identify a content's format by the `magic` field in its header.

### 7.5.8. Implementation example

*This section is informative*

Code for reading the header can be found in [Batched3DModelTileContent.js](#) in the CesiumJS implementation of 3D Tiles.

## 7.6. Instanced 3D Model

---

### WARNING

*Instanced 3D Model was deprecated in 3D Tiles 1.1. See [i3dm migration guide](#).*

### 7.6.1. Overview

*Instanced 3D Model* is a tile format for efficient streaming and rendering of a large number of models, called *instances*, with slight variations. In the simplest case, the same tree model, for example, may be located—or *instanced*—in several places. Each instance references the same model and has per-instance properties, such as position. Using the core 3D Tiles spec language, each instance is a *feature*.

In addition to trees, Instanced 3D Model is useful for exterior features such as fire hydrants, sewer caps, lamps, and traffic lights, and for interior CAD features such as bolts, valves, and electrical outlets.

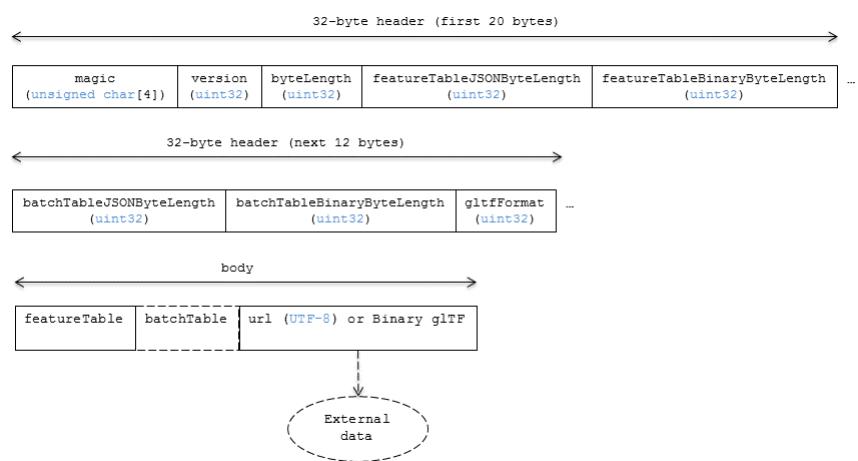
An Instanced 3D Model tile is a binary blob in little endian.

**NOTE 1A** Composite tile can be used to create tiles with different types of instanced models, e.g., trees and traffic lights by combining two Instanced 3D Model tiles.

**NOTE 2** Instanced 3D Model maps well to the `ANGLE_instanced_arrays` extension for efficient rendering with WebGL.

## 7.6.2. Layout

A tile is composed of a header section immediately followed by a binary body. The following figure shows the Instanced 3D Model layout (dashes indicate optional fields):



**Figure 37 – Data layout for Instanced 3D Models**

### 7.6.2.1. Padding

A tile's `byteLength` shall be aligned to an 8-byte boundary. The contained Feature Table and Batch Table shall conform to their respective padding requirement.

The binary glTF (if present) shall start and end on an 8-byte boundary so that glTF's byte-alignment guarantees are met. This can be done by padding the Feature Table or Batch Table if they are present.

Otherwise, if the glTF field is a UTF-8 string, it shall be padded with trailing Space characters (`0x20`) to satisfy alignment requirements of the tile, which shall be removed at runtime before requesting the glTF asset.

### 7.6.3. Header

The 32-byte header contains the following fields:

**Table 13 – Header fields for Instanced 3D Models**

FIELD NAME	DATA TYPE	DESCRIPTION
magic	4-byte ANSI string	"i3dm". This can be used to identify the content as an Instanced 3D Model tile.
version	uint32	The version of the Instanced 3D Model format. It is currently 1.
byteLength	uint32	The length of the entire tile, including the header, in bytes.
featureTableJSONByteLength	uint32	The length of the Feature Table JSON section in bytes.
featureTableBinaryByteLength	uint32	The length of the Feature Table binary section in bytes.
batchTableJSONByteLength	uint32	The length of the Batch Table JSON section in bytes. Zero indicates that there is no Batch Table.
batchTableBinaryByteLength	uint32	The length of the Batch Table binary section in bytes. If batchTableJSONByteLength is zero, this will also be zero.
glTFFormat	uint32	Indicates the format of the glTF field of the body. 0 indicates it is a URI, 1 indicates it is embedded binary glTF. See the glTF section below.

The body section immediately follows the header section and is composed of three fields: Feature Table, Batch Table, and glTF.

### 7.6.4. Feature Table

The Feature Table contains values for i3dm semantics used to create instanced models. More information is available in the Feature Table specification.

The full JSON schema can be found in `i3dm.featureTable.schema.json`.

## 7.6.4.1. Semantics

### 7.6.4.1.1. Instance semantics

These semantics map to an array of feature values that are used to create instances. The length of these arrays shall be the same for all semantics and is equal to the number of instances. The value for each instance semantic shall be a reference to the Feature Table binary body; they cannot be embedded in the Feature Table JSON header.

If a semantic has a dependency on another semantic, that semantic shall be defined. If both SCALE and SCALE\_NON\_UNIFORM are defined for an instance, both scaling operations will be applied. If both POSITION and POSITION\_QUANTIZED are defined for an instance, the higher precision POSITION will be used. If NORMAL\_UP, NORMAL\_RIGHT, NORMAL\_UP\_OCT32P, and NORMAL\_RIGHT\_OCT32P are defined for an instance, the higher precision NORMAL\_UP and NORMAL\_RIGHT will be used.

**Table 14 – Semantics for instances of an Instanced 3D Model**

SEMANTIC	DATA TYPE	DESCRIPTION	REQUIRED
POSITION	float32[3]	A 3-component array of numbers containing x, y, and z Cartesian coordinates for the position of the instance.	Yes, unless POSITION_QUANTIZED is defined.
POSITION_QUANTIZED	uint16[3]	A 3-component array of numbers containing x, y, and z in quantized Cartesian coordinates for the position of the instance.	Yes, unless POSITION is defined.
NORMAL_UP	float32[3]	A unit vector defining the up direction for the orientation of the instance.	No, unless NORMAL_RIGHT is defined.
NORMAL_RIGHT	float32[3]	A unit vector defining the right direction for the orientation of the instance. Shall be orthogonal to up.	No, unless NORMAL_UP is defined.
NORMAL_UP_OCT32P	uint16[2]	An oct-encoded unit vector with 32-bits of precision defining the up direction for the orientation of the instance.	No, unless NORMAL_RIGHT_OCT32P is defined.
NORMAL_RIGHT_OCT32P	uint16[2]	An oct-encoded unit vector with 32-bits of precision defining the right direction	No, unless NORMAL_UP_

SEMANTIC	DATA TYPE	DESCRIPTION	REQUIRED
		for the orientation of the instance. Shall be orthogonal to up.	OCT32P is defined.
SCALE	float32	A number defining a scale to apply to all axes of the instance.	No.
SCALE_NON_UNIFORM	float32[3]	A 3-component array of numbers defining the scale to apply to the x, y, and z axes of the instance.	No.
BATCH_ID	uint8, uint16 (default), or uint32	The batchId of the instance that can be used to retrieve metadata from the Batch Table.	No.

#### 7.6.4.1.2. Global semantics

These semantics define global properties for all instances.

**Table 15 – Global semantics for Instanced 3D Models**

SEMANTIC	DATA TYPE	DESCRIPTION	REQUIRED
INSTANCES_LENGTH	uint32	The number of instances to generate. The length of each array value for an instance semantic should be equal to this.	Yes.
RTC_CENTER	float32[3]	A 3-component array of numbers defining the center position when instance positions are defined relative-to-center.	No.
QUANTIZED_VOLUME_OFFSET	float32[3]	A 3-component array of numbers defining the offset for the quantized volume.	No, unless POSITION_QUANTIZED is defined.
QUANTIZED_VOLUME_SCALE	float32[3]	A 3-component array of numbers defining the scale for the quantized volume.	No, unless POSITION_QUANTIZED is defined.
EAST_NORTH_UP	boolean	When true and per-instance orientation is not defined, each instance will default to the east/north/up reference frame's orientation on the WGS84 ellipsoid.	No.

Examples using these semantics can be found in the examples section.

#### 7.6.4.2. Instance orientation

An instance's orientation is defined by an orthonormal basis created by an up and right vector. The orientation will be transformed by the tile transform.

The x vector in the standard basis maps to the right vector in the transformed basis, and the y vector maps to the up vector. The z vector would map to a forward vector, but it is omitted because it will always be the cross product of right and up.

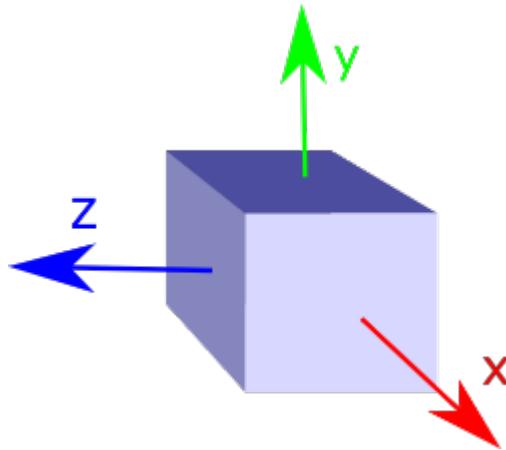


Figure 38 – A box in the standard basis

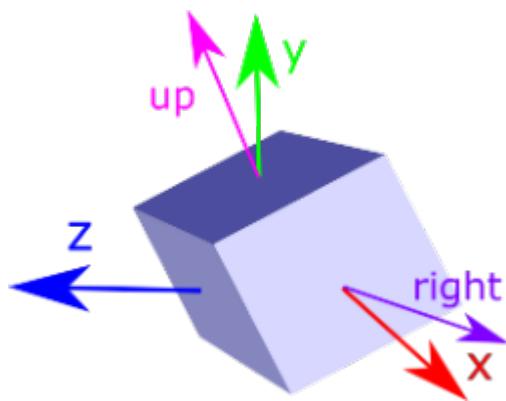


Figure 39 – A box transformed into a rotated basis

#### 7.6.4.2.1. Oct-encoded normal vectors

If NORMAL\_UP and NORMAL\_RIGHT are not defined for an instance, its orientation may be stored as oct-encoded normals in NORMAL\_UP\_OCT32P and NORMAL\_RIGHT\_OCT32P. These define up and right using the oct-encoding described in [A Survey of Efficient Representations of Independent Unit Vectors](#). Oct-encoded values are stored in unsigned, unnormalized range ( $[0, 65535]$ ) and then mapped to a signed normalized range ( $[-1.0, 1.0]$ ) at runtime.

**NOTE**An implementation for encoding and decoding these unit vectors can be found in CesiumJS's [AttributeCompression](#) module.

#### 7.6.4.2.2. Default orientation

If NORMAL\_UP and NORMAL\_RIGHT or NORMAL\_UP\_OCT32P and NORMAL\_RIGHT\_OCT32P are not present, the instance will not have a custom orientation. If EAST\_NORTH\_UP is true, the instance is assumed to be on the WGS84 ellipsoid and its orientation will default to the east/north/up reference frame at its cartographic position. This is suitable for instanced models such as trees whose orientation is always facing up from their position on the ellipsoid's surface.

#### 7.6.4.3. Instance position

POSITION defines the location for an instance before any tile transforms are applied.

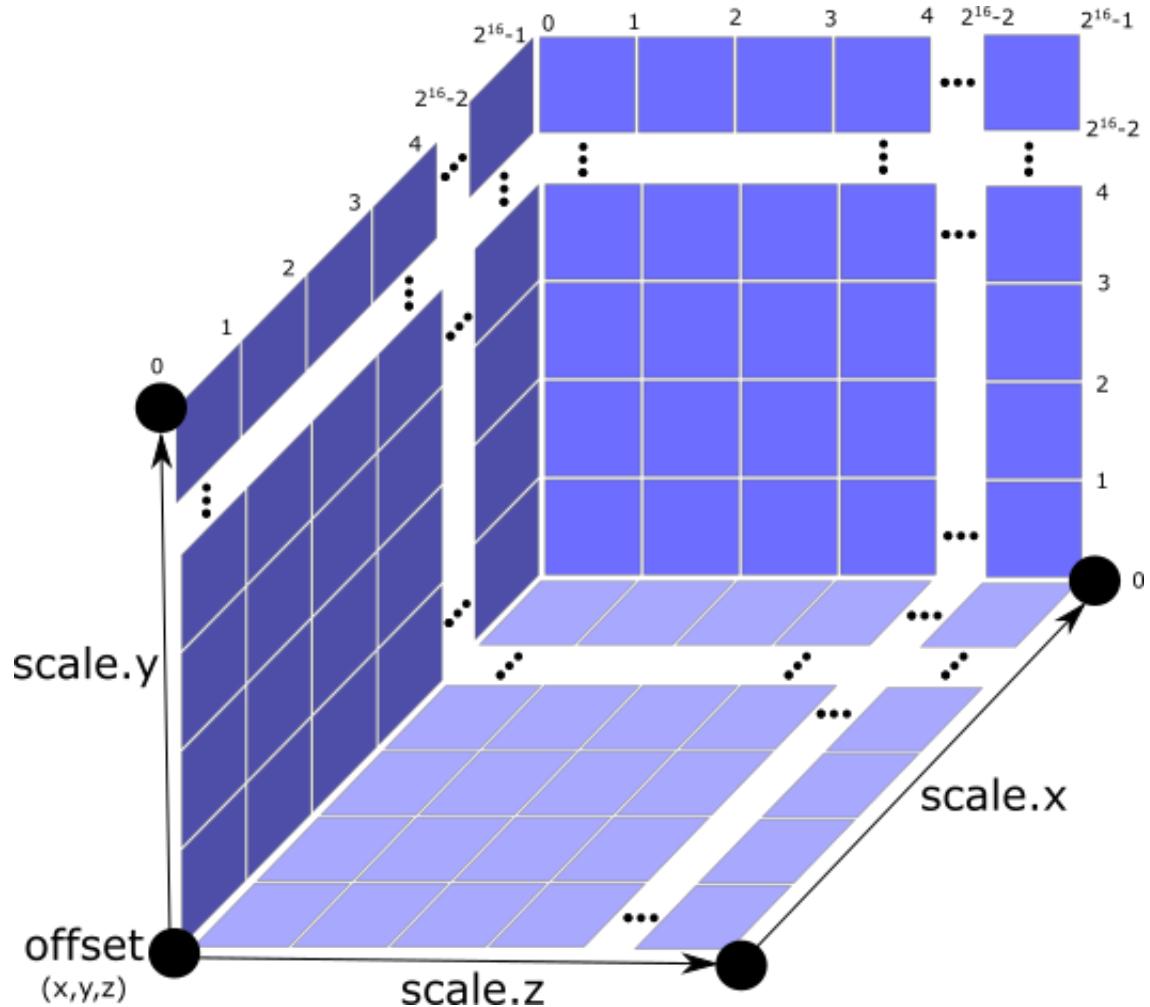
##### 7.6.4.3.1. RTC\_CENTER

Positions may be defined relative-to-center for high-precision rendering, see [Precisions](#), [Precisions](#). If defined, RTC\_CENTER specifies the center position and all instance positions are treated as relative to this value. See Coordinate System for the effect that this property has on the transform.

##### 7.6.4.3.2. Quantized positions

If POSITION is not defined for an instance, its position may be stored in POSITION\_QUANTIZED, which defines the instance position relative to the quantized volume. If neither POSITION or POSITION\_QUANTIZED are defined, the instance will not be created.

A quantized volume is defined by offset and scale to map quantized positions into local space, as shown in the following figure:



**Figure 40 – Illustration of the quantization that is used for the POSITION\_QUANTIZED semantic**  
 offset is stored in the global semantic QUANTIZED\_VOLUME\_OFFSET, and scale is stored in the global semantic QUANTIZED\_VOLUME\_SCALE. If those global semantics are not defined, POSITION\_QUANTIZED cannot be used.

Quantized positions can be mapped to local space using the following formula:

```
POSITION = POSITION_QUANTIZED * QUANTIZED_VOLUME_SCALE / 65535.0 + QUANTIZED_VOLUME_OFFSET
```

Compressed attributes should be decompressed before any other transforms are applied.

#### 7.6.4.4. Instance scaling

Scaling can be applied to instances using the SCALE and SCALE\_NON\_UNIFORM semantics. SCALE applies a uniform scale along all axes, and SCALE\_NON\_UNIFORM applies scaling to the x, y, and z axes independently.

#### 7.6.4.5. Examples

These examples show how to generate JSON and binary buffers for the Feature Table.

##### 7.6.4.5.1. Positions only

In this minimal example, we place four instances on the corners of a unit length square with the default orientation:

```
var featureTableJSON = {
  INSTANCES_LENGTH : 4,
  POSITION : {
    byteOffset : 0
  }
};

var featureTableBinary = new Buffer(new Float32Array([
  0.0, 0.0, 0.0,
  1.0, 0.0, 0.0,
  0.0, 0.0, 1.0,
  1.0, 0.0, 1.0
]).buffer);
```

##### 7.6.4.5.2. Quantized positions and oct-encoded normals

In this example, the four instances will be placed with an orientation up of [0.0, 1.0, 0.0] and right of [1.0, 0.0, 0.0] in oct-encoded format and they will be placed on the corners of a quantized volume that spans from -250.0 to 250.0 units in the x and z directions:

```
var featureTableJSON = {
  INSTANCES_LENGTH : 4,
  QUANTIZED_VOLUME_OFFSET : [-250.0, 0.0, -250.0],
  QUANTIZED_VOLUME_SCALE : [500.0, 0.0, 500.0],
  POSITION_QUANTIZED : {
    byteOffset : 0
  },
  NORMAL_UP_OCT32P : {
    byteOffset : 24
  },
  NORMAL_RIGHT_OCT32P : {
    byteOffset : 40
  }
};

var positionQuantizedBinary = new Buffer(new Uint16Array([
  0, 0, 0,
  65535, 0, 0,
  0, 0, 65535,
  65535, 0, 65535
]).buffer);

var normalUpOct32PBinary = new Buffer(new Uint16Array([
  32768, 65535,
  32768, 65535,
  32768, 65535,
  32768, 65535
]).buffer);
```

```

    32768, 65535,
    32768, 65535
]).buffer);

var normalRightOct32PBinary = new Buffer(new Uint16Array([
    65535, 32768,
    65535, 32768,
    65535, 32768,
    65535, 32768
]).buffer);

var featureTableBinary = Buffer.concat([positionQuantizedBinary,
normalUpOct32PBinary, normalRightOct32PBinary]);

```

## 7.6.5. Batch Table

Contains metadata organized by batchId that can be used for declarative styling. See the Batch Table reference for more information.

## 7.6.6. glTF

Instanced 3D Model embeds glTF 2.0 containing model geometry and texture information.

The glTF asset to be instanced is stored after the Feature Table and Batch Table. It may embed all of its geometry, texture, and animations, or it may refer to external sources for some or all of these data.

`header.gltfFormat` determines the format of the glTF field

- When the value of `header.gltfFormat` is 0, the glTF field is a UTF-8 string, which contains a URI of the glTF or binary glTF model content.
- When the value of `header.gltfFormat` is 1, the glTF field is a binary blob containing binary glTF.

When the glTF field contains a URI, then this URI may point to a relative external reference (RFC3986). When the URI is relative, its base is always relative to the referring .i3dm file. Client implementations are required to support relative external references. Optionally, client implementations may support other schemes (such as `http://`). All URIs shall be valid and resolvable.

### 7.6.6.1. Coordinate system

By default glTFs use a right handed coordinate system where the y-axis is up. For consistency with the z-up coordinate system of 3D Tiles, glTFs shall be transformed at runtime. See glTF transforms for more details.

When the `RTC_CENTER` is defined in the feature table of an Instanced 3D Model, the computation of the tile transform is done as follows:

1. gITF node hierarchy transformations
2. gITF y-up to z-up transform
3. The per-instance positions and scales, as defined in the feature table of the Instanced 3D Model.
4. The transform for the RTC\_CENTER, which is used to translate model vertices
5. Tile transform

### 7.6.7. File extension and media type

Instanced 3D models tiles use the .i3dm extension and application/octet-stream media type.

An explicit file extension is optional. Valid implementations may ignore it and identify a content's format by the magic field in its header.

## 7.7. Point Cloud

---

### WARNING

*Point Cloud was deprecated in 3D Tiles 1.1. See pnts migration guide.*

### 7.7.1. Overview

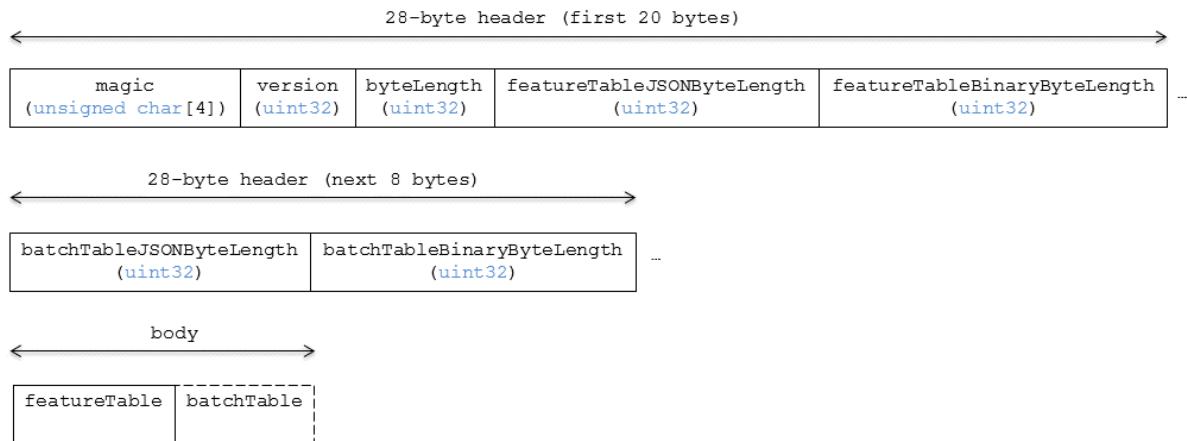
The *Point Cloud* tile format enables efficient streaming of massive point clouds for 3D visualization. Each point is defined by a position and by optional properties used to define its appearance, such as color and normal, as well as optional properties that define application-specific metadata.

Using 3D Tiles terminology, each point is a *feature*.

A Point Cloud tile is a binary blob in little endian.

### 7.7.2. Layout

A tile is composed of a header section immediately followed by a body section. The following figure shows the Point Cloud layout (dashes indicate optional fields):



**Figure 41 – Data layout for a Point Cloud**

### 7.7.2.1. Padding

A tile's byteLength shall be aligned to an 8-byte boundary. The contained Feature Table and Batch Table shall conform to their respective padding requirement.

### 7.7.3. Header

The 28-byte header contains the following fields:

**Table 16 – Header fields for Point Clouds**

FIELD NAME	DATA TYPE	DESCRIPTION
magic	4-byte ANSI string	"pnts". This can be used to identify the content as a Point Cloud tile.
version	uint32	The version of the Point Cloud format. It is currently 1.
byteLength	uint32	The length of the entire tile, including the header, in bytes.
featureTableJSONByteLength	uint32	The length of the Feature Table JSON section in bytes.
featureTableBinaryByteLength	uint32	The length of the Feature Table binary section in bytes.
batchTableJSONByteLength	uint32	The length of the Batch Table JSON section in bytes. Zero indicates that there is no Batch Table.

FIELD NAME	DATA TYPE	DESCRIPTION
batchTableBinaryByteLength	uint32	The length of the Batch Table binary section in bytes. If batchTableJSONByteLength is zero, this will also be zero.

The body section immediately follows the header section, and is composed of a Feature Table and Batch Table.

## 7.7.4. Feature Table

Contains per-tile and per-point values that define where and how to render points. More information is available in the Feature Table specification.

The full JSON schema can be found in pnts.featureTable.schema.json.

### 7.7.4.1. Semantics

#### 7.7.4.1.1. Point semantics

These semantics map to an array of feature values that define each point. The length of these arrays shall be the same for all semantics and is equal to the number of points. The value for each point semantic shall be a reference to the Feature Table binary body; they cannot be embedded in the Feature Table JSON header.

If a semantic has a dependency on another semantic, that semantic shall be defined. If both POSITION and POSITION\_QUANTIZED are defined for a point, the higher precision POSITION will be used. If both NORMAL and NORMAL\_OCT16P are defined for a point, the higher precision NORMAL will be used.

**Table 17** – Semantics for instances of a Point Cloud

SEMANTIC	DATA TYPE	DESCRIPTION	REQUIRED
POSITION	float32[3]	A 3-component array of numbers containing x, y, and z Cartesian coordinates for the position of the point.	Yes, unless POSITION_QUANTIZED is defined.
POSITION_QUANTIZED	uint16[3]	A 3-component array of numbers containing x, y, and z in quantized Cartesian coordinates for the position of the point.	Yes, unless POSITION is defined.
RGBA	uint8[4]	A 4-component array of values containing the RGBA color of the point.	No.

SEMANTIC	DATA TYPE	DESCRIPTION	REQUIRED
RGB	uint8[3]	A 3-component array of values containing the RGB color of the point.	No.
RGB565	uint16	A lossy compressed color format that packs the RGB color into 16 bits, providing 5 bits for red, 6 bits for green, and 5 bits for blue.	No.
NORMAL	float32[3]	A unit vector defining the normal of the point.	No.
NORMAL_OCT16P	uint8[2]	An oct-encoded unit vector with 16 bits of precision defining the normal of the point.	No.
BATCH_ID	uint8, uint16 (default), or uint32	The batchId of the point that can be used to retrieve metadata from the Batch Table.	No.

#### 7.7.4.1.2. Global semantics

These semantics define global properties for all points.

**Table 18 – Global semantics for Point Clouds**

SEMANTIC	DATA TYPE	DESCRIPTION	REQUIRED
POINTS_LENGTH	uint32	The number of points to render. The length of each array value for a point semantic should be equal to this.	Yes.
RTC_CENTER	float32[3]	A 3-component array of numbers defining the center position when point positions are defined relative-to-center.	No.
QUANTIZED_VOLUME_OFFSET	float32[3]	A 3-component array of numbers defining the offset for the quantized volume.	No, unless POSITION_QUANTIZED is defined.
QUANTIZED_VOLUME_SCALE	float32[3]	A 3-component array of numbers defining the scale for the quantized volume.	No, unless POSITION_QUANTIZED is defined.
CONSTANT_RGBA	uint8[4]	A 4-component array of values defining a constant RGBA color for all points in the tile.	No.

SEMANTIC	DATA TYPE	DESCRIPTION	REQUIRED
BATCH_LENGTH	uint32	The number of unique BATCH_ID values.	No, unless BATCH_ID is defined.

Examples using these semantics can be found in the examples section below.

#### 7.7.4.2. Point positions

POSITION defines the position for a point before any tileset transforms are applied.

##### 7.7.4.2.1. Coordinate reference system (CRS)

3D Tiles local coordinate systems use a right-handed 3-axis (x, y, z) Cartesian coordinate system; that is, the cross product of x and y yields z. 3D Tiles defines the z axis as up for local Cartesian coordinate systems (also see coordinate reference system).

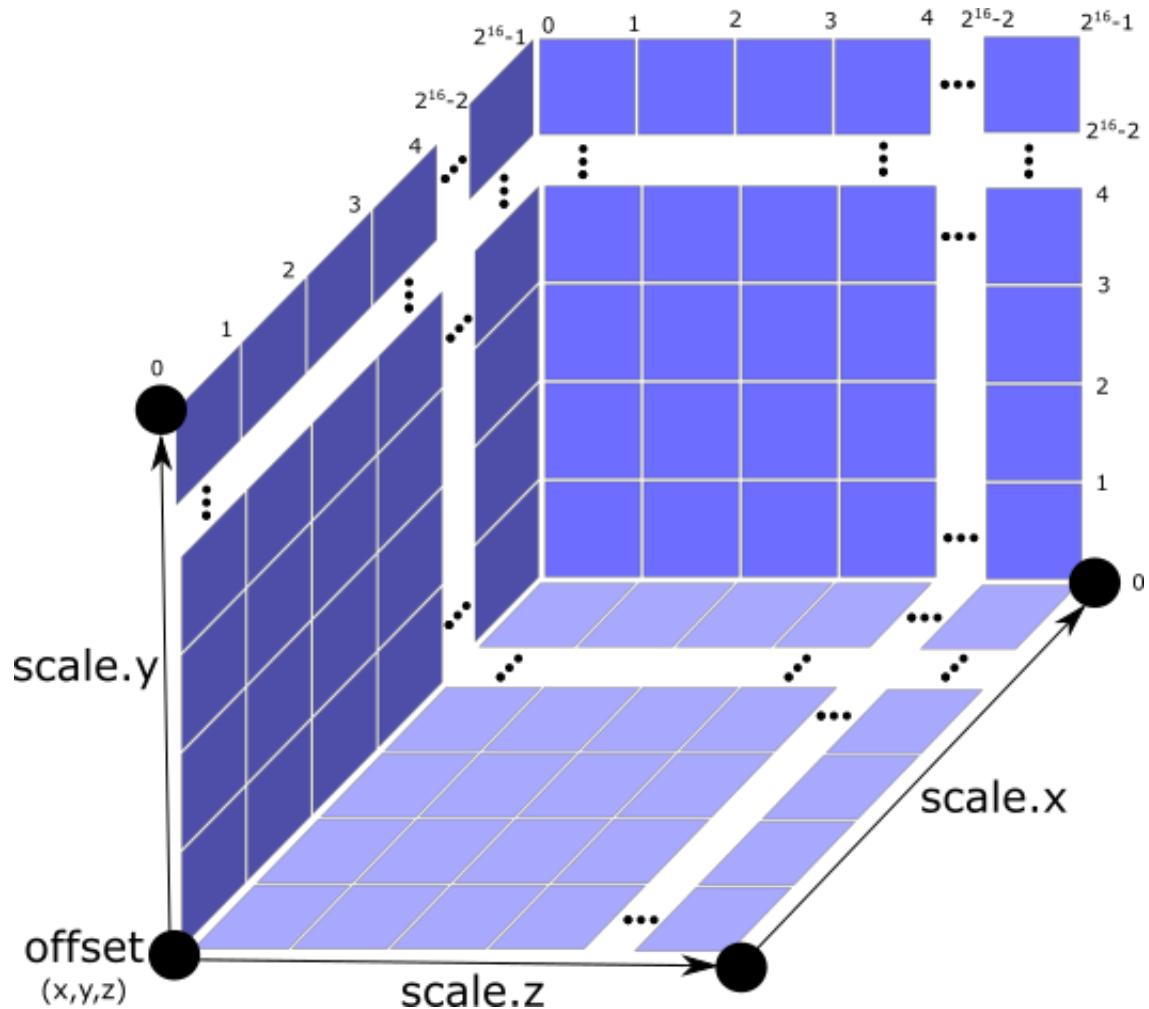
##### 7.7.4.2.2. RTC\_CENTER

Positions may be defined relative-to-center for high-precision rendering, see [Precisions](#), [Precisions](#). If defined, RTC\_CENTER specifies the center position and all point positions are treated as relative to this value.

##### 7.7.4.2.3. Quantized positions

If POSITION is not defined, positions may be stored in POSITION\_QUANTIZED, which defines point positions relative to the quantized volume. If neither POSITION nor POSITION\_QUANTIZED is defined, the tile does not need to be rendered.

A quantized volume is defined by offset and scale to map quantized positions to a position in local space. The following figure shows a quantized volume based on offset and scale:



**Figure 42 – Illustration of the quantization that is used for the POSITION\_QUANTIZED semantic**  
 offset is stored in the global semantic QUANTIZED\_VOLUME\_OFFSET, and scale is stored in the global semantic QUANTIZED\_VOLUME\_SCALE. If those global semantics are not defined, POSITION\_QUANTIZED cannot be used.

Quantized positions can be mapped to local space using the following formula:

```
POSITION = POSITION_QUANTIZED * QUANTIZED_VOLUME_SCALE / 65535.0 + QUANTIZED_VOLUME_OFFSET
```

Compressed attributes should be decompressed before any other transforms are applied.

#### 7.7.4.3. Point colors

If more than one color semantic is defined, the precedence order is RGBA, RGB, RGB565, then CONSTANT\_RGBA. For example, if a tile's Feature Table contains both RGBA and CONSTANT\_RGBA properties, the runtime would render with per-point colors using RGBA.

If no color semantics are defined, the runtime is free to color points using an application-specific default color.

In any case, 3D Tiles Styling may be used to change the final rendered color and other visual properties at runtime.

#### 7.7.4.4. Point normals

Per-point normals are an optional property that can help improve the visual quality of points by enabling lighting, hidden surface removal, and other rendering techniques. The normals will be transformed using the inverse transpose of the tileset transform.

##### 7.7.4.4.1. Oct-encoded normal vectors

Oct-encoding is described in [A Survey of Efficient Representations of Independent Unit Vectors](#). Oct-encoded values are stored in unsigned, unnormalized range ([0, 255]) and then mapped to a signed normalized range ([-1.0, 1.0]) at runtime.

**NOTE**An implementation for encoding and decoding these unit vectors can be found in CesiumJS's [AttributeCompression](#) module.

Compressed attributes should be decompressed before any other transforms are applied.

#### 7.7.4.5. Batched points

Points that make up distinct features of the Point Cloud may be batched together using the BATCH\_ID semantic. For example, the points that make up a door in a house would all be assigned the same BATCH\_ID, whereas points that make up a window would be assigned a different BATCH\_ID. This is useful for per-object picking and storing application-specific metadata for declarative styling and application-specific use cases such as populating a UI or issuing a REST API request on a per-object instead of per-point basis.

The BATCH\_ID semantic may have a componentType of UNSIGNED\_BYTE, UNSIGNED\_SHORT, or UNSIGNED\_INT. When componentType is not present, UNSIGNED\_SHORT is used. The global semantic BATCH\_LENGTH defines the number of unique batchId values, similar to the batchLength field in the Batched 3D Model header.

#### 7.7.4.6. Examples

*This section is informative*

These examples show how to generate JSON and binary buffers for the Feature Table.

#### 7.7.4.6.1. Positions only

This minimal example has four points on the corners of a unit length square:

```
var featureTableJSON = {
  POINTS_LENGTH : 4,
  POSITION : [
    {
      byteOffset : 0
    }
  ];
}

var featureTableBinary = new Buffer(new Float32Array([
  0.0, 0.0, 0.0,
  1.0, 0.0, 0.0,
  0.0, 0.0, 1.0,
  1.0, 0.0, 1.0
]).buffer);
```

#### 7.7.4.6.2. Positions and colors

The following example has four points (red, green, blue, and yellow) above the globe. Their positions are defined relative to center:

```
var featureTableJSON = {
  POINTS_LENGTH : 4,
  RTC_CENTER : [1215013.8, -4736316.7, 4081608.4],
  POSITION : [
    {
      byteOffset : 0
    },
    {
      byteOffset : 48
    }
  ];
}

var positionBinary = new Buffer(new Float32Array([
  0.0, 0.0, 0.0,
  1.0, 0.0, 0.0,
  0.0, 0.0, 1.0,
  1.0, 0.0, 1.0
]).buffer);

var colorBinary = new Buffer(new Uint8Array([
  255, 0, 0,
  0, 255, 0,
  0, 0, 255,
  255, 255, 0,
]).buffer);

var featureTableBinary = Buffer.concat([positionBinary, colorBinary]);
```

#### 7.7.4.6.3. Quantized positions and oct-encoded normals

In this example, the four points will have normals pointing up [0.0, 1.0, 0.0] in oct-encoded format, and they will be placed on the corners of a quantized volume that spans from -250.0 to 250.0 units in the x and z directions:

```
var featureTableJSON = {
  POINTS_LENGTH : 4,
  QUANTIZED_VOLUME_OFFSET : [-250.0, 0.0, -250.0],
  QUANTIZED_VOLUME_SCALE : [500.0, 0.0, 500.0],
  POSITION_QUANTIZED : {
    byteOffset : 0
  },
  NORMAL_OCT16P : {
    byteOffset : 24
  }
};

var positionQuantizedBinary = new Buffer(new Uint16Array([
  0, 0, 0,
  65535, 0, 0,
  0, 0, 65535,
  65535, 0, 65535
]).buffer);

var normalOct16PBinary = new Buffer(new Uint8Array([
  128, 255,
  128, 255,
  128, 255,
  128, 255
]).buffer);

var featureTableBinary = Buffer.concat([positionQuantizedBinary,
normalOct16PBinary]);
```

#### 7.7.4.6.4. Batched points

In this example, the first two points have a batchId of 0, and the next two points have a batchId of 1. Note that the Batch Table only has two names:

```
var featureTableJSON = {
  POINTS_LENGTH : 4,
  BATCH_LENGTH : 2,
  POSITION : {
    byteOffset : 0
  },
  BATCH_ID : {
    byteOffset : 48,
    componentType : "UNSIGNED_BYTE"
  }
};

var positionBinary = new Buffer(new Float32Array([
  0.0, 0.0, 0.0,
  1.0, 0.0, 0.0,
  0.0, 0.0, 1.0,
  1.0, 1.0, 1.0
]).buffer);
```

```

    1.0, 0.0, 1.0
]).buffer);

var batchIdBinary = new Buffer(new Uint8Array([
  0,
  0,
  1,
  1
]).buffer);

var featureTableBinary = Buffer.concat([positionBinary, batchIdBinary]);

var batchTableJSON = {
  names : ['object1', 'object2']
};

```

#### 7.7.4.6.5. Per-point properties

In this example, each of the 4 points will have metadata stored in the Batch Table JSON and binary.

```

var featureTableJSON = {
  POINTS_LENGTH : 4,
  POSITION : {
    byteOffset : 0
  }
};

var featureTableBinary = new Buffer(new Float32Array([
  0.0, 0.0, 0.0,
  1.0, 0.0, 0.0,
  0.0, 0.0, 1.0,
  1.0, 0.0, 1.0
]).buffer);

var batchTableJSON = {
  names : ['point1', 'point2', 'point3', 'point4']
};

```

### 7.7.5. Batch Table

The *Batch Table* contains application-specific metadata, indexable by `batchId`, that can be used for declarative styling and application-specific use cases such as populating a UI or issuing a REST API request.

- If the `BATCH_ID` semantic is defined, the Batch Table stores metadata for each `batchId`, and the length of the Batch Table arrays will equal `BATCH_LENGTH`.
- If the `BATCH_ID` semantic is not defined, then the Batch Table stores per-point metadata, and the length of the Batch Table arrays will equal `POINTS_LENGTH`.

See the Batch Table reference for more information.

## 7.7.6. Extensions

The following extensions can be applied to a Point Cloud tile.

- `3DTILES_draco_point_compression`

## 7.7.7. File extension and media type

Point cloud tiles use the `.pnts` extension and `application/octet-stream` media type.

An explicit file extension is optional. Valid implementations may ignore it and identify a content's format by the magic field in its header.

## 7.7.8. Implementation example

*This section is informative*

Code for reading the header can be found in `PointCloud3DModelTileContent.js` in the CesiumJS implementation of 3D Tiles.

# 7.8. Composite

### WARNING

*Composite was deprecated in 3D Tiles 1.1. See [cmpt migration guide](#).*

## 7.8.1. Overview

The `Composite` tile format enables concatenating tiles of different formats into one tile.

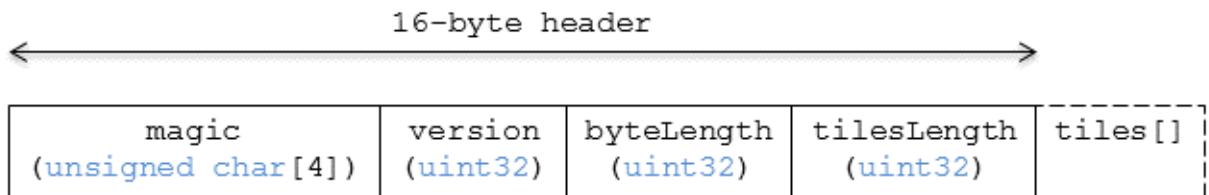
3D Tiles and the `Composite` tile allow flexibility for streaming heterogeneous datasets. For example, buildings and trees could be stored either in two separate `Batched 3D Model` and `Instanced 3D Model` tiles or, using a `Composite` tile, the tiles can be combined.

Supporting heterogeneous datasets with both inter-tile (separate tiles of different formats that are in the same tileset) and intra-tile (different tile formats that are in the same `Composite` tile) options allows conversion tools to make trade-offs between number of requests, optimal type-specific subdivision, and how visible/hidden layers are streamed.

A `Composite` tile is a binary blob in little endian.

## 7.8.2. Layout

Composite layout (dashes indicate optional fields):



**Figure 43 – Data layout of a Composite tile**

### 7.8.2.1. Padding

A tile's byteLength shall be aligned to an 8-byte boundary. All tiles contained in a composite tile shall also be aligned to an 8-byte boundary.

## 7.8.3. Header

The 16-byte header section contains the following fields:

**Table 19 – Header fields for Composite tiles**

FIELD NAME	DATA TYPE	DESCRIPTION
magic	4-byte ANSI string	"cmpt ". This can be used to identify the content as a Composite tile.
version	uint32	The version of the Composite format. It is currently 1.
byteLength	uint32	The length of the entire Composite tile, including this header and each inner tile, in bytes.
tilesLength	uint32	The number of tiles in the Composite.

## 7.8.4. Inner tiles

Inner tile fields are stored tightly packed immediately following the header section. The following information describes general characteristics of all tile formats that a Composite tile reader might exploit to find the boundaries of the inner tiles:

- Each tile starts with a 4-byte ANSI string, `magic`, that can be used to determine the tile format for further parsing. See tile format specifications for a list of possible formats. Composite tiles can contain Composite tiles.
- Each tile's header contains a `uint32 byteLength`, which defines the length of the inner tile, including its header, in bytes. This can be used to traverse the inner tiles.
- For any tile format's version 1, the first 12 bytes of all tiles is the following fields:

**Table 20 – Common header fields for tile formats**

FIELD NAME	DATA TYPE	DESCRIPTION
<code>magic</code>	4-byte ANSI string	Indicates the tile format
<code>version</code>	<code>uint32</code>	1
<code>byteLength</code>	<code>uint32</code>	Length, in bytes, of the entire tile.

Refer to the specification for each tile format for more details.

### 7.8.5. File extension and media type

Composite tiles use the `.cmpt` extension and `application/octet-stream` media type.

An explicit file extension is optional. Valid implementations may ignore it and identify a content's format by the `magic` field in its header.

### 7.8.6. Implementation examples

*This section is informative*

- [Python packcmpt tool in gltf2glb toolset](#) contains code for combining one or more *Batched 3D Model* or *Instanced 3D Model* tiles into a single Composite tile file.
- Code for reading the header can be found in [Composite3DTileContent.js](#) in the CesiumJS implementation of 3D Tiles.

8

# IMPLICIT TILING

---

## 8.1. Overview

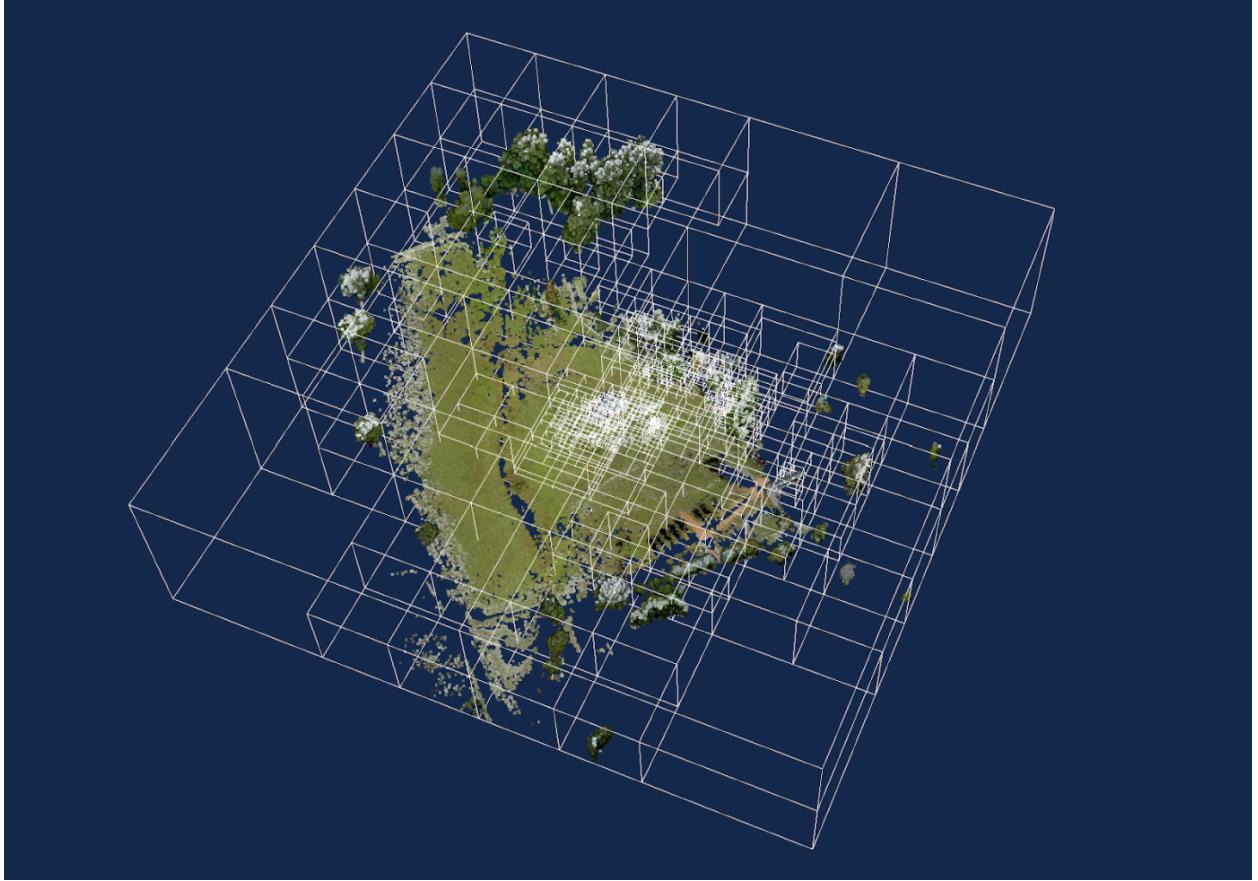
---

Implicit tiling defines a concise representation of quadtrees and octrees in 3D Tiles. This regular pattern allows for random access of tiles based on their tile coordinates which enables accelerated spatial queries, new traversal algorithms, and efficient updates of tile content, among other use cases.

Implicit tiling also allows for better interoperability with existing GIS data formats with implicitly defined tiling schemes. Some examples are [TMS](#), [WMTS](#), [S2](#), and [CDB](#).

In order to support sparse datasets, **availability** data determines which tiles exist. To support massive datasets, availability is partitioned into fixed-size **subtrees**. Subtrees may store **metadata** for available tiles and contents.

An `implicitTiling` object may be added to any tile in the tilesheet JSON. The object defines how the tile is subdivided and where to locate content resources. It may be added to multiple tiles to create more complex subdivision schemes.



**Figure 44** – A point cloud organized into a sparse octree. Data source: Trimble

## 8.2. Implicit Root Tile

---

An `implicitTiling` object may be added to any tile in the tiles JSON. Such a tile is called an **implicit root tile**, to distinguish it from the root tile of the tiles JSON.

```
{  
  "root": {  
    "boundingVolume": {  
      "region": [-1.318, 0.697, -1.319, 0.698, 0, 20]  
    },  
    "refine": "REPLACE",  
    "geometricError": 5000,  
    "content": {  
      "uri": "content/{level}/{x}/{y}.glb"  
    },  
    "implicitTiling": {  
      "subdivisionScheme": "QUADTREE",  
      "availableLevels": 21,  
      "subtreeLevels": 7,  
      "subtrees": {  
        "uri": "subtrees/{level}/{x}/{y}.json"  
      }  
    }  
  }  
}
```

```
    }  
}
```

The `implicitTiling` object has the following properties:

**Table 21** – Properties of implicit tiling object

PROPERTY	DESCRIPTION
<code>subdivisionScheme</code>	Either QUADTREE or OCTREE. See Subdivision scheme.
<code>availableLevels</code>	How many levels there are in the tree with available tiles.
<code>subtreeLevels</code>	How many levels there are in each subtree.
<code>subtrees</code>	Template URI for subtree files. See Subtrees.

Template URIs are used for locating subtree files as well as tile contents. For content, the template URI is specified in the tile's `content.uri` property.

The following constraints apply to implicit root tiles:

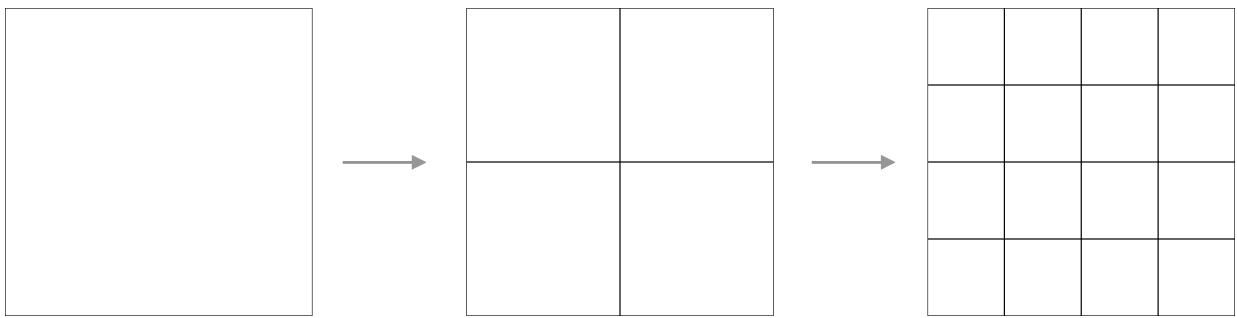
- The tile shall omit the `children` property
- The tile shall omit the `metadata` property
- The `content.uri` shall not point to an external tileset
- The `content` shall omit the `boundingVolume` property

## 8.3. Subdivision Scheme

---

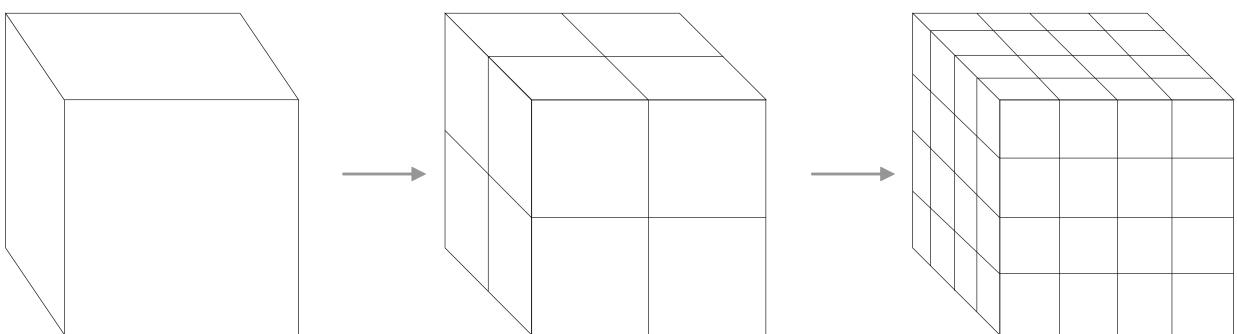
A **subdivision scheme** is a recursive pattern for dividing a bounding volume of a tile into smaller children tiles that take up the same space.

A **quadtree** divides space only on the x and y dimensions. It divides each tile into 4 smaller tiles where the x and y dimensions are halved. The quadtree z minimum and maximum remain unchanged. The resulting tree has 4 children per tile.



**Figure 45 – Subdivision in a quadtree**

An **octree** divides space along all 3 dimensions. It divides each tile into 8 smaller tiles where each dimension is halved. The resulting tree has 8 children per tile.



**Figure 46 – Subdivision in an octree**

For a region bounding volume, x, y, and z refer to longitude, latitude, and height respectively.

Sphere bounding volumes are disallowed, as these cannot be divided into a quadtree or octree.

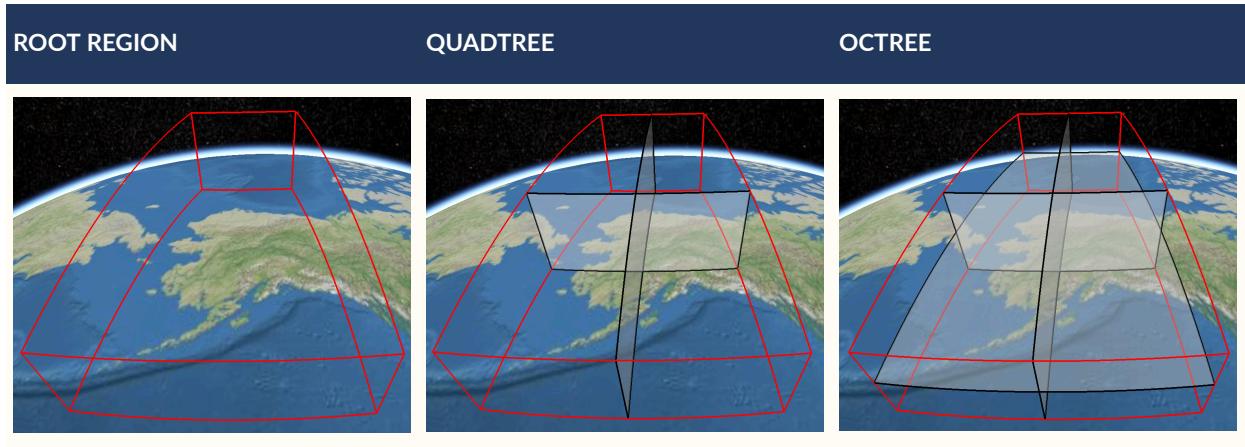
For subdivision of S2 bounding volumes refer to [3DTILES\\_bounding\\_volume\\_S2](#).

The following diagrams illustrate the subdivision in the bounding volume types supported by 3D Tiles:

**Table 22 – Subdivision for bounding boxes**

ROOT BOX	QUADTREE	OCTREE

**Table 23** – Subdivision for bounding regions



### 8.3.1. Subdivision Rules

Implicit tiling only requires defining the subdivision scheme, refinement strategy, bounding volume, and geometric error at the implicit root tile. For descendant tiles, these properties are computed automatically, based on the following rules:

**Table 24** – Subdivision rules for implicit tiling

PROPERTY	SUBDIVISION RULE
subdivisionScheme	Constant for all descendant tiles
refine	Constant for all descendant tiles
boundingVolume	Divided into four or eight parts depending on the subdivisionScheme
geometricError	Each child's geometricError is half of its parent's geometricError

NOTE In order to maintain numerical stability during this subdivision process, the actual bounding volumes should not be computed progressively by subdividing a non-root tile volume. Instead, the exact bounding volumes should be computed directly for a given level. Let the extent of the root bounding volume along one dimension  $d$  be  $(min_d, max_d)$ . The number of bounding volumes along that dimension for a given level is  $2^{level}$ . The size of each bounding volume at this level, along dimension  $d$ , is  $size_d = (max_d - min_d) / 2^{level}$ . The extent of the bounding volume of a child can then be computed directly as  $(min_d + size_d * i, min_d + size_d * (i + 1))$ , where  $i$  is the index of the child in dimension  $d$ .

The computed tile boundingVolume and geometricError can be overridden with tile metadata, if desired. Content bounding volumes are not computed automatically but they may be provided by content metadata. Tile and content bounding volumes shall maintain spatial coherence.

## 8.4. Tile Coordinates

**Tile coordinates** are a tuple of integers that uniquely identify a tile. Tile coordinates are either (level, x, y) for quadtrees or (level, x, y, z) for octrees. All tile coordinates are 0-indexed.

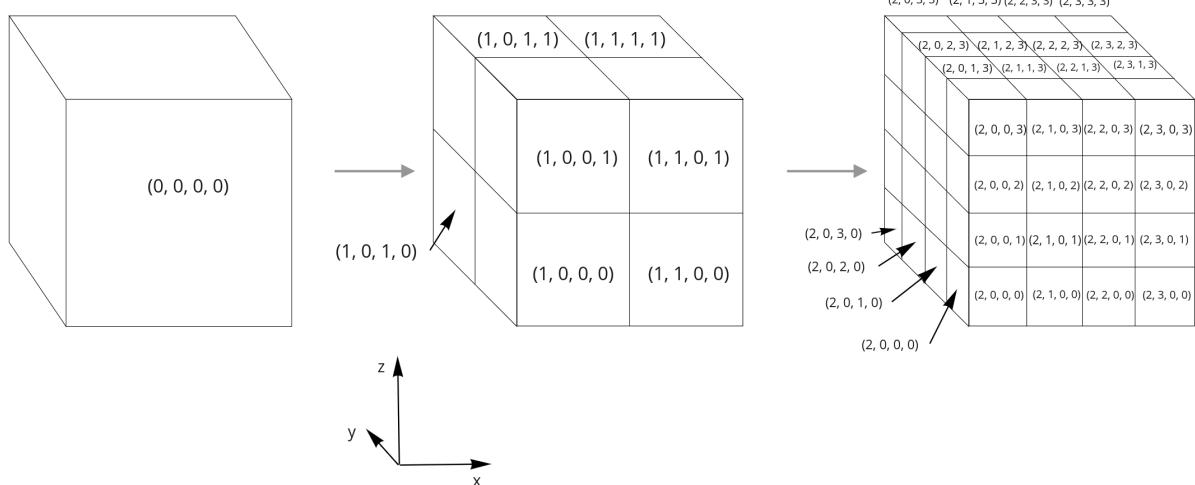
level is 0 for the implicit root tile. This tile's children are at level 1, and so on.

x, y, and z coordinates define the location of the tile within the level.

For box bounding volumes:

**Table 25 – Tile coordinates for bounding regions**

COORDINATE	POSITIVE DIRECTION
x	Along the +x axis of the bounding box
y	Along the +y axis of the bounding box
z	Along the +z axis of the bounding box

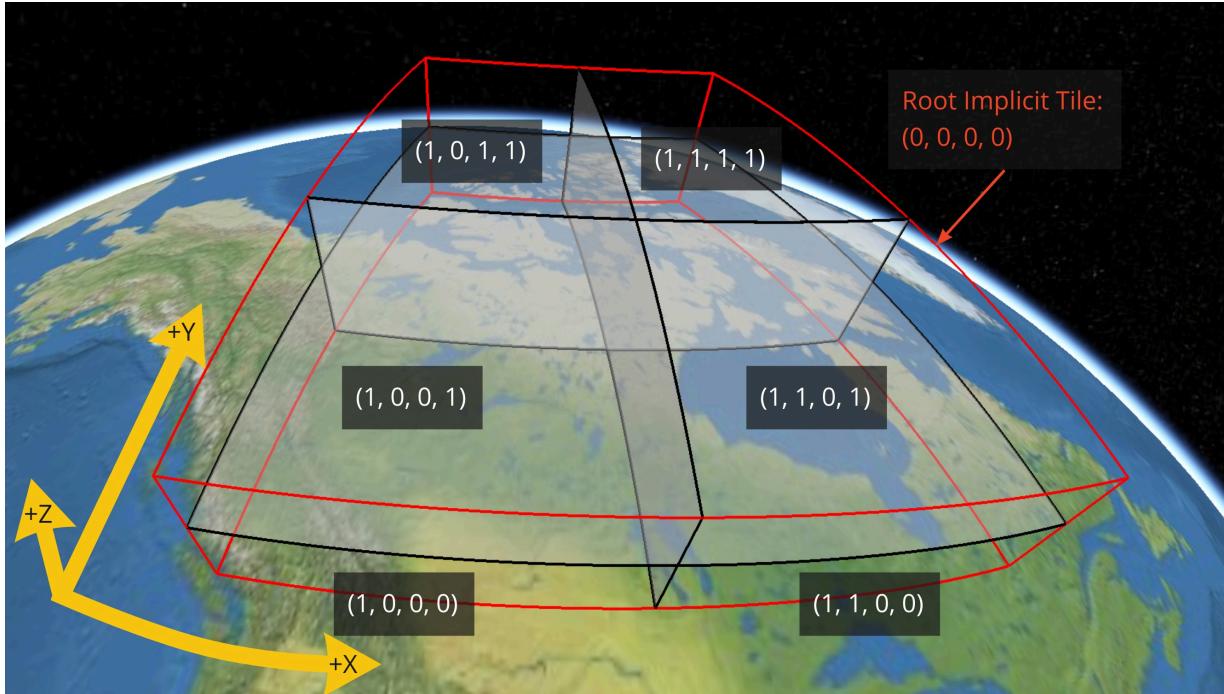


**Figure 47 – Coordinates of an octree node with a bounding box, and its child nodes**

For region bounding volumes:

**Table 26 – Tile coordinates for bounding regions**

COORDINATE	POSITIVE DIRECTION
x	From west to east (increasing longitude)
y	From south to north (increasing latitude)
z	From bottom to top (increasing height)



**Figure 48 – Coordinates of an octree node with a bounding region, and its child nodes**

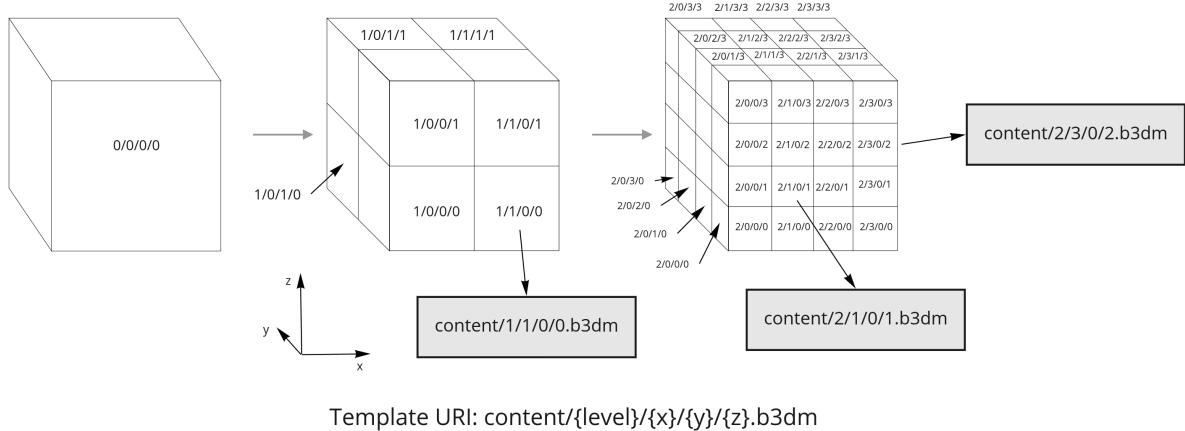
## 8.5. Template URIs

---

A **Template URI** is a URI pattern used to refer to tiles by their tile coordinates.

Template URIs shall include the variables `{level}`, `{x}`, `{y}`. Template URIs for octrees shall also include `{z}`. When referring to a specific tile, the tile's coordinates are substituted for these variables.

Template URIs, when given as relative paths, are resolved relative to the tileset JSON file.



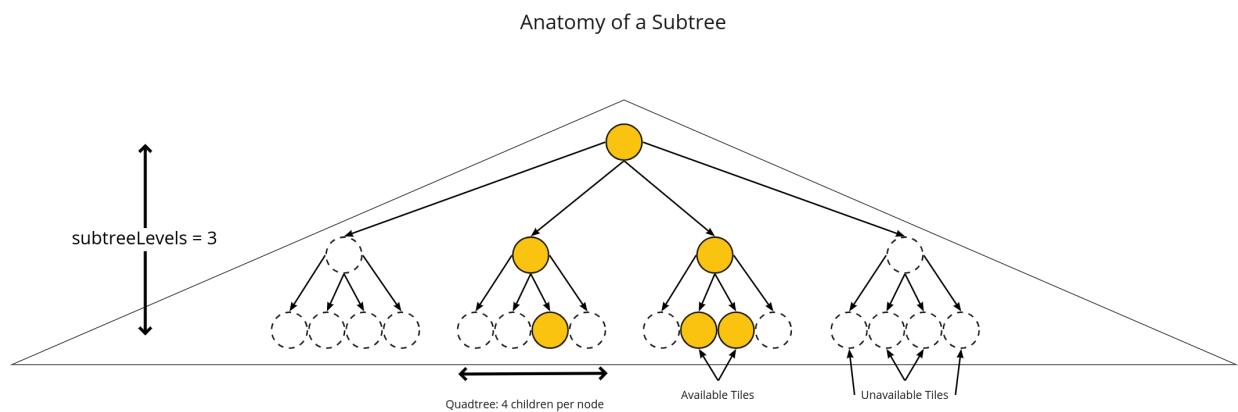
**Figure 49 – Examples of template URIs to identify the content for implicit tiles**

## 8.6. Subtrees

---

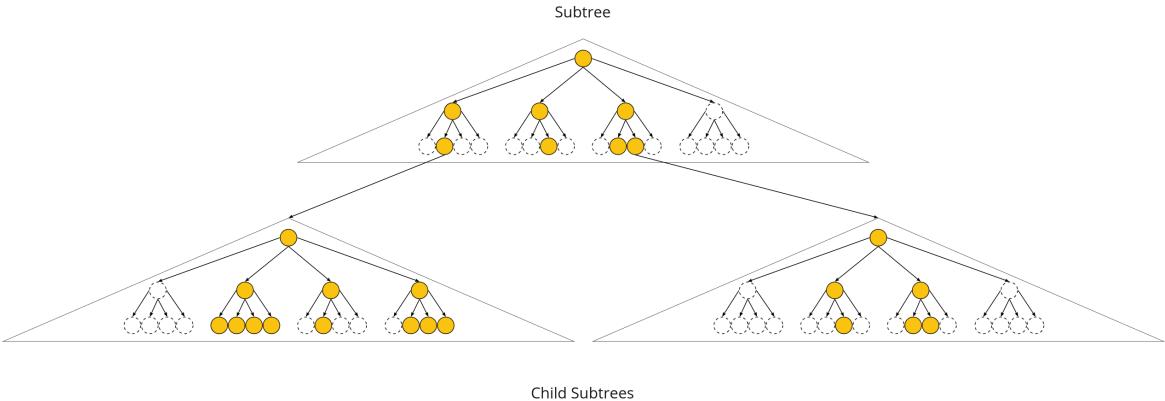
In order to support sparse datasets, additional information is needed to indicate which tiles or contents exist. This is called **availability**.

**Subtrees** are fixed size sections of the tileset tree used for storing availability. The tileset is partitioned into subtrees to bound the size of each availability buffer for optimal network transfer and caching. The `subtreeLevels` property defines the number of levels in each subtree. The subdivision scheme determines the number of children per tile.



**Figure 50 – The structure of a subtree for implicit tiling**

After partitioning a tileset into subtrees, the result is a tree of subtrees.



**Figure 51 – A tree of subtrees representing an implicit tileset**

### 8.6.1. Availability

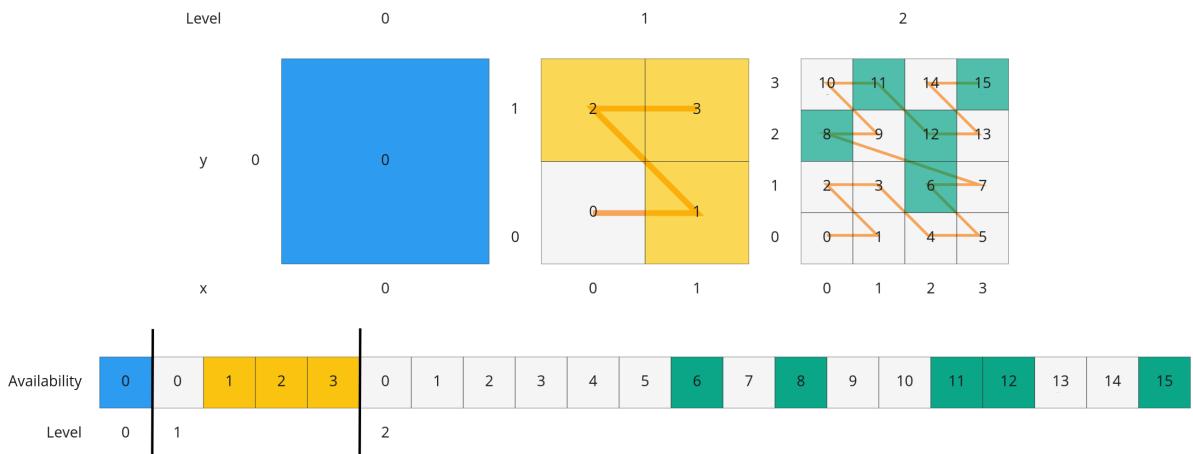
Each subtree contains tile availability, content availability, and child subtree availability.

- **Tile availability** indicates which tiles exist within the subtree
- **Content availability** indicates which tiles have associated content resources
- **Child subtree availability** indicates what subtrees are reachable from this subtree

Each type of availability is represented as a separate bitstream. Each bitstream is a 1D array where each element represents a node in the quadtree or octree. A 1 bit indicates that the element is available, while a 0 bit indicates that the element is unavailable. Alternatively, if all the bits in a bitstream are the same, a single constant value can be used instead.

To form the 1D bitstream, the tiles are ordered with the following rules:

- Within each level of the subtree, the tiles are ordered using the [Morton Z-order curve](#)
- The bits for each level are concatenated into a single bitstream



**Figure 52** – The computation of indices for accessing an availability bitstream, based on the coordinates of implicit tiles

In the diagram above, colored cells represent 1 bits, grey cells represent 0 bits.

Storing tiles in Morton order provides these benefits:

- Efficient indexing — The Morton index for a tile is computed in constant time by interleaving bits.
- Efficient traversal — The Morton index for a parent or child tile are computed in constant time by removing or adding bits, respectively.
- Locality of reference — Consecutive tiles are near to each other in 3D space.
- Better Compression — Locality of reference leads to better compression of availability bitstreams.

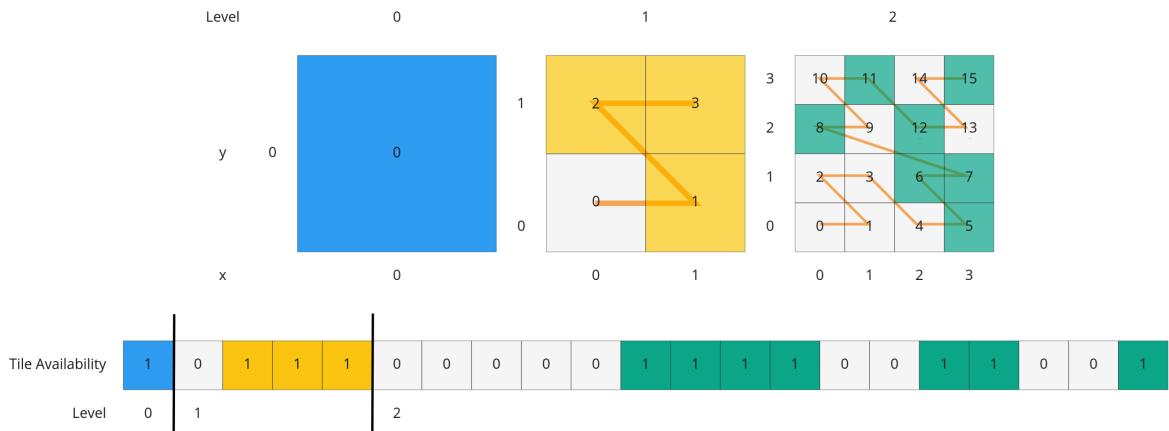
For more detailed information about working with Morton indices and availability bitstreams, see Availability Indexing.

### 8.6.1.1. Tile Availability

Tile availability determines which tiles exist in a subtree.

Tile availability has the following restrictions:

- If a non-root tile's availability is 1, its parent tile's availability shall also be 1.
- A subtree shall have at least one available tile.



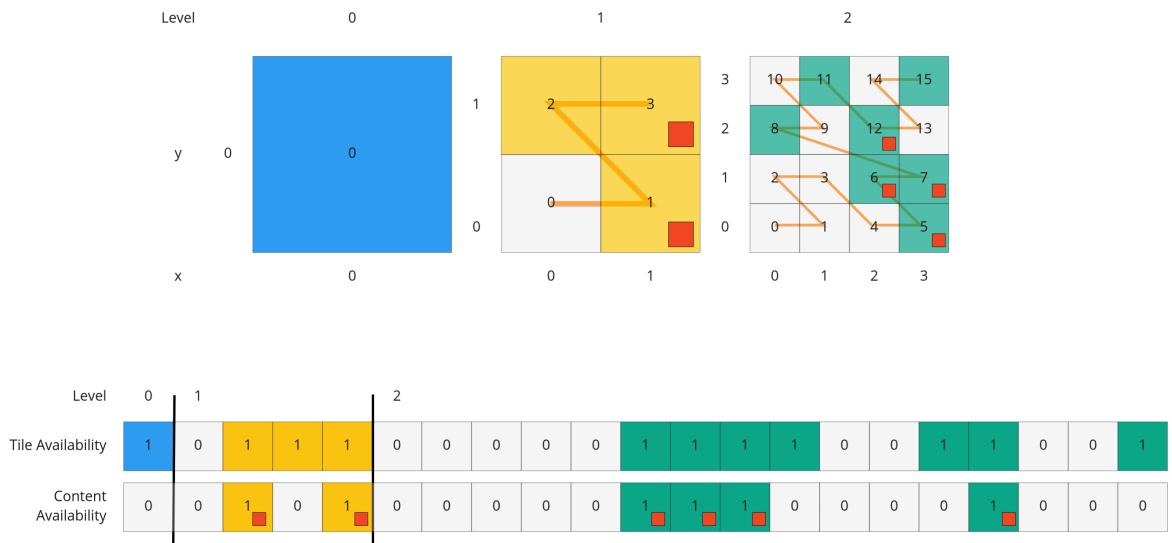
**Figure 53 – Illustration of a tile availability bitstream.** Tiles that are available are represented with a 1 in the bitstream.

### 8.6.1.2. Content Availability

Content availability determines which tiles have a content resource. The content resource is located using the `content.uri` template URI. If there are no tiles with a content resource, `tile.content` shall be omitted.

Content availability has the following restrictions:

- If content availability is 1 its corresponding tile availability shall also be 1. Otherwise, it would be possible to specify content files that are not reachable by the tiles of the tileset.
- If content availability is 0 and its corresponding tile availability is 1 then the tile is considered to be an empty tile.

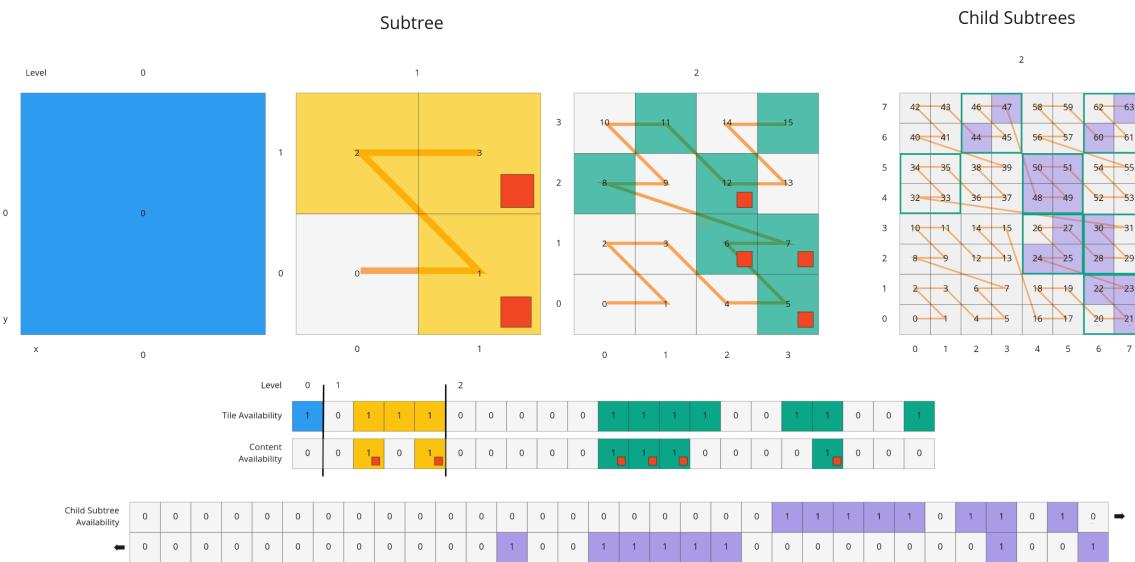


**Figure 54 — Illustration of a content availability bitstream.** Tiles that have associated content are represented with a 1 in the bitstream.

### 8.6.1.3. Child Subtree Availability

Child subtree availability determines which subtrees are reachable from the deepest level of this subtree. This links subtrees together to form a tree.

Unlike tile and content availability, which store bits for every level in the subtree, child subtree availability stores bits for nodes one level deeper than the deepest level of the subtree, and represent the root nodes of child subtrees. This is used to determine which other subtrees are reachable before requesting tiles. If availability is 0 for all child subtrees, then the tileset does not subdivide further.



**Figure 55** – Illustration of a child subtree availability bitstream. Tiles that are the roots of available subtrees are represented by a 1 in the bitstream.

## 8.6.2. Metadata

Subtrees may store metadata at multiple granularities.

- **Tile metadata** – metadata for available tiles in the subtree
- **Content metadata** – metadata for available content in the subtree
- **Subtree metadata** – metadata about the subtree as a whole

### 8.6.2.1. Tile Metadata

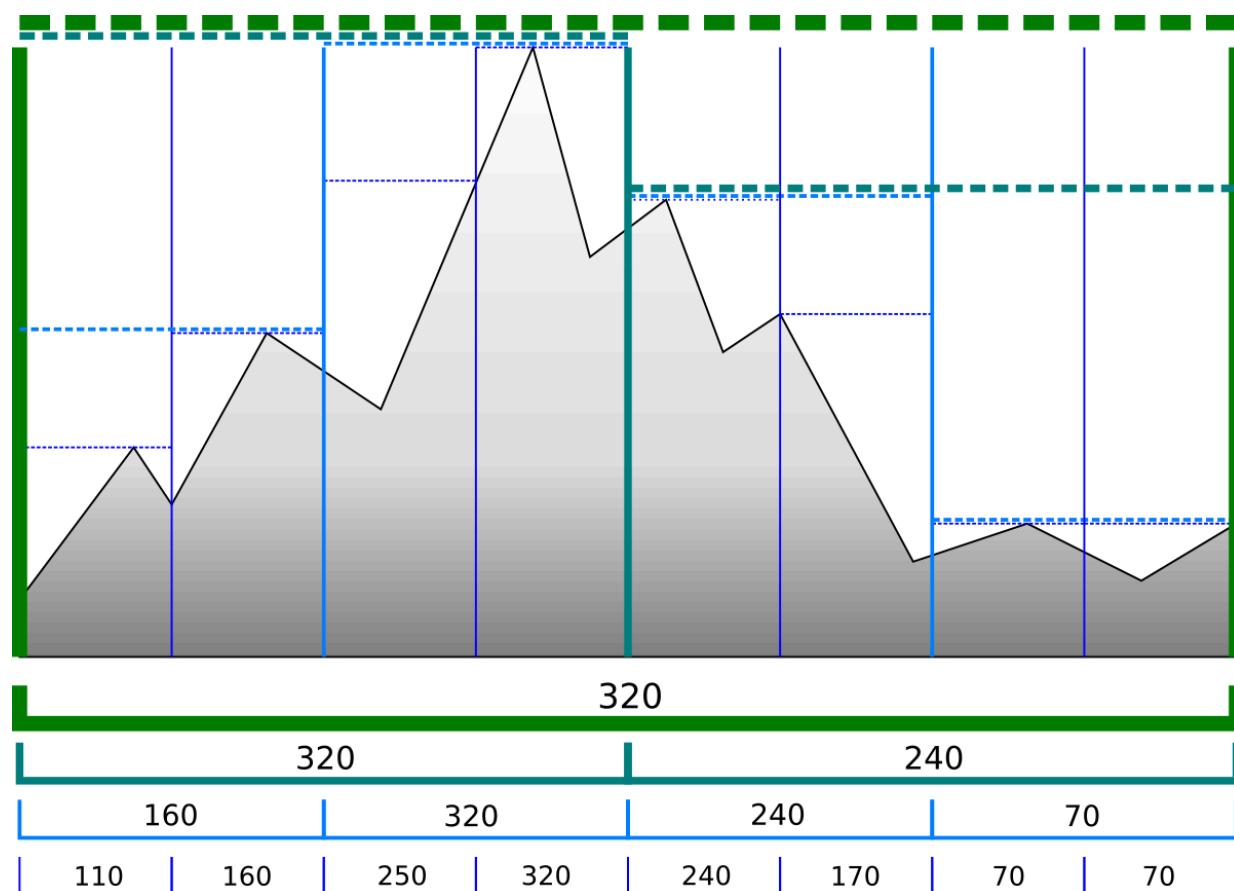
When tiles are listed explicitly within a tileset, each tile's metadata is also embedded explicitly within the tile definition. When the tile hierarchy is *implicit*, as enabled by implicit tiling, tiles are not listed exhaustively and metadata cannot be directly embedded in tile definitions. To support metadata for tiles within implicit tiling schemes, property values for all available tiles in a subtree are encoded in a property table. The binary representation is particularly efficient for larger datasets with many tiles.

Tile metadata exists only for available tiles and is tightly packed by an increasing tile index according to the Availability Ordering. Each available tile shall have a value – representation of missing values within a tile is possible only with the noData indicator defined by the *Binary Table Format* specification.

**NOTE 1**To determine the index into a property value array for a particular tile, count the number of available tiles occurring before that index, according to the tile Availability Ordering. If  $i$  available tiles occur before a particular tile, that tile's property values are stored at index  $i$  of each property value array. These indices may be precomputed for all available tiles, as a single pass over the subtree availability buffer.

Tile properties can have Semantics which define how property values should be interpreted. In particular, TILE\_BOUNDING\_BOX, TILE\_BOUNDING\_REGION, TILE\_BOUNDING\_SPHERE, TILE\_MINIMUM\_HEIGHT, and TILE\_MAXIMUM\_HEIGHT semantics each define a more specific bounding volume for a tile than is implicitly calculated from implicit tiling. If more than one of these semantics are available for a tile, clients may select the most appropriate option based on use case and performance requirements.

**NOTE 2**The following diagram shows how tile height semantics may be used to define tighter bounding regions for an implicit tilesset: The overall height of the bounding region of the whole tilesset is 320. The bounding regions for the child tiles will be computed by splitting the bounding regions of the respective parent tile at its center. By default, the height will remain constant. By storing the *actual* height of the contents in the respective region, and providing it as the `TILE_MAXIMUM_HEIGHT` for each available tile, it is possible to define the tightest-fitting bounding region for each level.



**Figure 56** – Illustration of storing the actual heights of individual tiles using the TILE MAXIMUM HEIGHT semantic

The `TILE_GEOMETRIC_ERROR` semantic allows tiles to provide a geometric error that overrides the implicitly computed geometric error.

### 8.6.2.2. Content Metadata

Subtrees may also store metadata for tile content. Content metadata exists only for available content and is tightly packed by increasing tile index. Binary property values are encoded in a compact *Binary Table Format* defined by the 3D Metadata Specification and are stored in a property table. If the implicit root tile has multiple contents then content metadata is stored in multiple property tables.

Content bounding volumes are not computed automatically by implicit tiling but may be provided by properties with semantics `CONTENT_BOUNDING_BOX`, `CONTENT_BOUNDING_REGION`, `CONTENT_BOUNDING_SPHERE`, `CONTENT_MINIMUM_HEIGHT`, and `CONTENT_MAXIMUM_HEIGHT`.

If the tile content is assigned to a group then all contents in the implicit tree are assigned to that group.

### 8.6.2.3. Subtree Metadata

Properties assigned to subtrees provide metadata about the subtree as a whole. Subtree metadata is encoded in JSON according to the JSON Format specification.

## 8.7. Subtree JSON Format

---

*Defined in `subtree.schema.json`.*

A **subtree file** is a JSON file that contains availability and metadata information for a single subtree. A subtree may reference external files containing binary data. An alternative Binary Format allows the JSON and binary data to be embedded into a single binary file.

### 8.7.1. Buffers and Buffer Views

The property table defines the storage of metadata in a binary form based on *buffer views* that are parts of a *buffer*.

A **buffer** is a binary blob. Each buffer has a `uri` that refers to an external file containing buffer data and a `byteLength` describing the buffer size in bytes. Relative paths are relative to the subtree file. Data URIs are not allowed.

In the Binary Format the first buffer may instead refer to the binary chunk of the subtree file, in which case the `uri` property shall be undefined. This buffer is referred to as the *internal buffer*.

A **buffer view** is a contiguous subset of a buffer. A buffer view's `buffer` property is an integer index to identify the buffer. A buffer view has a `byteOffset` and a `byteLength` to describe the

range of bytes within the buffer. The byteLength does not include any padding. There may be multiple buffer views referencing a single buffer.

For efficient memory access, the byteOffset of a buffer view shall be aligned to a multiple of 8 bytes.

## 8.7.2. Availability

Tile availability (`tileAvailability`) and child subtree availability (`childSubtreeAvailability`) shall always be provided for a subtree.

Content availability (`contentAvailability`) is an array of content availability objects. If the implicit root tile has a single content this array will have one element; if the tile has multiple contents this array will have multiple elements. If the implicit root tile does not have content then `contentAvailability` shall be omitted.

Availability may be represented either as a bitstream or a constant value. `bitstream` is an integer index that identifies the buffer view containing the availability bitstream. `constant` is an integer indicating whether all of the elements are available (1) or all are unavailable (0). `availableCount` is an integer indicating how many 1 bits exist in the availability bitstream.

Availability bitstreams are packed in binary using the format described in the Booleans section of the 3D Metadata Specification.

**NOTE**The JSON description of a subtree where each tile is available, but not all tiles have content, and not all child subtrees are available:

```
{  
    "buffers": [  
        {  
            "name": "Internal Buffer",  
            "byteLength": 16  
        },  
        {  
            "name": "External Buffer",  
            "uri": "external.bin",  
            "byteLength": 32  
        }  
    ],  
    "bufferViews": [  
        {  
            "buffer": 0,  
            "byteOffset": 0,  
            "byteLength": 11  
        },  
        {  
            "buffer": 1,  
            "byteOffset": 0,  
            "byteLength": 32  
        }  
    ],  
    "tileAvailability": {  
        "constant": 1,  
    },  
    "contentAvailability": [{  
        "bitstream": 0,  
        "availableCount": 60  
    }],  
    "childSubtreeAvailability": {  
        "bitstream": 1  
    }  
}
```

```
}
```

The tile availability can be encoded by setting `tileAvailability.constant` to 1, without needing an explicit bitstream, because all tiles in the subtree are available.

Only some tiles have content, and `contentAvailability.bufferView` indicates where the bitstream for the content availability is stored: The `bufferView` with index 0 refers to the buffer with index 0. This buffer does not have a `uri` property, and therefore refers to the *internal* buffer that is stored directly in the binary chunk of the subtree binary file. The `byteOffset` and `byteLength` indicate that the content availability bitstream is stored in the bytes [0...11) of the internal buffer.

Some child subtrees exist, so `childSubtreeAvailability.bufferView` refers to another bitstream. The `bufferView` with index 1 refers to the buffer with index 1. This buffer has a `uri` property, indicating that this second bitstream is stored in an external binary file.

### 8.7.3. Metadata

Subtrees may store metadata at multiple granularities. `tileMetadata` is a property table containing metadata for available tiles. `contentMetadata` is an array of property tables containing metadata for available content. If the implicit root tile has a single content this array will have one element; if the tile has multiple contents then this array will have multiple elements. If the implicit root tile does not have content then `contentMetadata` shall be omitted.

Subtree metadata (`subtreeMetadata`) is encoded in JSON according to the JSON Format specification.

**NOTE** The same JSON description of a subtree extended with tile, content, and subtree metadata. The subtree JSON refers to a class ID in the root tilesset schema. Tile and content metadata is stored in property table; subtree metadata is encoded directly in JSON.

*Schema in the root tilesset JSON*

```
{
  "schema": {
    "classes": {
      "tile": {
        "properties": {
          "horizonOcclusionPoint": {
            "semantic": "TITLE_HORIZON_OCCLUSION_POINT",
            "type": "VEC3",
            "componentType": "FLOAT64"
          },
          "countries": {
            "description": "Countries a tile intersects",
            "type": "STRING",
            "array": true
          }
        }
      },
      "content": {
        "properties": {
          "attributionIds": {
            "semantic": "ATTRIBUTION_IDS",
            "type": "SCALAR",
            "array": true
          }
        }
      }
    }
  }
}
```

```
        "componentType": "UINT16",
        "array": true
    },
    "minimumHeight": {
        "semantic": "CONTENT_MINIMUM_HEIGHT",
        "type": "SCALAR",
        "componentType": "FLOAT64"
    },
    "maximumHeight": {
        "semantic": "CONTENT_MAXIMUM_HEIGHT",
        "type": "SCALAR",
        "componentType": "FLOAT64"
    },
    "triangleCount": {
        "type": "SCALAR",
        "componentType": "UINT32"
    }
}
},
"subtree": {
    "properties": {
        "attributionStrings": {
            "semantic": "ATTRIBUTION_STRINGS",
            "type": "STRING",
            "array": true
        }
    }
}
}
```

## Subtree JSON

```
{  
  "buffers": [  
    {  
      "name": "Availability Buffer",  
      "uri": "availability.bin",  
      "byteLength": 48  
    },  
    {  
      "name": "Metadata Buffer",  
      "uri": "metadata.bin",  
      "byteLength": 6512  
    }  
,  
  ],  
  "bufferViews": [  
    { "buffer": 0, "byteOffset": 0, "byteLength": 11 },  
    { "buffer": 0, "byteOffset": 16, "byteLength": 32 },  
    { "buffer": 1, "byteOffset": 0, "byteLength": 2040 },  
    { "buffer": 1, "byteOffset": 2040, "byteLength": 1530 },  
    { "buffer": 1, "byteOffset": 3576, "byteLength": 344 },  
    { "buffer": 1, "byteOffset": 3920, "byteLength": 1024 },  
    { "buffer": 1, "byteOffset": 4944, "byteLength": 240 },  
    { "buffer": 1, "byteOffset": 5184, "byteLength": 122 },  
    { "buffer": 1, "byteOffset": 5312, "byteLength": 480 },  
    { "buffer": 1, "byteOffset": 5792, "byteLength": 480 },  
    { "buffer": 1, "byteOffset": 6272, "byteLength": 240 }  
,  
  "propertyTables": [  
    {  
      "class": "tile",  
      "count": 85,  
      "properties": {  
        "horizonOcclusionPoint": {  
          "values": 2  
        }  
      }  
    }  
  ]  
}
```

```

        "countries": {
            "values": 3,
            "arrayOffsets": 4,
            "stringOffsets": 5,
            "arrayOffsetType": "UINT32",
            "stringOffsetType": "UINT32"
        }
    },
    {
        "class": "content",
        "count": 60,
        "properties": {
            "attributionIds": {
                "values": 6,
                "arrayOffsets": 7,
                "arrayOffsetType": "UINT16"
            },
            "minimumHeight": {
                "values": 8
            },
            "maximumHeight": {
                "values": 9
            },
            "triangleCount": {
                "values": 10,
                "min": 520,
                "max": 31902
            }
        }
    }
],
"tileAvailability": {
    "constant": 1
},
"contentAvailability": [
    {
        "bitstream": 0,
        "availableCount": 60
    }
],
"childSubtreeAvailability": {
    "bitstream": 1
},
"tileMetadata": 0,
"contentMetadata": [1],
"subtreeMetadata": {
    "class": "subtree",
    "properties": {
        "attributionStrings": [
            "Source A",
            "Source B",
            "Source C",
            "Source D"
        ]
    }
}
}

```

#### 8.7.4. Multiple Contents

When the implicit root tile has multiple contents then `contentAvailability` and `contentMetadata` are provided for each content layer.

**NOTE** JSON description of a subtree with multiple contents. In this example all tiles are available, all building contents are available, and only some tree contents are available.

### *Implicit root tile*

```
{  
  "root": {  
    "boundingVolume": {  
      "region": [-1.318, 0.697, -1.319, 0.698, 0, 20]  
    },  
    "refine": "ADD",  
    "geometricError": 5000,  
    "contents": [  
      {  
        "uri": "buildings/{level}/{x}/{y}.glb",  
      },  
      {  
        "uri": "trees/{level}/{x}/{y}.glb",  
      }  
    ],  
    "implicitTiling": {  
      "subdivisionScheme": "QUADTREE",  
      "availableLevels": 21,  
      "subtreeLevels": 7,  
      "subtrees": {  
        "uri": "subtrees/{level}/{x}/{y}.json"  
      }  
    }  
  }  
}
```

### *Subtree JSON*

```
{  
  "propertyTables": [  
    {  
      "class": "building",  
      "count": 85,  
      "properties": {  
        "height": {  
          "values": 1  
        },  
        "owners": {  
          "values": 2,  
          "arrayOffsets": 3,  
          "stringOffsets": 4  
        }  
      }  
    },  
    {  
      "class": "tree",  
      "count": 52,  
      "properties": {  
        "height": {  
          "values": 5  
        },  
        "species": {  
          "values": 6  
        }  
      }  
    }  
  ],  
  "tileAvailability": {  
    "constant": 1  
  },  
  "contentAvailability": [  
    {  
      "constant": 1  
    },  
    {  
      "constant": 1  
    }  
  ]  
}
```

```

        "bitstream": 0,
        "availableCount": 52
    }
],
"childSubtreeAvailability": {
    "constant": 1
},
"contentMetadata": [0, 1]
}

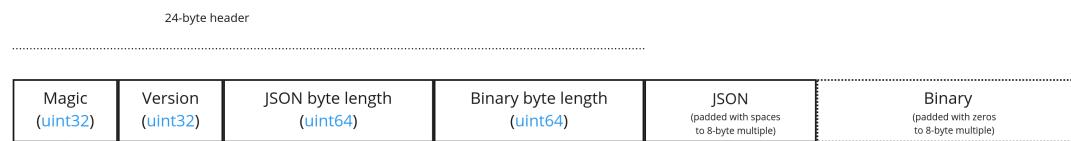
```

## 8.8. Subtree Binary Format

---

The subtree binary format is an alternative to the JSON file format that allows the JSON and binary data to be embedded into a single binary file.

The binary subtree format is little-endian and consists of a 24-byte header and a variable length payload:



**Figure 57 – Data layout for the subtree binary format**

Header fields:

**Table 27 – Header fields for subtree files**

BYTES	FIELD	TYPE	DESCRIPTION
0-3	Magic	UINT32	A magic number identifying this as a subtree file. This is always 0x74627573, the four bytes of the ASCII string <code>subt</code> stored in little-endian order.
4-7	Version	UINT32	The version number. Always 1 for this version of the specification.
8-15	JSON byte length	UINT64	The length of the subtree JSON, including any padding.
16-23	Binary byte length	UINT64	The length of the buffer (or 0 if the buffer does not exist) including any padding.

Each chunk shall be padded so it ends on an 8-byte boundary:

- The JSON chunk shall be padded with trailing Space chars (0x20)

- If it exists, the binary chunk shall be padded with trailing zeros (0x00)

9

# 3D METADATA SPECIFICATION

---

# 3D METADATA SPECIFICATION

## 9.1. Overview

The 3D Metadata Specification defines a standard format for structured metadata in 3D content. Metadata – represented as entities and properties – may be closely associated with parts of 3D content, with data representations appropriate for large, distributed datasets. For the most detailed use cases, properties allow vertex- and texel-level associations; higher-level property associations are also supported.

Many domains benefit from structured metadata – typical examples include historical details of buildings in a city, names of components in a CAD model, descriptions of regions on textured surfaces, and classification codes for point clouds.

The specification defines core concepts to be used by multiple 3D formats, and is language and format agnostic. This document defines concepts with purpose and terminology, but does not impose a particular schema or serialization format for implementation. For use of the format outside of abstract conceptual definitions, see:

- 3D Tiles Metadata – Assigns metadata to tilesets, tiles, and contents in 3D Tiles 1.1
- [3DTILES\\_metadata](#) – An extension for 3D Tiles 1.0 that assigns metadata to tilesets, tiles, and contents
- [EXT\\_structural\\_metadata \(glTF 2.0\)](#) – Assigns metadata to vertices, texels, and features in a glTF asset

The specification does not enumerate or define the semantic meanings of metadata, and assumes that separate specifications will define semantics for their particular application or domain. One example is the 3D Metadata Semantic Reference which defines built-in semantics for 3D Tiles and glTF. Identifiers for externally-defined semantics can be stored within the 3D Metadata Specification.

## 9.2. Concepts

This specification defines metadata schemas and methods for encoding metadata.

**Schemas** contain a set of **classes** and **enums**. A class represents a category of similar entities, and is defined as a set of **properties**. Each property describes values of a particular type. An enum defines a set of named values representing a single value type, and may be referenced by class properties. Schema definitions do not describe how entities or properties are stored,

and may be represented in a file format in various ways. Schemas can be reused across multiple assets or even file formats.

**Entities** are instantiations of a class, populated with **property values** conforming to the class definition. Every property value of an entity shall be defined by its class, and an entity shall not have extraneous property values. Properties of a class may be required, in which case all entities instantiating the class are required to include them.

**NOTE** Entities may be defined at various levels of abstraction. Within a large dataset, individual vertices or texels may represent entities with granular metadata properties. Vertices and texels may be organized into higher-order groups (such as meshes, scene graphs, or tilesets) having their own associated properties.

**Metadata**, as used throughout this specification, refers to any association of 3D content with entities and properties, such that entities represent meaningful units within an overall structure. Other common definitions of metadata, particularly in relation to filesystems or networking as opposed to 3D content, remain outside the scope of the document.

Property values are stored with flexible representations to allow compact transmission and efficient lookups. This specification defines two possible storage formats.

## 9.2.1. Identifiers

Throughout this specification, IDs (identifiers) are strings that match the regular expression `^[a-zA-Z_][a-zA-Z0-9_]*$`: Strings that consist of upper- or lowercase letters, digits, or underscores, starting with either a letter or an underscore. These strings should be camel case strings that are human-readable (wherever possible). When IDs subject to these restrictions are not sufficiently clear for human readers, applications should also provide a name for the structures that support dedicated names.

# 9.3. Schemas

---

## 9.3.1. Schema

A schema defines the organization and types of metadata used in 3D content, represented as a set of classes and enums. Class definitions are referenced by entities whose metadata conforms to the class definition. This provides a consistent and machine-readable structure for all entities in a dataset.

Components of a schema are listed below, and implementations may define additional components.

### **9.3.1.1. ID**

IDs (`id`) are unique identifiers for a schema.

### **9.3.1.2. Version**

Schema version (`version`) is an application-specific identifier for a given schema revision. Version shall be a string, and should be syntactically compatible with [SemVer](#).

When a schema has multiple versions, the (`id`, `version`) pair uniquely identifies a particular schema and revision.

**NOTE**Valid semantic versions include strings like `0.1.2`, `1.2.3`, and `1.2.3-alpha`.

### **9.3.1.3. Name**

Names (`name`) provide a human-readable label for a schema, and are not required to be unique. Names shall be valid Unicode strings, and should be written in natural language.

### **9.3.1.4. Description**

Descriptions (`description`) provide a human-readable explanation of a schema, its purpose, or its contents. Typically at least a phrase, and possibly several sentences or paragraphs. Descriptions shall be valid Unicode strings.

### **9.3.1.5. Enums**

Unordered set of enums.

### **9.3.1.6. Classes**

Unordered set of classes.

## **9.3.2. Enum**

An enum consists of a set of named values, represented as (`string`, `integer`) pairs. Each enum is identified by a unique ID.

**NOTE**A “species” enum with three possible tree species, as well as an “Unknown” value.

- **ID:** “species”

- **Name:** “Species”
- **Description:** “Common tree species identified in the study.”
- **Value type:** INT32

#### Names and values of an example enum

NAME	VALUE
“Oak”	0
“Pine”	1
“Maple”	2
“Unknown”	-1

#### 9.3.2.1. ID

IDs (`id`) are unique identifiers for an enum within a schema.

#### 9.3.2.2. Name

Names (`name`) provide a human-readable label for an enum, and are not required to be unique within a schema. Names shall be valid Unicode strings, and should be written in natural language.

#### 9.3.2.3. Description

Descriptions (`description`) provide a human-readable explanation of an enum, its purpose, or its contents. Typically at least a phrase, and possibly several sentences or paragraphs. Descriptions shall be valid Unicode strings.

#### 9.3.2.4. Values

An enum consists of a set of named values, represented as (`string`, `integer`) pairs. The following enum value types are supported: INT8, UINT8, INT16, UINT16, INT32, UINT32, INT64, and UINT64. See the Component Type section for definitions of each. Smaller enum types limit the range of possible enum values, and allow more efficient binary encoding. Duplicate names or values within the same enum are not allowed.

### **9.3.3. Class**

Classes represent categories of similar entities, and are defined by a collection of one or more properties shared by the entities of a class. Each class has a unique ID within the schema, and each property has a unique ID within the class, to be used for references within the schema and externally.

#### **9.3.3.1. ID**

IDs (`id`) are unique identifiers for a class within a schema.

#### **9.3.3.2. Name**

Names (`name`) provide a human-readable label for a class, and are not required to be unique within a schema. Names shall be valid Unicode strings, and should be written in natural language.

#### **9.3.3.3. Description**

Descriptions (`description`) provide a human-readable explanation of a class, its purpose, or its contents. Typically at least a phrase, and possibly several sentences or paragraphs. Descriptions shall be valid Unicode strings.

#### **9.3.3.4. Properties**

Unordered set of properties.

### **9.3.4. Property**

#### **9.3.4.1. Overview**

Properties describe the type and structure of values that may be associated with entities of a class. Entities may omit values for a property, unless the property is required. Entities shall not contain values other than those defined by the properties of their class.

**NOTE**The following example shows the basics of how classes describe the types of metadata. A building class describes the heights of various buildings in a dataset. Likewise, the tree class describes trees that have a height, species, and leaf color.

**building**

## Properties and types of a building

PROPERTY	TYPE	COMPONENTTYPE
height	SCALAR	FLOAT32

tree

## Properties and types of a tree

PROPERTY	TYPE	COMPONENTTYPE	ENUMTYPE
height	SCALAR	FLOAT32	
species	ENUM		species
leafColor	STRING		

### 9.3.4.2. ID

IDs (id) are unique identifiers for a property within a class.

### 9.3.4.3. Name

Names (name) provide a human-readable label for a property, and shall be unique to a property within a class. Names shall be valid Unicode strings, and should be written in natural language. Property names do not have inherent meaning; to provide such a meaning, a property shall also define a semantic.

**NOTE** A typical ID / Name pair, in English, would be localTemperature and "Local Temperature". In Japanese, the name might be represented as “きおん”. Because IDs are restricted to identifiers, use of helpful property names is essential for clarity in many languages.

### 9.3.4.4. Description

Descriptions (description) provide a human-readable explanation of a property, its purpose, or its contents. Typically at least a phrase, and possibly several sentences or paragraphs. Descriptions shall be valid Unicode strings. To provide a machine-readable semantic meaning, a property shall also define a semantic.

#### 9.3.4.5. Semantic

Property IDs, names, and descriptions do not have an inherent meaning. To provide a machine-readable meaning, properties may be assigned a semantic identifier string (semantic), indicating how the property's content should be interpreted. Semantic identifiers may be defined by the 3D Metadata Semantic Reference or by external semantic references, and may be application-specific. Identifiers should be uppercase, with underscores as word separators.

**NOTE**Semantic definitions might include temperature in degrees Celsius (e.g. TEMPERATURE\_DEGREES\_CELSIUS), time in milliseconds (e.g. TIME\_MILLISECONDS), or mean squared error (e.g. MEAN\_SQUARED\_ERROR). These examples are only illustrative.

#### 9.3.4.6. Type

A property's type (type) describes the structure of the value given for each entity.

**Table 28 – Types of metadata properties**

NAME	TYPE
SCALAR	Single numeric component
VEC2	Fixed-length vector with two (2) numeric components
VEC3	Fixed-length vector with three (3) numeric components
VEC4	Fixed-length vector with four (4) numeric components
MAT2	2×2 matrix with numeric components
MAT3	3×3 matrix with numeric components
MAT4	4×4 matrix with numeric components
STRING	A sequence of characters
BOOLEAN	True or false
ENUM	An enumerated type

#### 9.3.4.7. Component Type

Scalar, vector, and matrix types comprise of numeric components. Each component is an instance of the property's component type (`componentType`), with the following component types supported:

**Table 29** – Component types of metadata properties

NAME	COMPONENTTYPE
INT8	Signed integer in the range [-128, 127]
UINT8	Unsigned integer in the range [0, 255]
INT16	Signed integer in the range [-32768, 32767]
UINT16	Unsigned integer in the range [0, 65535]
INT32	Signed integer in the range [-2147483648, 2147483647]
UINT32	Unsigned integer in the range [0, 4294967295]
INT64	Signed integer in the range [-9223372036854775808, 9223372036854775807]
UINT64	Unsigned integer in the range [0, 18446744073709551615]
FLOAT32	A number that can be represented as a 32-bit IEEE floating point number
FLOAT64	A number that can be represented as a 64-bit IEEE floating point number

Floating-point properties (FLOAT32 and FLOAT64) shall not include values NaN, +Infinity, or -Infinity.

**NOTE** Developers of authoring tools should be aware that many JSON implementations support only numeric values that can be represented as IEEE-754 double precision floating point numbers. Floating point numbers should be representable as double precision IEEE-754 floats when encoded in JSON. When those numbers represent property values (such as `noData`, `min`, or `max`) having lower precision (e.g. single-precision float, 8-bit integer, or 16-bit integer), the values should be rounded to the same precision in JSON to avoid any potential mismatches. Numeric property values encoded in binary storage are unaffected by these limitations of JSON implementations.

#### 9.3.4.8. Enum Type

Enum properties are denoted by ENUM. An enum property shall additionally provide the ID of the specific enum it uses, referred to as its enum type (enumType).

#### 9.3.4.9. Arrays

A property can be declared to be a fixed- and variable-length array, consisting of elements of the given type. For fixed-length arrays, a count (count) denotes the number of elements in each array, and shall be greater than or equal to 2. Variable-length arrays do not define a count and may have any length, including zero.

#### 9.3.4.10. Normalized Values

Normalized properties (normalized) provide a compact alternative to larger floating-point types. Normalized values are stored as integers, but when accessed are transformed to floating-point according to the following equations:

**Table 30 – Conversion of component types**

COMPONENT	INT TO FLOAT	FLOAT TO INT
INT8	$f = \max(i / 127.0, -1.0)$	$i = \text{round}(f * 127.0)$
UINT8	$f = i / 255.0$	$i = \text{round}(f * 255.0)$
INT16	$f = \max(i / 32767.0, -1.0)$	$i = \text{round}(f * 32767.0)$
UINT16	$f = i / 65535.0$	$i = \text{round}(f * 65535.0)$
INT32	$f = \max(i / 2147483647.0, -1.0)$	$i = \text{round}(f * 2147483647.0)$
UINT32	$f = i / 4294967295.0$	$i = \text{round}(f * 4294967295.0)$
INT64	$f = \max(i / 9223372036854775807.0, -1.0)$	$i = \text{round}(f * 9223372036854775807.0)$
UINT64	$f = i / 18446744073709551615.0$	$i = \text{round}(f * 18446744073709551615.0)$

normalized is only applicable to scalar, vector, and matrix types with integer component types.

**NOTE**Depending on the implementation and the chosen integer type, there may be some loss of precision in values after denormalization. For example, if the implementation uses 32-bit

floating point variables to represent the value of a normalized 32-bit integer, there are only 23 bits in the mantissa of the float, and lower bits will be truncated by denormalization. Client implementations should use higher precision floats when appropriate for correctly representing the result.

#### 9.3.4.11. Offset and Scale

A property may declare an offset (offset) and scale (scale) to apply to property values. This is useful when mapping property values to a different range.

The offset and scale can be defined for types that either have a floating-point componentType, or when normalized is set to true. This applies to SCALAR, VECN, and MATN types, and to fixed-length arrays of these types. The structure of offset and scale is explained in the Property Values Structure section.

The following equation is used to transform the original property value into the actual value that is used by the client:

```
transformedValue = offset + scale * normalize(value)
```

These operations are applied component-wise, both for array elements and for vector and matrix components.

The transformation that is described here allows arbitrary source value ranges to be mapped to arbitrary target value ranges, by first computing the float value for the original normalized value, and then mapping that floating point range to the desired target range.

**NOTE** The result of transforming a normalized integer value into a floating point value may be lossy, as described in the section about Normalized Values. Depending on the range of property values, the values of offset and scale, and the floating point precision that is used in the client implementation, the computation may cause low-significance bits to be truncated from the final result. Client implementations should retain as much precision as reasonably possible.

When the offset for a property is not given, then it is assumed to be 0 for each component of the respective type. When the scale value of a property is not given, then it is assumed to be 1 for each component of the respective type. Instances of the class that defines the respective property can override the offset- and scale factors, to account for the actual range of property values that are provided by the instance.

#### 9.3.4.12. Minimum and Maximum Values

Properties may specify a minimum (min) and maximum (max) value. Minimum and maximum values represent component-wise bounds of the valid range of values for a property. Both values are *inclusive*, meaning that they denote the smallest and largest allowed value, respectively.

The min and max value can be defined for SCALAR, VECN, and MATN types with numeric component types, and for fixed-length arrays of these types. The structure of min and max is explained in the Property Values Structure section.

For properties that are normalized, the component type of `min` and `max` is a floating point type. Their values represent the bounds of the final, transformed property values. This includes the normalization and offset- or scale computations, as well as other transforms or constraints that are not part of the class definition itself: A normalized unsigned value is in the range [0.0, 1.0] after the normalization has been applied, but `[min, max]` may specify a different value range.

For all other properties, the component type of `min` and `max` matches the `componentType` of the property, and the values are the bounds of the original property values.

**NOTE**A property storing GPS coordinates might define a range of [-180, 180] degrees for longitude values and [-90, 90] degrees for latitude values.

Property values outside the `[minimum, maximum]` range are not allowed, with the exception of `noData` values.

#### 9.3.4.13. Required Properties, No Data Values, and Default Values

When associated property values shall exist for all entities of a class, a property is considered required (required).

Individual elements in an array or individual components in a vector or matrix cannot be marked as required; only the property itself can be marked as required.

Properties may optionally specify a No Data value (`noData`, or “sentinel value”) to be used when property values do not exist. A `noData` value may be provided for any type except `BOOLEAN`. For `ENUM` types, a `noData` value should contain the name of the enum value as a string, rather than its integer value. The structure of the `noData` value is explained in the Property Values Structure section.

A `noData` value is especially useful when only some entities in a property table are missing property values (see Binary Table Format). Otherwise if all entities are missing property values the column may be omitted from the table and a `noData` value need not be provided. Entities encoded in the JSON Format may omit the property instead of providing a `noData` value. `noData` values and omitted properties are functionally equivalent.

A default value (`default`) may be provided for missing property values. For `ENUM` types, a `default` value should contain the name of the enum value as a string, rather than its integer value. For all other cases, the structure of the `default` value is explained in the Property Values Structure section.

If a `default` value is not provided, the behavior when encountering missing property values is implementation-defined.

**NOTE**In the example below, a “tree” class is defined with `noData` indicating a specific enum value to be interpreted as missing data.

## Properties and types of a tree

PROPERTY	COMPONENTTYPE	REQUIRED	NODATA
height	FLOAT32	Yes	
species	ENUM		"Unknown"
leafColor	STRING	Yes	

### 9.3.4.14. Property Values Structure

Property values that appear as part of the class definition are the offset, scale, minimum, maximum, default values and no-data values. The structure of these values inside the class definition depends on the type of the property. For SCALAR (non-array) types, they are single values. For all other cases, they are arrays:

- For SCALAR array types with fixed length count, they are arrays with length count.
- For VECN types, they are arrays, with length N.
- For MATN types, they are arrays, with length  $N * N$ .
- For VECN array types with fixed length count, they are arrays with length count, where each array element is itself an array of length N
- For MATN array types with fixed length count, they are arrays with length count, where each array element is itself an array of length  $N * N$ .

For noData values and numeric values that are not normalized, the type of the innermost elements of these arrays corresponds to the componentType. For numeric values that are normalized, the innermost elements are floating-point values.

## 9.4. Storage Formats

---

### 9.4.1. Overview

Schemas provide templates for entities, but creating an entity requires specific property values and storage. This section covers two storage formats for entity metadata:

- **Binary Table Format** – property values are stored in parallel 1D arrays, encoded as binary data

- **JSON Format** – property values are stored in key/value dictionaries, encoded as JSON objects

Both formats are suitable for general purpose metadata storage. Binary formats may be preferable for larger quantities of metadata.

Additional serialization methods may be defined outside of this specification. For example, property values could be stored in texture channels or retrieved from a REST API as XML data.

**NOTE** Any specification that references 3D Metadata shall state explicitly which storage formats are supported, or define its own serialization. For example, the [EXT\\_structural\\_metadata](#) glTF extension implements the binary table format described below, and defines an additional image-based format for per-texel metadata.

## 9.4.2. Binary Table Format

### 9.4.2.1. Overview

The binary table format is similar to a database table where entities are rows and properties are columns. Each column represents one of the properties of the class. Each row represents a single entity conforming to the class.



The illustration shows three simple house icons: a green two-story house with a pink roof, a blue three-story house with a pink roof, and a brown two-story house with a yellow roof. To the right of the houses is a table with the title "Houses". The table has three columns: "ID", "Name", and "Stories". The data is as follows:

ID	Name	Stories
0	Green House	2
1	Blue House	3
2	Brown House	2

**Figure 58** – Illustration of metadata that can be stored in a table

The rows of a table are addressed by an integer index called an **entity ID**. Entity IDs are always numbered  $0, 1, \dots, N - 1$  where  $N$  is the number of rows in the table.

Property values are stored in parallel arrays called **property arrays**, one per column. Each property array stores values for a single property. The  $i$ -th value of each property array is the value of that property for the entity with an entity ID of  $i$ .

Binary encoding is efficient for runtime use, and scalable to large quantities of metadata. Because property arrays contain elements of a single type, bitstreams may be tightly packed or may use compression methods appropriate for a particular data type.

Property values are binary-encoded according to their data type, in little-endian format. Values are tightly packed: there is no padding between values.

#### 9.4.2.2. Scalars

A scalar value is encoded based on the `componentType`. Multiple values are packed tightly in the same buffer. The following data types are supported:

**Table 31** – Types for scalar metadata values

NAME	DESCRIPTION
INT8	8-bit two's complement signed integer
UINT8	8-bit unsigned integer
INT16	16-bit two's complement signed integer
UINT16	16-bit unsigned integer
INT32	32-bit two's complement signed integer
UINT32	32-bit unsigned integer
INT64	64-bit two's complement signed integer
UINT64	64-bit unsigned integer
FLOAT32	32-bit IEEE floating point number
FLOAT64	64-bit IEEE floating point number

#### 9.4.2.3. Vectors

Vector components are tightly packed and encoded based on the `componentType`.

#### 9.4.2.4. Matrices

Matrix components are tightly packed in column-major order and encoded based on the `componentType`.

#### 9.4.2.5. Booleans

A boolean value is encoded as a single bit, either 0 (false) or 1 (true). Multiple boolean values are packed tightly in the same buffer. These buffers of tightly-packed bits are sometimes referred to as bitstreams.

For a table with N rows, the buffer that stores these boolean values will consist of  $\text{ceil}(N / 8)$  bytes. When N is not divisible by 8, then the unused bits of the last byte of this buffer shall be set to 0.

**NOTE** Example accessing a boolean value for entity ID i.

```
byteIndex = floor(i / 8)
bitIndex = i % 8
bitValue = (buffer[byteIndex] >> bitIndex) & 1
value = bitValue == 1
```

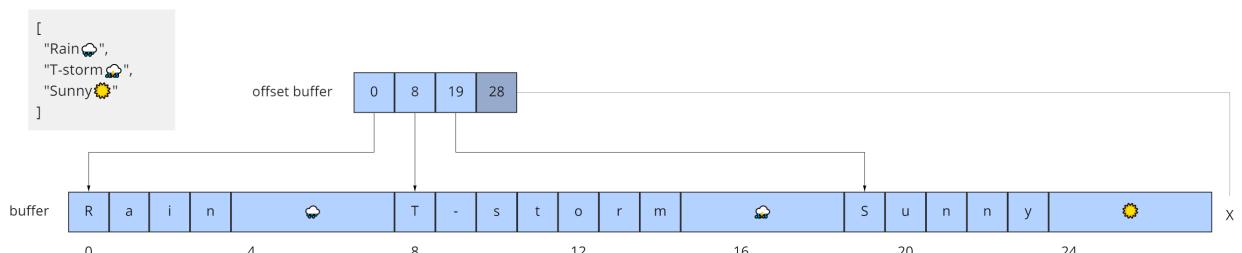
#### 9.4.2.6. Strings

A string value is a UTF-8 encoded byte sequence. Multiple strings are packed tightly in the same buffer.

Because string lengths may vary, a **string offset** buffer is used to identify strings in the buffer. If there are N strings in the property array, the string offset buffer has N + 1 elements. The first N of these point to the first byte of each string, while the last points to the byte immediately after the last string. The number of bytes in the i-th string is given by  $\text{stringOffset}[i + 1] - \text{stringOffset}[i]$ . UTF-8 encodes each character as 1-4 bytes, so string offsets do not necessarily represent the number of characters in the string.

The data type used for offsets is defined by a **string offset type**, which may be `UINT8`, `UINT16`, `UINT32`, or `UINT64`.

**NOTE** Three UTF-8 strings, binary-encoded in a buffer.



**Figure 60 – Data layout for the buffers storing string metadata**

#### 9.4.2.7. Enums

Enums are encoded as integer values according to the enum value type (see Enums). Any integer data type supported for Scalars may be used for enum values.

#### 9.4.2.8. Fixed-Length Arrays

A fixed-length array value is encoded as a tightly packed array of count elements, where each element is encoded according to the type.

#### 9.4.2.9. Variable-Length Arrays

Variable-length arrays use an additional **array offset** buffer. The  $i$ -th value in the array offset buffer is an element index — not a byte offset — identifying the beginning of the  $i$ -th array. String values within an array may have inconsistent lengths, requiring both array offset and **string offset** buffers (see: Strings).

The data type used for offsets is defined by an **array offset type**, which may be `UINT8`, `UINT16`, `UINT32`, or `UINT64`.

If there are  $N$  arrays in the property array, the array offset buffer has  $N + 1$  elements. The first  $N$  of these point to the first element of an array within the property array, or within a string offset buffer for string component types. The last value points to a (non-existent) element immediately following the last array element.

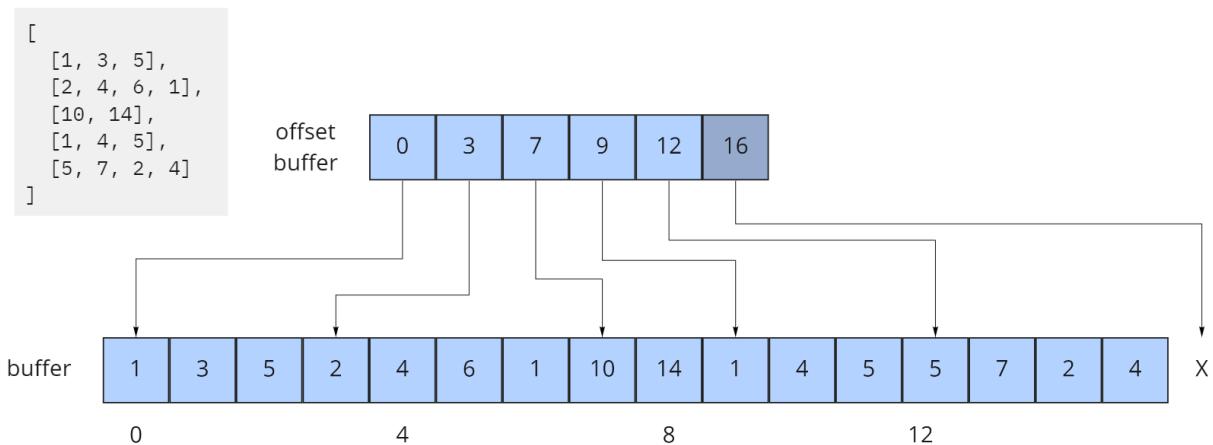
For each case below, the offset of an array element  $i$  within its binary storage is expressed in terms of entity ID  $id$  and element index  $i$ .

**Table 32 – Offset types for strings and arrays**

TYPE	OFFSET TYPE	OFFSET
STRING	byte offset	<code>stringOffset[arrayOffset[id] + i]</code>
All other types	array index	<code>arrayOffset[id] + i</code>

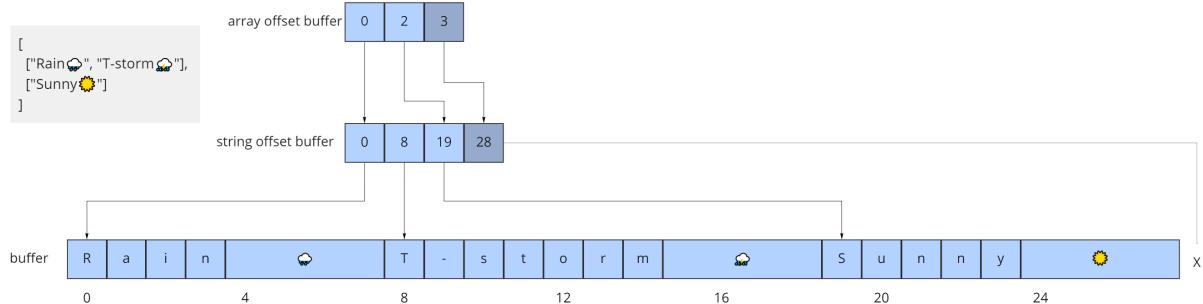
Each expression in the table above defines an index into the underlying property array. For a property array of SCALAR elements with `FLOAT32` component type, index 3 corresponds to byte offset  $3 * \text{sizeof}(\text{FLOAT32})$ . For a property array of `VEC4` elements with `FLOAT32` component type, index 3 corresponds to byte offset  $3 * 4 * \text{sizeof}(\text{FLOAT32}) = 48$ . For an array of `BOOLEAN` elements, offset 3 would correspond to bit offset 3.

**NOTE 1** Five variable-length arrays of `UINT8` components, binary-encoded in a buffer. The associated property definition would be `type = "SCALAR"`, `componentType = "UINT8"`, and `array = true`.



**Figure 61 – Data layout for the buffers storing string variable-length arrays**

**NOTE 2** Two variable-length arrays of strings, binary-encoded in a buffer. The associated property definition would be `type = "STRING"` and `array = true` (variable-length). Observe that the last element of the array offset buffer points to the last element of the string offset buffer. This is because the last valid string offset is the next-to-last element of the string offset buffer.



**Figure 62 – Data layout for the buffers storing string variable-length arrays of strings**

### 9.4.3. JSON Format

#### 9.4.3.1. Overview

JSON encoding is useful for storing a small number of entities in human readable form.

Each entity is represented as a JSON object with its class identified by a string ID. Property values are defined in a key/value properties dictionary, having property IDs as its keys. Property values are encoded as corresponding JSON types: numeric types are represented as

number, booleans as boolean, strings as string, enums as string, vectors and matrices as array of number, and arrays as array of the containing type.

**NOTE**The following example demonstrates usage for both fixed- and variable-length arrays:

An enum, “basicEnum”, composed of three (*name: value*) pairs:

Names and values of an example enum

NAME	VALUE
"Enum A"	0
"Enum B"	1
"Enum C"	2

A class, “basicClass”, composed of ten properties. *stringArrayProperty count* is undefined and therefore variable-length.

Properties of an example class

ID	TYPE	COMPONENT ARRAY	COUNT	ENUMTYPE	REQUIRED
floatProperty	SCALAR	FLOAT64	false		Yes
integerProperty	SCALAR	INT32	false		Yes
vectorProperty	VEC2	FLOAT32	false		Yes
floatArrayProperty	SCALAR	FLOAT32	true	3	Yes
vectorArrayProperty	VEC2	FLOAT32	true	2	Yes
booleanProperty	BOOLEAN		false		Yes
stringProperty	STRING		false		Yes
enumProperty	ENUM		false	basicEnum	Yes
stringArrayProperty	STRING		true		Yes
optionalProperty	STRING		false		

A single entity encoded in JSON. Note that the optional property is omitted in this example.

```
{  
  "entity": {  
    "class": "basicClass",  
    "properties": {  
      "floatProperty": 1.5,  
      "integerProperty": -90,  
      "vectorProperty": [0.0, 1.0],  
      "floatArrayProperty": [1.0, 0.5, -0.5],  
      "vectorArrayProperty": [[0.0, 1.0], [1.0, 2.0]],  
      "booleanProperty": true,  
      "stringProperty": "x123",  
      "enumProperty": "Enum B",  
      "stringArrayProperty": ["abc", "12345", "おはようございます"]  
    }  
  }  
}
```

#### 9.4.3.2. Scalars

All component types (INT8, UINT8, INT16, UINT16, INT32, UINT32, INT64, UINT64, FLOAT32, and FLOAT64) are encoded as JSON numbers. Floating point values shall be representable as IEEE floating point numbers.

**NOTE**For numeric types the size in bits is made explicit. Even though JSON only has a single number type for all integers and floating point numbers, the application that consumes the JSON may make a distinction. For example, C and C++ have several different integer types such as uint8\_t, uint32\_t. The application is responsible for interpreting the metadata using the type specified in the property definition.

#### 9.4.3.3. Vectors

Vectors are encoded as a JSON array of numbers.

#### 9.4.3.4. Matrices

Matrices are encoded as a JSON array of numbers in column-major order.

#### 9.4.3.5. Booleans

Booleans are encoded as a JSON boolean, either true or false.

#### 9.4.3.6. Strings

Strings are encoded as JSON strings.

#### **9.4.3.7. Enums**

Enums are encoded as JSON strings using the name of the enum value rather than the integer value. Therefore the enum value type, if specified, is ignored for the JSON encoding.

#### **9.4.3.8. Arrays**

Arrays are encoded as JSON arrays, where each element is encoded according to the type. When a count is specified, the length of the JSON array shall match the count. Otherwise, for variable-length arrays, the JSON array may be any length, including zero-length.

10

# STYLING

---

## 10.1. Overview

3D Tiles styles provide concise declarative styling of tileset features. A style defines expressions to evaluate the display of a feature, for example color (RGB and translucency) and show properties, often based on the feature's properties stored in the tile's Batch Table.

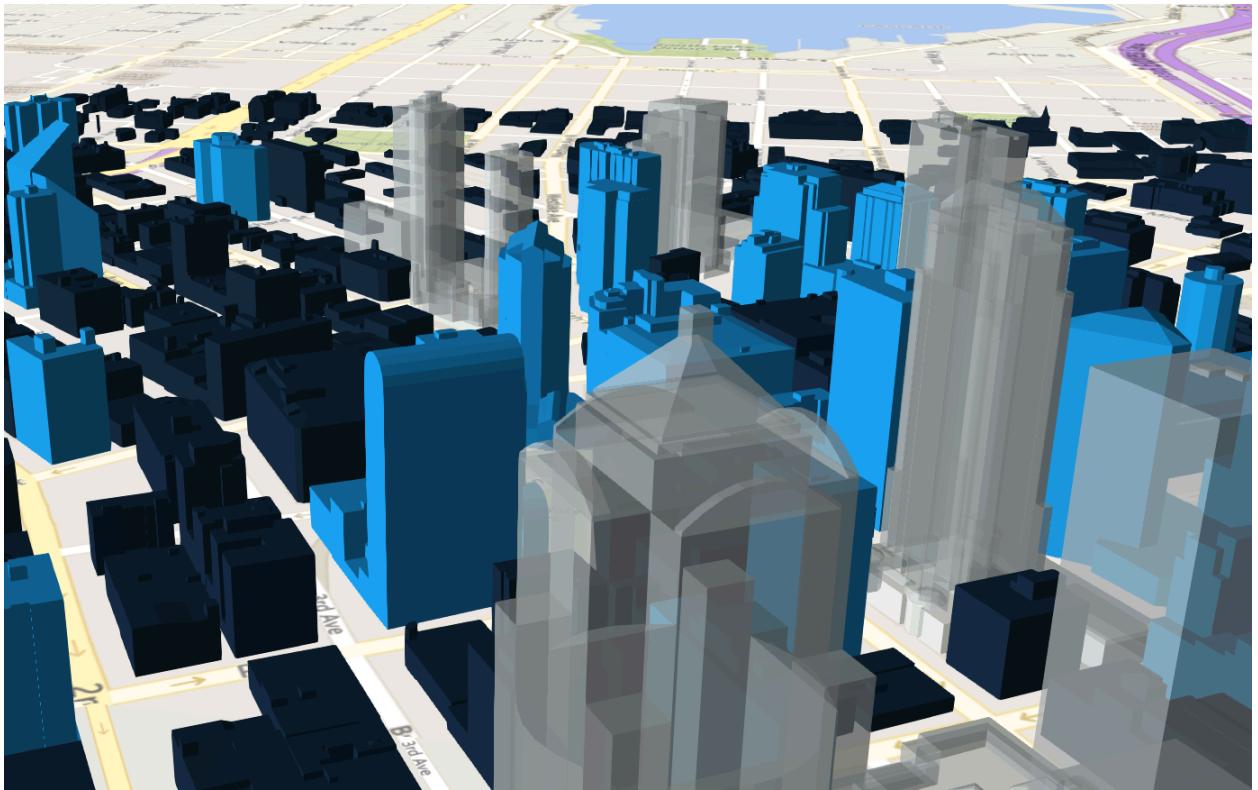
A style may be applied to a tile that doesn't contain features, in which case the tile is treated as an implicit single feature without properties.

While a style may be created for and reference properties of a tileset, a style is independent of a tileset, such that any style can be applied to any tileset.

Styles are defined with JSON and expressions written in a small subset of JavaScript augmented for styling. Additionally, the styling language provides a set of built-in functions to support common math operations.

The following example assigns a color based on building height.

```
{  
    "show" : "${Area} > 0",  
    "color" : {  
        "conditions" : [  
            ["${Height} < 60", "color('#13293D')"],  
            ["${Height} < 120", "color('#1B98E0')"],  
            ["true", "color('#E8F1F2', 0.5)"]  
        ]  
    }  
}
```



**Figure 63** – Rendering of buildings with different colors, based on the height of the buildings

## 10.2. Concepts

---

### 10.2.1. Styling features

Visual properties available for styling features are the `show` property, the assigned expression of which will evaluate to a boolean that determines if the feature is visible, and the `color` property, the assigned expression of which will evaluate to a `Color` object (RGB and translucency) which determines the displayed color of a feature.

The following style assigns the default `show` and `color` properties to each feature:

```
{  
  "show" : "true",  
  "color" : "color('#ffffff')"  
}
```

Instead of showing all features, `show` can be an expression dependent on a feature's properties, for example, the following expression will show only features in the 19341 zip code:

```
{  
  "show" : "${ZipCode} === '19341'"  
}
```

show can also be used for more complex queries; for example, here a compound condition and regular expression are used to show only features whose county starts with 'Chest' and whose year built is greater than or equal to 1970:

```
{  
    "show" : "(RegExp('^Chest').test(${County})) && (${YearBuilt} >= 1970)"  
}
```

Colors can also be defined by expressions dependent on a feature's properties. For example, the following expression colors features with a temperature above 90 as red and the others as white:

```
{  
    "color" : "(${Temperature} > 90) ? color('red') : color('white')"  
}
```

The color's alpha component defines the feature's opacity. For example, the following sets the feature's RGB color components from the feature's properties and makes features with volume greater than 100 transparent:

```
{  
    "color" : "rgba(${red}, ${green}, ${blue}, (${volume} > 100 ? 0.5 : 1.0))"  
}
```

## 10.2.2. Conditions

In addition to a string containing an expression, color and show can be an array defining a series of conditions (similar to if...else statements). Conditions can, for example, be used to make color maps and color ramps with any type of inclusive/exclusive intervals.

For example, the following expression maps an ID property to colors. Conditions are evaluated in order, so if \${id} is not '1' or '2', the "true" condition returns white. If no conditions are met, the color of the feature will be undefined:

```
{  
    "color" : {  
        "conditions" : [  
            ["${id} === '1'", "color('#FF0000')"],  
            ["${id} === '2'", "color('#00FF00')"],  
            ["true", "color('#FFFFFF')"]  
        ]  
    }  
}
```

The next example shows how to use conditions to create a color ramp using intervals with an inclusive lower bound and exclusive upper bound:

```
"color" : {  
    "conditions" : [  
        [">= 1.0) && (< 10.0)", "color('#FF00FF')"],  
        [">= 10.0) && (< 30.0)", "color('#FF0000')"],  
        [">= 30.0) && (< 50.0)", "color('#FFFF00')"],  
        [">= 50.0) && (< 70.0)", "color('#00FF00')"],  
        [">= 70.0) && (< 100.0)", "color('#00FFFF')"],  
        [">= 100.0)", "color('#0000FF')"]  
    ]  
}
```

Since conditions are evaluated in order, the above can be written more concisely as the following:

```
"color" : {
    "conditions" : [
        ["(${Height}) >= 100.0)", "color('#0000FF')"],
        ["(${Height}) >= 70.0)", "color('#00FFFF')"],
        ["(${Height}) >= 50.0)", "color('#00FF00')"],
        ["(${Height}) >= 30.0)", "color('#FFFF00')"],
        ["(${Height}) >= 10.0)", "color('#FF0000')"],
        ["(${Height}) >= 1.0)", "color('#FF00FF')"]
    ]
}
```

### 10.2.3. Defining variables

Commonly used expressions may be stored in a `defines` object with a variable name as a key. If a variable references the name of a defined expression, it is replaced with the result of the referenced evaluated expression:

```
{
    "defines" : {
        "NewHeight" : "clamp(${Height} - 0.5) / 2.0, 1.0, 255.0",
        "HeightColor" : "rgb(${Height}, ${Height}, ${Height})"
    },
    "color" : {
        "conditions" : [
            ["(${NewHeight}) >= 100.0)", "color('#0000FF') * ${HeightColor}"],
            ["(${NewHeight}) >= 50.0)", "color('#00FF00') * ${HeightColor}"],
            ["(${NewHeight}) >= 1.0)", "color('#FF0000') * ${HeightColor}"
        ]
    },
    "show" : "${NewHeight} < 200.0"
}
```

A define expression may not reference other defines; however, it may reference feature properties with the same name. In the style below a feature of height 150 gets the color red:

```
{
    "defines" : {
        "Height" : "${Height}/2.0",
    },
    "color" : {
        "conditions" : [
            ["(${Height}) >= 100.0)", "color('#0000FF')"],
            ["(${Height}) >= 1.0)", "color('#FF0000')"
        ]
    }
}
```

### 10.2.4. Meta property

Non-visual properties of a feature can be defined using the `meta` property. For example, the following sets a `description` meta property to a string containing the feature name:

```
{
    "meta" : {
```

```
        "description" : "'Hello, ${featureName}.'"
    }
}
```

A meta property expression can evaluate to any type. For example:

```
{
  "meta" : {
    "featureColor" : "rgb(${red}, ${green}, ${blue})",
    "featureVolume" : "${height} * ${width} * ${depth}"
  }
}
```

## 10.3. Expressions

---

The language for expressions is a small subset of JavaScript ([ECMAScript 5](#)), plus native vector and regular expression types and access to tileset feature properties in the form of readonly variables.

**NOTE**CesiumJS uses the [jsep](#) JavaScript expression parser library to parse style expressions into an [abstract syntax tree \(AST\)](#).

### 10.3.1. Semantics

Dot notation is used to access properties by name, e.g., `building.name`.

Bracket notation `([])` is also used to access properties, e.g., `building['name']`, or arrays, e.g., `temperatures[1]`.

Functions are called with parenthesis `(( ))` and comma-separated arguments, e.g., `(isNaN(0.0), color('cyan', 0.5))`.

### 10.3.2. Operators

The following operators are supported with the same semantics and precedence as JavaScript.

- Unary: `+`, `-`, `!`
  - Not supported: `~`
- Binary: `|`, `&&`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `+`, `-`, `*`, `/`, `%`, `=~`, `!~`
  - Not supported: `|`, `^`, `&`, `<<`, `>>`, and `>>>`
- Ternary: `? :`

( and ) are also supported for grouping expressions for clarity and precedence.

Logical || and && implement short-circuiting; true || expression does not evaluate the right expression, and false && expression does not evaluate the right expression.

Similarly, true ? leftExpression : rightExpression only executes the left expression, and false ? leftExpression : rightExpression only executes the right expression.

### 10.3.3. Types

The following types are supported:

- Boolean
- Null
- Undefined
- Number
- String
- Array
- vec2
- vec3
- vec4
- RegExp

All of the types except vec2, vec3, vec4, and RegExp have the same syntax and runtime behavior as JavaScript. vec2, vec3, and vec4 are derived from GLSL vectors and behave similarly to JavaScript Object (see the Vector section). Colors derive from [CSS3 Colors](#) and are implemented as vec4. RegExp is derived from JavaScript and described in the RegExp section.

Example expressions for different types include the following:

- true, false
- null
- undefined
- 1.0, NaN, Infinity
- 'Cesium', "Cesium"
- [0, 1, 2]
- vec2(1.0, 2.0)

- `vec3(1.0, 2.0, 3.0)`
- `vec4(1.0, 2.0, 3.0, 4.0)`
- `color('#00FFFF')`
- `RegExp('^Chest'))`

### 10.3.3.1. Number

As in JavaScript, numbers can be `Nan` or `Infinity`. The following test functions are supported:

- `isNaN(testValue : Number) : Boolean`
- `isFinite(testValue : Number) : Boolean`

### 10.3.3.2. String

Strings are encoded in UTF-8.

### 10.3.3.3. Vector

The styling language includes 2, 3, and 4 component floating-point vector types: `vec2`, `vec3`, and `vec4`. Vector constructors share the same rules as GLSL:

#### 10.3.3.3.1. vec2

- `vec2(xy : Number)` – initialize each component with the number
- `vec2(x : Number, y : Number)` – initialize with two numbers
- `vec2(xy : vec2)` – initialize with another `vec2`
- `vec2(xyz : vec3)` – drops the third component of a `vec3`
- `vec2(xyzw : vec4)` – drops the third and fourth component of a `vec4`

#### 10.3.3.3.2. vec3

- `vec3(xyz : Number)` – initialize each component with the number
- `vec3(x : Number, y : Number, z : Number)` – initialize with three numbers
- `vec3(xyz : vec3)` – initialize with another `vec3`

- `vec3(xyzw : vec4)` – drops the fourth component of a `vec4`
- `vec3(xy : vec2, z : Number)` – initialize with a `vec2` and number
- `vec3(x : Number, yz : vec2)` – initialize with a `vec2` and number

#### 10.3.3.3.3. `vec4`

- `vec4(xyzw : Number)` – initialize each component with the number
- `vec4(x : Number, y : Number, z : Number, w : Number)` – initialize with four numbers
- `vec4(xyzw : vec4)` – initialize with another `vec4`
- `vec4(xy : vec2, z : Number, w : Number)` – initialize with a `vec2` and two numbers
- `vec4(x : Number, yz : vec2, w : Number)` – initialize with a `vec2` and two numbers
- `vec4(x : Number, y : Number, zw : vec2)` – initialize with a `vec2` and two numbers
- `vec4(xyz : vec3, w : Number)` – initialize with a `vec3` and number
- `vec4(x : Number, yzw : vec3)` – initialize with a `vec3` and number

#### 10.3.3.3.4. Vector usage

`vec2` components may be accessed with

- `.x, .y`
- `.r, .g`
- `[0], [1]`

`vec3` components may be accessed with

- `.x, .y, .z`
- `.r, .g, .b`
- `[0], [1], [2]`

`vec4` components may be accessed with

- `.x, .y, .z, .w`
- `.r, .g, .b, .a`

- [0], [1], [2], [3]

Unlike GLSL, the styling language does not support swizzling. For example, `vec3(1.0).xy` is not supported.

Vectors support the following unary operators: `-`, `+`.

Vectors support the following binary operators by performing component-wise operations: `==`, `!==`, `+`, `-`, `*`, `/`, and `%`. For example `vec4(1.0) === vec4(1.0)` is true since the `x`, `y`, `z`, and `w` components are equal. Operators are essentially overloaded for `vec2`, `vec3`, and `vec4`.

`vec2`, `vec3`, and `vec4` have a `toString` function for explicit (and implicit) conversion to strings in the format '`(x, y)`', '`(x, y, z)`', and '`(x, y, z, w)`'.

- `toString() : String`

`vec2`, `vec3`, and `vec4` do not expose any other functions or a prototype object.

#### 10.3.3.4. Color

Colors are implemented as `vec4` and are created with one of the following functions:

- `color()`
- `color(keyword : String, [alpha : Number])`
- `color(6-digit-hex : String, [alpha : Number])`
- `color(3-digit-hex : String, [alpha : Number])`
- `rgb(red : Number, green : Number, blue : Number)`
- `rgba(red : Number, green : Number, blue : Number, alpha : Number)`
- `hsl(hue : Number, saturation : Number, lightness : Number)`
- `hsla(hue : Number, saturation : Number, lightness : Number, alpha : Number)`

Calling `color()` with no arguments is the same as calling `color('#FFFFFF')`.

Colors defined by a case-insensitive keyword (e.g., `'cyan'`) or hex `rgb` are passed as strings to the `color` function. For example:

- `color('cyan')`
- `color('#00FFFF')`
- `color('#0FF')`

These color functions have an optional second argument that is an alpha component to define opacity, where `0.0` is fully transparent and `1.0` is fully opaque. For example:

- `color('cyan', 0.5)`

Colors defined with decimal RGB or HSL are created with `rgb` and `hsl` functions, respectively, just as in CSS (but with percentage ranges from `0.0` to `1.0` for `0%` to `100%`, respectively). For example:

- `rgb(100, 255, 190)`
- `hsl(1.0, 0.6, 0.7)`

The range for `rgb` components is `0` to `255`, inclusive. For `hsl`, the range for hue, saturation, and lightness is `0.0` to `1.0`, inclusive.

Colors defined with `rgba` or `hsla` have a fourth argument that is an alpha component to define opacity, where `0.0` is fully transparent and `1.0` is fully opaque. For example:

- `rgba(100, 255, 190, 0.25)`
- `hsla(1.0, 0.6, 0.7, 0.75)`

Colors are equivalent to the `vec4` type and share the same functions, operators, and component accessors. Color components are stored in the range `0.0` to `1.0`.

For example:

- `color('red').x`, `color('red').r`, and `color('red')[0]` all evaluate to `1.0`.
- `color('red').toString()` evaluates to `(1.0, 0.0, 0.0, 1.0)`
- `color('red') * vec4(0.5)` is equivalent to `vec4(0.5, 0.0, 0.0, 0.5)`

### 10.3.3.5. RegExp

Regular expressions are created with the following functions, which behave like the JavaScript `RegExp` constructor:

- `RegExp()`
- `RegExp(pattern : String, [flags : String])`

Calling `RegExp()` with no arguments is the same as calling `RegExp('(?:)')`.

If specified, `flags` can have any combination of the following values:

- `g` – global match

- i – ignore case
- m – multiline
- u – unicode
- y- sticky

Regular expressions support these functions:

- `test(string : String) : Boolean` – Tests the specified string for a match.
- `exec(string : String) : String` – Executes a search for a match in the specified string. If the search succeeds, it returns the first instance of a captured String. If the search fails, it returns null.

For example:

```
{
  "Name" : "Building 1"
}

regExp('a').test('abc') === true
regExp('a(.)', 'i').exec('Abc') === 'b'
regExp('Building\s(\d)').exec(${Name}) === '1'
```

Regular expressions have a `toString` function for explicit (and implicit) conversion to strings in the format 'pattern':

- `toString() : String`

Regular expressions do not expose any other functions or a prototype object.

The operators `=~` and `!~` are overloaded for regular expressions. The `=~` operator matches the behavior of the `test` function, and tests the specified string for a match. It returns `true` if one is found, and `false` if not found. The `!~` operator is the inverse of the `=~` operator. It returns `true` if no matches are found, and `false` if a match is found. Both operators are commutative.

For example, the following expressions all evaluate to true:

```
regExp('a') =~ 'abc'
'abc' =~ regExp('a')

regExp('a') !~ 'bcd'
'bcd' !~ regExp('a')
```

#### 10.3.4. Operator rules

- Unary operators `+` and `-` operate only on number and vector expressions.
- Unary operator `!` operates only on boolean expressions.

- Binary operators <, <=, >, and >= operate only on number expressions.
- Binary operators || and && operate only on boolean expressions.
- Binary operator + operates on the following expressions:
  - Number expressions
  - Vector expressions of the same type
  - If at least one expression is a string, the other expression is converted to a string following String Conversions, and the operation returns a concatenated string, e.g. "name" + 10 evaluates to "name10"
- Binary operator - operates on the following expressions:
  - Number expressions
  - Vector expressions of the same type
- Binary operator \* operates on the following expressions:
  - Number expressions
  - Vector expressions of the same type
  - Mix of number expression and vector expression, e.g. 3 \* vec3(1.0) and vec2(1.0) \* 3
- Binary operator / operates on the following expressions:
  - Number expressions
  - Vector expressions of the same type
  - Vector expression followed by number expression, e.g. vec3(1.0) / 3
- Binary operator % operates on the following expressions:
  - Number expressions
  - Vector expressions of the same type
- Binary equality operators === and !== operate on any expressions. The operation returns false if the expression types do not match.
- Binary regExp operators =~ and !~ require one argument to be a string expression and the other to be a RegExp expression.
- Ternary operator ? : conditional argument shall be a boolean expression.

### 10.3.5. Type conversions

Explicit conversions between primitive types are handled with Boolean, Number, and String functions.

- Boolean(value : Any) : Boolean
- Number(value : Any) : Number
- String(value : Any) : String

For example:

```
Boolean(1) === true  
Number('1') === 1  
String(1) === '1'
```

Boolean and Number follow JavaScript conventions. String follows String Conversions.

These are essentially casts, not constructor functions.

The styling language does not allow for implicit type conversions, unless stated above. Expressions like vec3(1.0) === vec4(1.0) and "5" < 6 are not valid.

### 10.3.6. String conversions

vec2, vec3, vec4, and RegExp expressions are converted to strings using their `toString` methods. All other types follow JavaScript conventions.

- true — "true"
- false — "false"
- null — "null"
- undefined — "undefined"
- 5.0 — "5"
- NaN — "NaN"
- Infinity — "Infinity"
- "name" — "name"
- [0, 1, 2] — "[0, 1, 2]"
- vec2(1, 2) — "(1, 2)"
- vec3(1, 2, 3) — "(1, 2, 3)"

- `vec4(1, 2, 3, 4) - "(1, 2, 3, 4)"`
- `RegExp('a') - "/a/"`

## 10.3.7. Constants

The following constants are supported by the styling language:

- `Math.PI`
- `Math.E`

### 10.3.7.1. PI

The mathematical constant PI, which represents a circle's circumference divided by its diameter, approximately 3.14159.

```
{
  "show" : "cos(${Angle} + Math.PI) < 0"
}
```

### 10.3.7.2. E

Euler's constant and the base of the natural logarithm, approximately 2.71828.

```
{
  "color" : "color() * pow(Math.E / 2.0, ${Temperature})"
}
```

## 10.3.8. Variables

Variables are used to retrieve the property values of individual features in a tileset. Variables are identified using the ES 6 ([ECMAScript 2015](#)) template literal syntax, i.e.,  `${feature.identifier}` or  `${feature['identifier']}`, where the identifier is the case-sensitive property name. Variable names are encoded in UTF-8. `feature` is implicit and can be omitted in most cases. If the identifier contains non-alphanumeric characters, such as `:`, `-`, `#`, or spaces, the  `${feature['identifier']}` form should be used.

Variables can be used anywhere a valid expression is accepted, except inside other variable identifiers. For example, the following is not allowed:

```
 ${foo[$bar]}  
----
```

If a feature does not have a property with the specified name, the variable evaluates to `undefined`. Note that the property may also be `null` if `null` was explicitly stored for a property.

Variables may be any of the supported native JavaScript types:

- Boolean
- Null
- Undefined
- Number
- String
- Array

For example:

```
{  
    "enabled" : true,  
    "description" : null,  
    "order" : 1,  
    "name" : "Feature name"  
}  
  
${enabled} === true  
${description} === null  
${order} === 1  
${name} === 'Feature name'
```

Additionally, variables originating from vector properties stored in the Batch Table binary are treated as vector types:

**Table 33 – Vector type variables**

componentType	VARIABLE TYPE
"VEC2"	vec2
"VEC3"	vec3
"VEC4"	vec4

Variables can be used to construct colors or vectors. For example:

```
rgba(${red}, ${green}, ${blue}, ${alpha})  
vec4(${temperature})
```

Dot or bracket notation is used to access feature subproperties. For example:

```
{  
    "address" : {  
        "street" : "Example street",  
        "city" : "Example city"  
    }  
}  
  
${address.street} === `Example street`
```

```

${address['street']} === `Example street`
${address.city} === `Example city`
${address['city']} === `Example city`

Bracket notation supports only string literals.

```

Top-level properties can be accessed with bracket notation by explicitly using the `feature` keyword. For example:

```

{
  "address.street" : "Maple Street",
  "address" : {
    "street" : "Oak Street"
  }
}

${address.street} === `Oak Street`
${feature.address.street} === `Oak Street`
${feature['address'].street} === `Oak Street`
${feature['address.street']} === `Maple Street`

```

To access a feature named `feature`, use the variable `${feature}`. This is equivalent to accessing `${feature.feature}`

```

{
  "feature" : "building"
}

${feature} === `building`
${feature.feature} === `building`

```

Variables can also be substituted inside strings defined with backticks, for example:

```

{
  "order" : 1,
  "name" : "Feature name"
}

`Name is ${name}, order is ${order}`

```

Bracket notation is used to access feature subproperties or arrays. For example:

```

{
  "temperatures" : {
    "scale" : "fahrenheit",
    "values" : [70, 80, 90]
  }
}

${temperatures['scale']} === 'fahrenheit'
${temperatures.values[0]} === 70
${temperatures['values'][0]} === 70 // Same as (temperatures.values)[0] and
temperatures.values[0]

```

### 10.3.9. Built-in functions

The following built-in functions are supported by the styling language:

- `abs`

- sqrt
- cos
- sin
- tan
- acos
- asin
- atan
- atan2
- radians
- degrees
- sign
- floor
- ceil
- round
- exp
- log
- exp2
- log2
- fract
- pow
- min
- max
- clamp
- mix
- length
- distance
- normalize
- dot

- cross

Many of the built-in functions take either scalars or vectors as arguments. For vector arguments the function is applied component-wise and the resulting vector is returned.

### 10.3.9.1. abs

```
abs(x : Number) : Number
abs(x : vec2) : vec2
abs(x : vec3) : vec3
abs(x : vec4) : vec4
```

Returns the absolute value of x.

```
{
  "show" : "abs(${temperature}) > 20.0"
}
```

### 10.3.9.2. sqrt

```
sqrt(x : Number) : Number
sqrt(x : vec2) : vec2
sqrt(x : vec3) : vec3
sqrt(x : vec4) : vec4
```

Returns the square root of x when  $x \geq 0$ . Returns NaN when  $x < 0$ .

```
{
  "color" : {
    "conditions" : [
      ["${temperature} >= 0.5", "color('#00FFFF')"],
      ["${temperature} >= 0.0", "color('#FF00FF')"]
    ]
  }
}
```

### 10.3.9.3. cos

```
cos(angle : Number) : Number
cos(angle : vec2) : vec2
cos(angle : vec3) : vec3
cos(angle : vec4) : vec4
```

Returns the cosine of angle in radians.

```
{
  "show" : "cos(${Angle}) > 0.0"
}
```

### 10.3.9.4. sin

```
sin(angle : Number) : Number
sin(angle : vec2) : vec2
```

```
sin(angle : vec3) : vec3
sin(angle : vec4) : vec4

>Returns the sine of angle in radians.

{
    "show" : "sin(${Angle}) > 0.0"
}
```

#### 10.3.9.5. tan

```
tan(angle : Number) : Number
tan(angle : vec2) : vec2
tan(angle : vec3) : vec3
tan(angle : vec4) : vec4
```

Returns the tangent of angle in radians.

```
{
    "show" : "tan(${Angle}) > 0.0"
}
```

#### 10.3.9.6. acos

```
acos(angle : Number) : Number
acos(angle : vec2) : vec2
acos(angle : vec3) : vec3
acos(angle : vec4) : vec4
```

Returns the arccosine of angle in radians.

```
{
    "show" : "acos(${Angle}) > 0.0"
}
```

#### 10.3.9.7. asin

```
asin(angle : Number) : Number
asin(angle : vec2) : vec2
asin(angle : vec3) : vec3
asin(angle : vec4) : vec4
```

Returns the arcsine of angle in radians.

```
{
    "show" : "asin(${Angle}) > 0.0"
}
```

#### 10.3.9.8. atan

```
atan(angle : Number) : Number
atan(angle : vec2) : vec2
atan(angle : vec3) : vec3
atan(angle : vec4) : vec4
```

Returns the arctangent of angle in radians.

```
{  
    "show" : "atan(${Angle}) > 0.0"  
}
```

### 10.3.9.9. atan2

```
atan2(y : Number, x : Number) : Number  
atan2(y : vec2, x : vec2) : vec2  
atan2(y : vec3, x : vec3) : vec3  
atan2(y : vec4, x : vec4) : vec4
```

Returns the arctangent of the quotient of y and x.

```
{  
    "show" : "atan2(${GridY}, ${GridX}) > 0.0"  
}
```

### 10.3.9.10. radians

```
radians(angle : Number) : Number  
radians(angle : vec2) : vec2  
radians(angle : vec3) : vec3  
radians(angle : vec4) : vec4
```

Converts angle from degrees to radians.

```
{  
    "show" : "radians(${Angle}) > 0.5"  
}
```

### 10.3.9.11. degrees

```
degrees(angle : Number) : Number  
degrees(angle : vec2) : vec2  
degrees(angle : vec3) : vec3  
degrees(angle : vec4) : vec4
```

Converts angle from radians to degrees.

```
{  
    "show" : "degrees(${Angle}) > 45.0"  
}
```

### 10.3.9.12. sign

```
sign(x : Number) : Number  
sign(x : vec2) : vec2  
sign(x : vec3) : vec3  
sign(x : vec4) : vec4
```

Returns 1.0 when x is positive, 0.0 when x is zero, and -1.0 when x is negative.

```
{
```

```
        "show" : "sign(${Temperature}) * sign(${Velocity}) === 1.0"
    }
```

### 10.3.9.13. floor

```
floor(x : Number) : Number
floor(x : vec2) : vec2
floor(x : vec3) : vec3
floor(x : vec4) : vec4
```

Returns the nearest integer less than or equal to x.

```
{
    "show" : "floor(${Position}) === 0"
}
```

### 10.3.9.14. ceil

```
ceil(x : Number) : Number
ceil(x : vec2) : vec2
ceil(x : vec3) : vec3
ceil(x : vec4) : vec4
```

Returns the nearest integer greater than or equal to x.

```
{
    "show" : "ceil(${Position}) === 1"
}
```

### 10.3.9.15. round

```
round(x : Number) : Number
round(x : vec2) : vec2
round(x : vec3) : vec3
round(x : vec4) : vec4
```

Returns the nearest integer to x. A number with a fraction of 0.5 will round in an implementation-defined direction.

```
{
    "show" : "round(${Position}) === 1"
}
```

### 10.3.9.16. exp

```
exp(x : Number) : Number
exp(x : vec2) : vec2
exp(x : vec3) : vec3
exp(x : vec4) : vec4
```

Returns e to the power of x, where e is Euler's constant, approximately 2.71828.

```
{
    "show" : "exp(${Density}) > 1.0"
```

```
}
```

### 10.3.9.17. log

```
log(x : Number) : Number  
log(x : vec2) : vec2  
log(x : vec3) : vec3  
log(x : vec4) : vec4
```

Returns the natural logarithm (base e) of x.

```
{  
    "show" : "log(${Density}) > 1.0"  
}
```

### 10.3.9.18. exp2

```
exp2(x : Number) : Number  
exp2(x : vec2) : vec2  
exp2(x : vec3) : vec3  
exp2(x : vec4) : vec4
```

Returns 2 to the power of x.

```
{  
    "show" : "exp2(${Density}) > 1.0"  
}
```

### 10.3.9.19. log2

```
log2(x : Number) : Number  
log2(x : vec2) : vec2  
log2(x : vec3) : vec3  
log2(x : vec4) : vec4
```

Returns the base 2 logarithm of x.

```
{  
    "show" : "log2(${Density}) > 1.0"  
}
```

### 10.3.9.20. fract

```
fract(x : Number) : Number  
fract(x : vec2) : vec2  
fract(x : vec3) : vec3  
fract(x : vec4) : vec4
```

Returns the fractional part of x. Equivalent to  $x - \text{floor}(x)$ .

```
{  
    "color" : "color() * fract(${Density})"  
}
```

### 10.3.9.21. pow

```
pow(base : Number, exponent : Number) : Number  
pow(base : vec2, exponent : vec2) : vec2  
pow(base : vec3, exponent : vec3) : vec3  
pow(base : vec4, exponent : vec4) : vec4
```

Returns base raised to the power of exponent.

```
{  
    "show" : "pow(${Density}, ${Temperature}) > 1.0"  
}
```

### 10.3.9.22. min

```
min(x : Number, y : Number) : Number  
min(x : vec2, y : vec2) : vec2  
min(x : vec3, y : vec3) : vec3  
min(x : vec4, y : vec4) : vec4  
  
min(x : Number, y : Number) : Number  
min(x : vec2, y : Number) : vec2  
min(x : vec3, y : Number) : vec3  
min(x : vec4, y : Number) : vec4
```

Returns the smaller of x and y.

```
{  
    "show" : "min(${Width}, ${Height}) > 10.0"  
}
```

### 10.3.9.23. max

```
max(x : Number, y : Number) : Number  
max(x : vec2, y : vec2) : vec2  
max(x : vec3, y : vec3) : vec3  
max(x : vec4, y : vec4) : vec4  
  
max(x : Number, y : Number) : Number  
max(x : vec2, y : Number) : vec2  
max(x : vec3, y : Number) : vec3  
max(x : vec4, y : Number) : vec4
```

Returns the larger of x and y.

```
{  
    "show" : "max(${Width}, ${Height}) > 10.0"  
}
```

### 10.3.9.24. clamp

```
clamp(x : Number, min : Number, max : Number) : Number  
clamp(x : vec2, min : vec2, max : vec2) : vec2  
clamp(x : vec3, min : vec3, max : vec3) : vec3
```

```
clamp(x : vec4, min : vec4, max : vec4) : vec4
clamp(x : Number, min : Number, max : Number) : Number
clamp(x : vec2, min : Number, max : Number) : vec2
clamp(x : vec3, min : Number, max : Number) : vec3
clamp(x : vec4, min : Number, max : Number) : vec4
```

Constrains x to lie between min and max.

```
{
    "color" : "color() * clamp(${temperature}, 0.1, 0.2)"
}
```

### 10.3.9.25. mix

```
mix(x : Number, y : Number, a : Number) : Number
mix(x : vec2, y : vec2, a : vec2) : vec2
mix(x : vec3, y : vec3, a : vec3) : vec3
mix(x : vec4, y : vec4, a : vec4) : vec4

mix(x : Number, y : Number, a : Number) : Number
mix(x : vec2, y : vec2, a : Number) : vec2
mix(x : vec3, y : vec3, a : Number) : vec3
mix(x : vec4, y : vec4, a : Number) : vec4
```

Computes the linear interpolation of x and y.

```
{
    "show" : "mix(20.0, ${Angle}, 0.5) > 25.0"
}
```

### 10.3.9.26. length

```
length(x : Number) : Number
length(x : vec2) : vec2
length(x : vec3) : vec3
length(x : vec4) : vec4
```

Computes the length of vector x, i.e., the square root of the sum of the squared components. If x is a number, length returns x.

```
{
    "show" : "length(${Dimensions}) > 10.0"
}
```

### 10.3.9.27. distance

```
distance(x : Number, y : Number) : Number
distance(x : vec2, y : vec2) : vec2
distance(x : vec3, y : vec3) : vec3
distance(x : vec4, y : vec4) : vec4
```

Computes the distance between two points x and y, i.e.,  $\text{length}(x - y)$ .

```
{
    "show" : "distance(${BottomRight}, ${UpperLeft}) > 50.0"
}
```

### 10.3.9.28. normalize

```
normalize(x : Number) : Number  
normalize(x : vec2) : vec2  
normalize(x : vec3) : vec3  
normalize(x : vec4) : vec4
```

Returns a vector with length 1.0 that is parallel to x. When x is a number, normalize returns 1.0.

```
{  
    "show" : "normalize(${RightVector}, ${UpVector}) > 0.5"  
}
```

### 10.3.9.29. dot

```
dot(x : Number, y : Number) : Number  
dot(x : vec2, y : vec2) : vec2  
dot(x : vec3, y : vec3) : vec3  
dot(x : vec4, y : vec4) : vec4
```

Computes the dot product of x and y.

```
{  
    "show" : "dot(${RightVector}, ${UpVector}) > 0.5"  
}
```

### 10.3.9.30. cross

```
cross(x : vec3, y : vec3) : vec3
```

Computes the cross product of x and y. This function only accepts vec3 arguments.

```
{  
    "color" : "vec4(cross(${RightVector}, ${UpVector}), 1.0)"  
}
```

## 10.3.10. Notes

Comments are not supported.

## 10.4. Point Cloud

---

A Point Cloud is a collection of points that may be styled like other features. In addition to evaluating a point's color and show properties, a Point Cloud style may evaluate pointSize, or the size of each point in pixels. The default pointSize is 1.0.

```
{  
    "color" : "color('red')",
```

```

    "pointSize" : "${Temperature} * 0.5"
}

```

Implementations may clamp the evaluated pointSize to the system's supported point size range. For example, WebGL renderers may query ALIASED\_POINT\_SIZE\_RANGE to get the system limits when rendering with POINTS. A pointSize of 1.0 shall be supported.

Point Cloud styles may also reference semantics from the Feature Table including position, color, and normal to allow for more flexible styling of the source data.

- \${POSITION} is a vec3 storing the xyz Cartesian coordinates of the point before the RTC\_CENTER and tile transform are applied. When the positions are quantized, \${POSITION} refers to the position after the QUANTIZED\_VOLUME\_SCALE is applied, but before QUANTIZED\_VOLUME\_OFFSET is applied.
- \${POSITION\_ABSOLUTE} is a vec3 storing the xyz Cartesian coordinates of the point after the RTC\_CENTER and tile transform are applied. When the positions are quantized, \${POSITION\_ABSOLUTE} refers to the position after the QUANTIZED\_VOLUME\_SCALE, QUANTIZED\_VOLUME\_OFFSET, and tile transform are applied.
- \${COLOR} evaluates to a Color storing the rgba color of the point. When the Feature Table's color semantic is RGB or RGB565, \${COLOR}.alpha is 1.0. If no color semantic is defined, \${COLOR} evaluates to the application-specific default color.
- \${NORMAL} is a vec3 storing the normal, in Cartesian coordinates, of the point before the tile transform is applied. When normals are oct-encoded, \${NORMAL} refers to the decoded normal. If no normal semantic is defined in the Feature Table, \${NORMAL} evaluates to undefined.

For example:

```

{
  "color" : "${COLOR} * color('red')",
  "show" : "${POSITION}.x > 0.5",
  "pointSize" : "${NORMAL}.x > 0 ? 2 : 1"
}

```

**NOTE** Point cloud styling engines may often use a shader (GLSL) implementation, however some features of the expression language are not possible in pure a GLSL implementation. Some of these features include:

- Evaluation of isNaN and isFinite (GLSL 2.0+ supports isnan and isinf for these functions respectively)
- The types null and undefined
- Strings, including accessing object properties (color()['r']) and batch table values
- Regular expressions
- Arrays of lengths other than 2, 3, or 4
- Mismatched type comparisons (e.g. 1.0 === false)

- Array index out of bounds

## 10.5. File extension and media type

---

Tileset styles use the .json extension and the application/json media type.



A

# ANNEX A (NORMATIVE) PROPERTIES REFERENCE

---

# ANNEX A (NORMATIVE) PROPERTIES REFERENCE

---

## A.1. Asset

---

Metadata about the entire tileset.

**Table A.1 — Asset Properties**

Type	Description	Required	
version	string	The 3D Tiles version. The version defines the JSON schema for the tileset JSON and the base set of tile formats.	✓ Yes
tileset Version	string	Application-specific version of this tileset, e.g., for when an existing tileset is updated.	No
extensions	extension	Dictionary object with extension-specific objects.	No
extras	extras	Application-specific data.	No

- Additional properties are allowed.
- **JSON schema:** asset.schema.json

### A.1.1. asset.version

The 3D Tiles version. The version defines the JSON schema for the tileset JSON and the base set of tile formats.

- **Type:** string
- **Required:** ✓ Yes

## A.1.2. asset.tilesetVersion

Application-specific version of this tileset, e.g., for when an existing tileset is updated.

- **Type:** string
- **Required:** No

## A.1.3. asset.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

## A.1.4. asset.extras

Application-specific data.

- **Type:** extras
- **Required:** No

## A.2. Availability

---

An object describing the availability of a set of elements.

**Table A.2 — Availability Properties**

TYPE	DESCRIPTION	REQUIRED
bitstream	integer  Index of a buffer view that indicates whether each element is available. The bitstream conforms to the boolean array encoding described in the 3D Metadata specification. If an element is available, its bit is 1, and if it is unavailable, its bit is 0.	No

Type	Description	Required
<b>availableCount</b>	integer A number indicating how many 1 bits exist in the availability bitstream.	No
<b>constant</b>	integer Integer indicating whether all of the elements are available (1) or all are unavailable (0).	No
<b>extensions</b>	extension Dictionary object with extension-specific objects.	No
<b>extras</b>	extras Application-specific data.	No

- Additional properties are allowed.
- JSON schema: `availability.schema.json`

## A.2.1. availability.bitstream

Index of a buffer view that indicates whether each element is available. The bitstream conforms to the boolean array encoding described in the 3D Metadata specification. If an element is available, its bit is 1, and if it is unavailable, its bit is 0.

- Type: integer
- Required: No
- Minimum:  $\geq 0$

## A.2.2. availability.availableCount

A number indicating how many 1 bits exist in the availability bitstream.

- Type: integer
- Required: No
- Minimum:  $\geq 0$

## A.2.3. availability.constant

Integer indicating whether all of the elements are available (1) or all are unavailable (0).

- Type: integer
- Required: No

- Allowed values:
  - 0 UNAVAILABLE
  - 1 AVAILABLE

## A.2.4. availability.extensions

Dictionary object with extension-specific objects.

- Type: extension
- Required: No
- Additional properties are allowed.
- Type of each property: object

## A.2.5. availability.extras

Application-specific data.

- Type: extras
- Required: No

## A.3. Batched 3D Model Feature Table

---

A set of Batched 3D Model semantics that contain additional information about features in a tile.

**Table A.3 — Batched 3D Model Feature Table Properties**

Type	Description	Required
BATCH_LENGTH	featureTable-definitions-globalPropertyInteger	A GlobalPropertyInteger object defining an integer property for all features. Details about this property are described in the 3D Tiles specification. ✓ Yes
RTC_CENTER	featureTable-definitions-globalPropertyCartesian3	A GlobalPropertyCartesian3 object defining a 3-component numeric property for all features. Details about this property are described in the 3D Tiles specification. No

Type	Description	Required
<code>extensions</code> extension	Dictionary object with extension-specific objects.	No
<code>extras</code> extras	Application-specific data.	No

- Additional properties are allowed.
- **JSON schema:** b3dm.featureTable.schema.json

### A.3.1. b3dm.featureTable.BATCH\_LENGTH

A GlobalPropertyInteger object defining an integer property for all features. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-globalPropertyInteger
- **Required:** ✓ Yes

### A.3.2. b3dm.featureTable.RTC\_CENTER

A GlobalPropertyCartesian3 object defining a 3-component numeric property for all features. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-globalPropertyCartesian3
- **Required:** No

### A.3.3. b3dm.featureTable.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- Additional properties are allowed.
- **Type of each property:** object

### A.3.4. b3dm.featureTable.extras

Application-specific data.

- **Type:** extras
- **Required:** No

## A.4. Batch Table

---

A set of properties defining application-specific metadata for features in a tile.

**Table A.4** — Batch Table Properties

	TYPE	DESCRIPTION	REQUIRED
extensions	extension	Dictionary object with extension-specific objects.	No
extras	extras	Application-specific data.	No

- **Additional properties are allowed.**
- **Type of additional properties:** batchTable-definitions-property
- **JSON schema:** batchTable.schema.json

### A.4.1. batchTable.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

## A.4.2. batchTable.extras

Application-specific data.

- **Type:** extras
- **Required:** No

## A.4.3. Batch Table Definitions

### A.4.3.1. batchTable-definitions-binaryBodyReference

An object defining the reference to a section of the binary body of the batch table where the property values are stored if not defined directly in the JSON.

- **Type:** rootProperty

**Table A.5 — BinaryBodyReference Properties**

	TYPE	DESCRIPTION	REQUIRED
byteOffset	integer	The offset into the buffer in bytes.	✓ Yes
componentType	string	The datatype of components in the property.	✓ Yes
type	string	Specifies if the property is a scalar or vector.	✓ Yes
extensions	extension	Dictionary object with extension-specific objects.	No
extras	extras	Application-specific data.	No

- Additional properties are allowed.

### A.4.3.2. batchTable-definitions-binaryBodyReference.byteOffset

The offset into the buffer in bytes.

- **Type:** integer
- **Required:** ✓ Yes

- **Minimum:**  $\geq 0$

#### A.4.3.3. batchTable-definitions-binaryBodyReference.componentType

The datatype of components in the property.

- **Type:** string
- **Required:** ✓ Yes
- **Allowed values:**
  - "BYTE"
  - "UNSIGNED\_BYTE"
  - "SHORT"
  - "UNSIGNED\_SHORT"
  - "INT"
  - "UNSIGNED\_INT"
  - "FLOAT"
  - "DOUBLE"

#### A.4.3.4. batchTable-definitions-binaryBodyReference.type

Specifies if the property is a scalar or vector.

- **Type:** string
- **Required:** ✓ Yes
- **Allowed values:**
  - "SCALAR"
  - "VEC2"
  - "VEC3"
  - "VEC4"

#### A.4.3.5. batchTable-definitions-binaryBodyReference.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

#### A.4.3.6. batchTable-definitions-binaryBodyReference.extras

Application-specific data.

- **Type:** extras
- **Required:** No

#### A.4.3.7. batchTable-definitions-property

A user-defined property which specifies per-feature application-specific metadata in a tile. Values either can be defined directly in the JSON as an array, or can refer to sections in the binary body with a BinaryBodyReference object.

- **Type:** One of batchTable-definitions-binaryBodyReference, array
- **Additional properties are allowed.**

## A.5. Bounding Volume

---

A bounding volume that encloses a tile or its content. At least one bounding volume property is required. Bounding volumes include box, region, or sphere.

**Table A.6 — Bounding Volume Properties**

TYPE	DESCRIPTION	REQUIRED
box	number [12] An array of 12 numbers that define an oriented bounding box. The first three elements define the x, y, and z values for the center of the box. The	No

Type	Description	Required
	next three elements (with indices 3, 4, and 5) define the x axis direction and half-length. The next three elements (indices 6, 7, and 8) define the y axis direction and half-length. The last three elements (indices 9, 10, and 11) define the z axis direction and half-length.	
region number [6]	An array of six numbers that define a bounding geographic region in EPSG:4979 coordinates with the order [west, south, east, north, minimum height, maximum height]. Longitudes and latitudes are in radians, and heights are in meters above (or below) the WGS84 ellipsoid.	No
sphere number [4]	An array of four numbers that define a bounding sphere. The first three elements define the x, y, and z values for the center of the sphere. The last element (with index 3) defines the radius in meters.	No
extensions extension	Dictionary object with extension-specific objects.	No
extras extras	Application-specific data.	No

- Additional properties are allowed.
- JSON schema: `boundingVolume.schema.json`

## A.5.1. `boundingVolume.box`

An array of 12 numbers that define an oriented bounding box. The first three elements define the x, y, and z values for the center of the box. The next three elements (with indices 3, 4, and 5) define the x axis direction and half-length. The next three elements (indices 6, 7, and 8) define the y axis direction and half-length. The last three elements (indices 9, 10, and 11) define the z axis direction and half-length.

- Type: number [12]
- Required: No

## A.5.2. `boundingVolume.region`

An array of six numbers that define a bounding geographic region in EPSG:4979 coordinates with the order [west, south, east, north, minimum height, maximum height]. Longitudes and latitudes are in radians, and heights are in meters above (or below) the WGS84 ellipsoid.

- Type: number [6]
- Required: No

### A.5.3. boundingVolume.sphere

An array of four numbers that define a bounding sphere. The first three elements define the x, y, and z values for the center of the sphere. The last element (with index 3) defines the radius in meters.

- **Type:** number [4]
- **Required:** No

### A.5.4. boundingVolume.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

### A.5.5. boundingVolume.extras

Application-specific data.

- **Type:** extras
- **Required:** No

## A.6. Buffer

---

A buffer is a binary blob. It is either the binary chunk of the subtree file, or an external buffer referenced by a URI.

**Table A.7 — Buffer Properties**

TYPE	DESCRIPTION	REQUIRED
uri	string	The URI (or IRI) of the external schema file. Relative paths are relative to the file containing the buffer JSON. uri is required when using the JSON

Type	Description		Required
subtree format and not required when using the binary subtree format — when omitted the buffer refers to the binary chunk of the subtree file. Data URIs are not allowed.			
byteLength	integer	The length of the buffer in bytes.	✓ Yes
name	string	The name of the buffer.	No
extensions	extension	Dictionary object with extension-specific objects.	No
extras	extras	Application-specific data.	No

- Additional properties are allowed.
- JSON schema: `buffer.schema.json`

### A.6.1. `buffer.uri`

The URI (or IRI) of the external schema file. Relative paths are relative to the file containing the buffer JSON. `uri` is required when using the JSON subtree format and not required when using the binary subtree format — when omitted the buffer refers to the binary chunk of the subtree file. Data URIs are not allowed.

- Type: `string`
- Required: No
- Format: `iri-reference`

### A.6.2. `buffer.byteLength`

The length of the buffer in bytes.

- Type: `integer`
- Required: ✓ Yes
- Minimum: `>= 1`

### A.6.3. `buffer.name`

The name of the buffer.

- **Type:** string
- **Required:** No
- **Minimum Length:** >= 1

## A.6.4. buffer.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

## A.6.5. buffer.extras

Application-specific data.

- **Type:** extras
- **Required:** No

# A.7. Buffer View

---

A contiguous subset of a buffer

**Table A.8 — Buffer View Properties**

TYPE	DESCRIPTION	REQUIRED
buffer	integer	The index of the buffer.
byteOffset	integer	The offset into the buffer in bytes.
byteLength	integer	The total byte length of the buffer view.
name	string	The name of the bufferView.
		No

Type	Description	Required
extensions	extension	Dictionary object with extension-specific objects.
extras	extras	Application-specific data.

- Additional properties are allowed.
- JSON schema: bufferView.schema.json

### A.7.1. bufferView.buffer

The index of the buffer.

- Type: integer
- Required: ✓ Yes
- Minimum: >= 0

### A.7.2. bufferView.byteOffset

The offset into the buffer in bytes.

- Type: integer
- Required: ✓ Yes
- Minimum: >= 0

### A.7.3. bufferView.byteLength

The total byte length of the buffer view.

- Type: integer
- Required: ✓ Yes
- Minimum: >= 1

### A.7.4. bufferView.name

The name of the bufferView.

- **Type:** string
- **Required:** No
- **Minimum Length:** >= 1

## A.7.5. bufferView.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

## A.7.6. bufferView.extras

Application-specific data.

- **Type:** extras
- **Required:** No

## A.8. Class

---

A class containing a set of properties.

**Table A.9 – Class Properties**

TYPE	DESCRIPTION	REQUIRED	
name	string	The name of the class, e.g. for display purposes.	No
description	string	The description of the class.	No
properties	object	A dictionary, where each key is a property ID and each value is an object defining the property. Property IDs shall be alphanumeric identifiers matching the regular expression ^[a-zA-Z_][a-zA-Z0-9_]*\$.	No
extensions	extension	Dictionary object with extension-specific objects.	No

TYPE	DESCRIPTION	REQUIRED
extras	extras Application-specific data.	No

- Additional properties are allowed.
- JSON schema: `class.schema.json`

### A.8.1. class.name

The name of the class, e.g. for display purposes.

- Type: string
- Required: No
- Minimum Length: `>= 1`

### A.8.2. class.description

The description of the class.

- Type: string
- Required: No
- Minimum Length: `>= 1`

### A.8.3. class.properties

A dictionary, where each key is a property ID and each value is an object defining the property. Property IDs shall be alphanumeric identifiers matching the regular expression `^[a-zA-Z_][a-zA-Z0-9_]*$`.

- Type: object
- Required: No
- Minimum number of properties: 1
- Additional properties are allowed.
- Type of each property: `class.property`

## A.8.4. class.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

## A.8.5. class.extras

Application-specific data.

- **Type:** extras
- **Required:** No

# A.9. Class Property

---

A single property of a metadata class.

**Table A.10 – Class Property Properties**

TYPE	DESCRIPTION	REQUIRED
name	string	The name of the property, e.g. for display purposes.
description	string	The description of the property.
type	string	The element type.
component Type	string	The datatype of the element's components. Only applicable to SCALAR, VECN, and MATN types.
enumType	string	Enum ID as declared in the enums dictionary. Required when type is ENUM.
array	boolean	Whether the property is an array. When count is defined the property is a fixed-length array. Otherwise the property is a variable-length array.

Type	Description	Required
count	integer The number of array elements. May only be defined when array is true.	No
normalized	boolean Specifies whether integer values are normalized. Only applicable to SCALAR, VECN, and MATN types with integer component types. For unsigned integer component types, values are normalized between [0.0, 1.0]. For signed integer component types, values are normalized between [-1.0, 1.0]. For all other component types, this property shall be false.	No, default: false
offset	definitions-definitions-numericValue An offset to apply to property values. Only applicable to SCALAR, VECN, and MATN types when the component type is FLOAT32 or FLOAT64, or when the property is normalized.	No
scale	definitions-definitions-numericValue A scale to apply to property values. Only applicable to SCALAR, VECN, and MATN types when the component type is FLOAT32 or FLOAT64, or when the property is normalized.	No
max	definitions-definitions-numericValue Maximum allowed value for the property. Only applicable to SCALAR, VECN, and MATN types. This is the maximum of all property values, after the transforms based on the normalized, offset, and scale properties have been applied.	No
min	definitions-definitions-numericValue Minimum allowed value for the property. Only applicable to SCALAR, VECN, and MATN types. This is the minimum of all property values, after the transforms based on the normalized, offset, and scale properties have been applied.	No
required	boolean If required, the property shall be present in every entity conforming to the class. If not required, individual entities may include noData values, or the entire property may be omitted. As a result, noData has no effect on a required property. Client implementations may use required properties to make performance optimizations.	No, default: false
noData	definitions-definitions-noDataValue A noData value represents missing data — also known as a sentinel value — wherever it appears. BOOLEAN properties may not specify noData values. This is given as the plain property value, without the transforms from the normalized, offset, and scale properties. Shall not be defined if required is true.	No
default	definitions-definitions-anyValue A default value to use when encountering a noData value or an omitted property. The value is given in its final form, taking the effect of normalized, offset, and scale properties into account. Shall not be defined if required is true.	No

Type	Description	Required	
semantic	string	An identifier that describes how this property should be interpreted. The semantic cannot be used by other properties in the class.	No
extensions	extension	Dictionary object with extension-specific objects.	No
extras	extras	Application-specific data.	No

- Additional properties are allowed.
- JSON schema: `class.property.schema.json`

### A.9.1. `class.property.name`

The name of the property, e.g. for display purposes.

- Type: string
- Required: No
- Minimum Length:  $\geq 1$

### A.9.2. `class.property.description`

The description of the property.

- Type: string
- Required: No
- Minimum Length:  $\geq 1$

### A.9.3. `class.property.type`

The element type.

- Type: string
- Required: ✓ Yes

- **Allowed values:**

- "SCALAR"
- "VEC2"
- "VEC3"
- "VEC4"
- "MAT2"
- "MAT3"
- "MAT4"
- "STRING"
- "BOOLEAN"
- "ENUM"

#### A.9.4. class.property.componentType

The datatype of the element's components. Only applicable to SCALAR, VECN, and MATN types.

- **Type:** string
- **Required:** No
- **Allowed values:**

- "INT8"
- "UINT8"
- "INT16"
- "UINT16"
- "INT32"
- "UINT32"
- "INT64"
- "UINT64"
- "FLOAT32"
- "FLOAT64"

## A.9.5. class.property.enumType

Enum ID as declared in the enums dictionary. Required when type is ENUM.

- **Type:** string
- **Required:** No

## A.9.6. class.property.array

Whether the property is an array. When count is defined the property is a fixed-length array. Otherwise the property is a variable-length array.

- **Type:** boolean
- **Required:** No, default: false

## A.9.7. class.property.count

The number of array elements. May only be defined when array is true.

- **Type:** integer
- **Required:** No
- **Minimum:** >= 2

## A.9.8. class.property.normalized

Specifies whether integer values are normalized. Only applicable to SCALAR, VECN, and MATN types with integer component types. For unsigned integer component types, values are normalized between [0.0, 1.0]. For signed integer component types, values are normalized between [-1.0, 1.0]. For all other component types, this property shall be false.

- **Type:** boolean
- **Required:** No, default: false

## A.9.9. class.property.offset

An offset to apply to property values. Only applicable to SCALAR, VECN, and MATN types when the component type is FLOAT32 or FLOAT64, or when the property is normalized.

- **Type:** definitions-definitions-numericValue
- **Required:** No

## A.9.10. class.property.scale

A scale to apply to property values. Only applicable to SCALAR, VECN, and MATN types when the component type is FLOAT32 or FLOAT64, or when the property is normalized.

- **Type:** definitions-definitions-numericValue
- **Required:** No

## A.9.11. class.property.max

Maximum allowed value for the property. Only applicable to SCALAR, VECN, and MATN types. This is the maximum of all property values, after the transforms based on the normalized, offset, and scale properties have been applied.

- **Type:** definitions-definitions-numericValue
- **Required:** No

## A.9.12. class.property.min

Minimum allowed value for the property. Only applicable to SCALAR, VECN, and MATN types. This is the minimum of all property values, after the transforms based on the normalized, offset, and scale properties have been applied.

- **Type:** definitions-definitions-numericValue
- **Required:** No

## A.9.13. class.property.required

If required, the property shall be present in every entity conforming to the class. If not required, individual entities may include noData values, or the entire property may be omitted. As a result, noData has no effect on a required property. Client implementations may use required properties to make performance optimizations.

- **Type:** boolean
- **Required:** No, default: false

## A.9.14. class.property.noData

A noData value represents missing data — also known as a sentinel value — wherever it appears. BOOLEAN properties may not specify noData values. This is given as the plain property value, without the transforms from the normalized, offset, and scale properties. Shall not be defined if required is true.

- **Type:** definitions-definitions-noDataValue
- **Required:** No

## A.9.15. class.property.default

A default value to use when encountering a noData value or an omitted property. The value is given in its final form, taking the effect of normalized, offset, and scale properties into account. Shall not be defined if required is true.

- **Type:** definitions-definitions-anyValue
- **Required:** No

## A.9.16. class.property.semantic

An identifier that describes how this property should be interpreted. The semantic cannot be used by other properties in the class.

- **Type:** string
- **Required:** No
- **Minimum Length:** >= 1

## A.9.17. class.property.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

## A.9.18. class.property.extras

Application-specific data.

- **Type:** extras
- **Required:** No

## A.10. Content

Metadata about the tile's content and a link to the content.

**Table A.11** – Content Properties

	TYPE	DESCRIPTION	REQUIRED
boundingVolume	boundingVolume	An optional bounding volume that tightly encloses tile content. tile.boundingVolume provides spatial coherence and tile.content.boundingVolume enables tight view frustum culling. When this is omitted, tile.boundingVolume is used.	No
uri	string	A uri that points to tile content. When the uri is relative, it is relative to the referring tileset JSON file.	✓ Yes
metadata	metadataEntity	Metadata that is associated with this content.	No
group	integer	The group this content belongs to. The value is an index into the array of groups that is defined for the containing tileset.	No
extensions	extension	Dictionary object with extension-specific objects.	No
extras	extras	Application-specific data.	No

- Additional properties are allowed.
- **JSON schema:** content.schema.json

### A.10.1. content.boundingVolume

An optional bounding volume that tightly encloses tile content. tile.boundingVolume provides spatial coherence and tile.content.boundingVolume enables tight view frustum culling. When this is omitted, tile.boundingVolume is used.

- **Type:** boundingVolume
- **Required:** No
- **Minimum number of properties:** 1

## A.10.2. content.uri

A uri that points to tile content. When the uri is relative, it is relative to the referring tileset JSON file.

- **Type:** string
- **Required:** ✓ Yes

## A.10.3. content.metadata

Metadata that is associated with this content.

- **Type:** metadataEntity
- **Required:** No

## A.10.4. content.group

The group this content belongs to. The value is an index into the array of groups that is defined for the containing tileset.

- **Type:** integer
- **Required:** No
- **Minimum:** >= 0

## A.10.5. content.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

## A.10.6. content.extras

Application-specific data.

- **Type:** extras
- **Required:** No

## A.11. Definitions

---

Common definitions used in schema files.

- **Additional properties are allowed.**
- **JSON schema:** definitions.schema.json

### A.11.1. Definitions Definitions

#### A.11.1.1. definitions-definitions-numericArray1D

An array of numeric values

- **Type:** number [1-\*]
- **Minimum items:** 1
- **Additional properties are allowed.**

#### A.11.1.2. definitions-definitions-numericArray2D

An array of arrays of numeric values

- **Type:** definitions-definitions-numericArray1D [1-\*]
- **Minimum items:** 1
- **Additional properties are allowed.**

### A.11.1.3. definitions-definitions-booleanArray1D

An array of boolean values

- **Type:** boolean [1-\*]
- **Minimum items:** 1
- **Additional properties are allowed.**

### A.11.1.4. definitions-definitions-stringArray1D

An array of string values

- **Type:** string [1-\*]
- **Minimum items:** 1
- **Additional properties are allowed.**

### A.11.1.5. definitions-definitions-numericValue

For SCALAR this is a number. For VECN this is an array of N numbers. For MATN this is an array of  $N^2$  numbers. For fixed-length arrays this is an array of count elements of the given type.

- **Type:** One of number, definitions-definitions-numericArray1D, definitions-definitions-numericArray2D
- **Additional properties are allowed.**

### A.11.1.6. definitions-definitions-noDataValue

For SCALAR this is a number. For STRING this is a string. For ENUM this is a string that shall be a valid enum name, not an integer value. For VECN this is an array of N numbers. For MATN this is an array of  $N^2$  numbers. For fixed-length arrays this is an array of count elements of the given type.

- **Type:** One of definitions-definitions-numericValue, string, definitions-definitions-stringArray1D
- **Additional properties are allowed.**

### A.11.1.7. definitions-definitions-anyValue

For SCALAR this is a number. For STRING this is a string. For ENUM this is a string that shall be a valid enum name, not an integer value. For BOOLEAN this is a boolean. For VECN this is an array of N numbers. For MATN this is an array of  $N^2$  numbers. For fixed-length array this is an array of count elements of the given type. For variable-length arrays this is an array of any length of the given type.

- **Type:** One of definitions-definitions-numericValue, string, definitions-definitions-stringArray1D, boolean, definitions-definitions-booleanArray1D
- **Additional properties are allowed.**

## A.12. Enum

---

An object defining the values of an enum.

**Table A.12 — Enum Properties**

TYPE	DESCRIPTION	REQUIRED	
name	string	The name of the enum, e.g. for display purposes.	No
description	string	The description of the enum.	No
valueType	string	The type of the integer enum value.	No, default: UINT16
values	enum.value [1- *]	An array of enum values. Duplicate names or duplicate integer values are not allowed.	✓ Yes
extensions	extension	Dictionary object with extension-specific objects.	No
extras	extras	Application-specific data.	No

- **Additional properties are allowed.**
- **JSON schema:** enum.schema.json

### A.12.1. enum.name

The name of the enum, e.g. for display purposes.

- **Type:** string
- **Required:** No
- **Minimum Length:** >= 1

### A.12.2. enum.description

The description of the enum.

- **Type:** string
- **Required:** No
- **Minimum Length:** >= 1

### A.12.3. enum.valueType

The type of the integer enum value.

- **Type:** string
- **Required:** No, default: UINT16
- **Allowed values:**
  - "INT8"
  - "UINT8"
  - "INT16"
  - "UINT16"
  - "INT32"
  - "UINT32"
  - "INT64"
  - "UINT64"

## A.12.4. enum.values

An array of enum values. Duplicate names or duplicate integer values are not allowed.

- **Type:** enum.value [1-\*]
- **Required:** ✓ Yes

## A.12.5. enum.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

## A.12.6. enum.extras

Application-specific data.

- **Type:** extras
- **Required:** No

# A.13. Enum value

---

An enum value.

**Table A.13 – Enum value Properties**

TYPE	DESCRIPTION	REQUIRED
name	string	The name of the enum value.
description	string	The description of the enum value.

Type	Description	Required
value	integer	The integer enum value.
extensions	extension	Dictionary object with extension-specific objects.
extras	extras	Application-specific data.

- Additional properties are allowed.
- JSON schema: enum.value.schema.json

### A.13.1. enum.value.name

The name of the enum value.

- Type: string
- Required: ✓ Yes
- Minimum Length: >= 1

### A.13.2. enum.value.description

The description of the enum value.

- Type: string
- Required: No
- Minimum Length: >= 1

### A.13.3. enum.value.value

The integer enum value.

- Type: integer
- Required: ✓ Yes

#### A.13.4. enum.value.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

#### A.13.5. enum.value.extras

Application-specific data.

- **Type:** extras
- **Required:** No

### A.14. Extension

---

Dictionary object with extension-specific objects.

- **Additional properties are allowed.**
- **Type of additional properties:** object
- **JSON schema:** extension.schema.json

### A.15. Extras

---

Application-specific data.

- **Additional properties are allowed.**
- **JSON schema:** extras.schema.json

## A.16. Feature Table

---

A set of semantics containing per-tile and per-feature values defining the position and appearance properties for features in a tile.

**Table A.14 — Feature Table Properties**

Type	Description	Required	
extensions	extension	Dictionary object with extension-specific objects.	No
extras	extras	Application-specific data.	No

- Additional properties are allowed.
- Type of additional properties: featureTable-definitions-property
- JSON schema: featureTable.schema.json

### A.16.1. featureTable.extensions

Dictionary object with extension-specific objects.

- Type: extension
- Required: No
- Additional properties are allowed.
- Type of each property: object

### A.16.2. featureTable.extras

Application-specific data.

- Type: extras
- Required: No

### A.16.3. Feature Table Definitions

#### A.16.3.1. featureTable-definitions-binaryBodyOffset

An object defining the offset into a section of the binary body of the features table where the property values are stored if not defined directly in the JSON.

- **Type:** rootProperty

**Table A.15 – BinaryBodyOffset Properties**

	TYPE	DESCRIPTION	REQUIRED
byteOffset	integer	The offset into the buffer in bytes.	✓ Yes
extensions	extension	Dictionary object with extension-specific objects.	No
extras	extras	Application-specific data.	No

- Additional properties are allowed.

#### A.16.3.2. featureTable-definitions-binaryBodyOffset.byteOffset

The offset into the buffer in bytes.

- **Type:** integer
- **Required:** ✓ Yes
- **Minimum:** >= 0

#### A.16.3.3. featureTable-definitions-binaryBodyOffset.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- Additional properties are allowed.

- **Type of each property:** object

#### A.16.3.4. featureTable-definitions-binaryBodyOffset.extras

Application-specific data.

- **Type:** extras
- **Required:** No

#### A.16.3.5. featureTable-definitions-binaryBodyReference

An object defining the reference to a section of the binary body of the features table where the property values are stored if not defined directly in the JSON.

- **Type:** featureTable-definitions-binaryBodyOffset

**Table A.16** – BinaryBodyReference Properties

TYPE	DESCRIPTION		REQUIRED
component Type	string	The datatype of components in the property. This is defined only if the semantic allows for overriding the implicit component type. These cases are specified in each tile format.	No
byteOffset	integer	The offset into the buffer in bytes.	✓ Yes
extensions	extension	Dictionary object with extension-specific objects.	No
extras	extras	Application-specific data.	No

- Additional properties are allowed.

#### A.16.3.6. featureTable-definitions-binaryBodyReference.componentType

The datatype of components in the property. This is defined only if the semantic allows for overriding the implicit component type. These cases are specified in each tile format.

- **Type:** string
- **Required:** No

- **Allowed values:**
  - "BYTE"
  - "UNSIGNED\_BYTE"
  - "SHORT"
  - "UNSIGNED\_SHORT"
  - "INT"
  - "UNSIGNED\_INT"
  - "FLOAT"
  - "DOUBLE"

#### A.16.3.7. featureTable-definitions-binaryBodyReference.byteOffset

The offset into the buffer in bytes.

- **Type:** integer
- **Required:** ✓ Yes
- **Minimum:** >= 0

#### A.16.3.8. featureTable-definitions-binaryBodyReference.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

#### A.16.3.9. featureTable-definitions-binaryBodyReference.extras

Application-specific data.

- **Type:** extras
- **Required:** No

### A.16.3.10. featureTable-definitions-property

A user-defined property which specifies application-specific metadata in a tile. Values can refer to sections in the binary body with a BinaryBodyReference object. Global values can be also be defined directly in the JSON.

- **Type:** One of featureTable-definitions-binaryBodyReference, featureTable-definitions-globalPropertyBoolean, featureTable-definitions-globalPropertyInteger, featureTable-definitions-globalPropertyNumber, featureTable-definitions-globalPropertyCartesian3, featureTable-definitions-globalPropertyCartesian4
- **Additional properties are allowed.**

### A.16.3.11. featureTable-definitions-globalPropertyBoolean

An object defining a global boolean property value for all features.

- **Type:** boolean
- **Additional properties are allowed.**

### A.16.3.12. featureTable-definitions-globalPropertyInteger

An object defining a global integer property value for all features.

- **Type:** One of featureTable-definitions-binaryBodyOffset, integer
- **Additional properties are allowed.**

### A.16.3.13. featureTable-definitions-globalPropertyNumber

An object defining a global numeric property value for all features.

- **Type:** One of featureTable-definitions-binaryBodyOffset, number
- **Additional properties are allowed.**

### A.16.3.14. featureTable-definitions-globalPropertyCartesian3

An object defining a global 3-component numeric property values for all features.

- **Type:** One of featureTable-definitions-binaryBodyOffset, number [3]
- **Additional properties are allowed.**

### A.16.3.15. featureTable-definitions-globalPropertyCartesian4

An object defining a global 4-component numeric property values for all features.

- **Type:** One of featureTable-definitions-binaryBodyOffset, number [4]
- **Additional properties are allowed.**

## A.17. Group Metadata

---

An object containing metadata about a group.

**Table A.17 — Group Metadata Properties**

TYPE	DESCRIPTION	REQUIRED
class string	The class that property values conform to. The value shall be a class ID declared in the classes dictionary of the metadata schema.	✓ Yes
properties object	A dictionary, where each key corresponds to a property ID in the class' properties dictionary and each value contains the property values. The type of the value shall match the property definition: For BOOLEAN use true or false. For STRING use a JSON string. For numeric types use a JSON number. For ENUM use a valid enum name, not an integer value. For ARRAY, VECN, and MATN types use a JSON array containing values matching the componentType. Required properties shall be included in this dictionary.	No
extensions extension	Dictionary object with extension-specific objects.	No
extras extras	Application-specific data.	No

- **Additional properties are allowed.**
- **JSON schema:** group.schema.json

## A.17.1. group.class

The class that property values conform to. The value shall be a class ID declared in the classes dictionary of the metadata schema.

- **Type:** string
- **Required:** ✓ Yes

## A.17.2. group.properties

A dictionary, where each key corresponds to a property ID in the class' properties dictionary and each value contains the property values. The type of the value shall match the property definition: For BOOLEAN use true or false. For STRING use a JSON string. For numeric types use a JSON number. For ENUM use a valid enum name, not an integer value. For ARRAY, VECN, and MATN types use a JSON array containing values matching the componentType. Required properties shall be included in this dictionary.

- **Type:** object
- **Required:** No
- **Minimum number of properties:** 1
- **Additional properties are allowed.**
- **Type of each property:** definitions-definitions-anyValue

## A.17.3. group.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

## A.17.4. group.extras

Application-specific data.

- **Type:** extras
- **Required:** No

## A.18. Instanced 3D Model Feature Table

---

A set of Instanced 3D Model semantics that contains values defining the position and appearance properties for instanced models in a tile.

**Table A.18 – Instanced 3D Model Feature Table Properties**

TYPE	DESCRIPTION	REQUIRED
POSITION	featureTable-definitions-binaryBodyReference  A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.	No
POSITION_QUANTIZED	featureTable-definitions-binaryBodyReference  A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.	No
NORMAL_UP	featureTable-definitions-binaryBodyReference  A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.	No
NORMAL_RIGHT	featureTable-definitions-binaryBodyReference  A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.	No
NORMAL_UP_OCT32P	featureTable-definitions-binaryBodyReference  A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.	No
NORMAL_RIGHT_OCT32P	featureTable-definitions-binaryBodyReference  A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.	No

	TYPE	DESCRIPTION	REQUIRED
SCALE	featureTable-definitions-binaryBodyReference	A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.	No
SCALE_NON_UNIFORM	featureTable-definitions-binaryBodyReference	A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.	No
BATCH_ID	featureTable-definitions-binaryBodyReference	A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.	No
INSTANCES_LENGTH	featureTable-definitions-globalPropertyInteger	A GlobalPropertyInteger object defining an integer property for all features. Details about this property are described in the 3D Tiles specification.	✓ Yes
RTC_CENTER	featureTable-definitions-globalPropertyCartesian3	A GlobalPropertyCartesian3 object defining a 3-component numeric property for all features. Details about this property are described in the 3D Tiles specification.	No
QUANTIZED_VOLUME_OFFSET	featureTable-definitions-globalPropertyCartesian3	A GlobalPropertyCartesian3 object defining a 3-component numeric property for all features. Details about this property are described in the 3D Tiles specification.	No
QUANTIZED_VOLUME_SCALE	featureTable-definitions-globalPropertyCartesian3	A GlobalPropertyCartesian3 object defining a 3-component numeric property for all features. Details about this property are described in the 3D Tiles specification.	No
EAST_NORTH_UP	featureTable-definitions-globalPropertyBoolean	A GlobalPropertyBoolean object defining a boolean property for all features. Details about this property are described in the 3D Tiles specification.	No
extensions	extension	Dictionary object with extension-specific objects.	No
extras	extras	Application-specific data.	No

- Additional properties are allowed.
- JSON schema: i3dm.featureTable.schema.json

## A.18.1. i3dm.featureTable.POSITION

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-binaryBodyReference
- **Required:** No

## A.18.2. i3dm.featureTable.POSITION\_QUANTIZED

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-binaryBodyReference
- **Required:** No

## A.18.3. i3dm.featureTable.NORMAL\_UP

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-binaryBodyReference
- **Required:** No

## A.18.4. i3dm.featureTable.NORMAL\_RIGHT

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-binaryBodyReference
- **Required:** No

## A.18.5. i3dm.featureTable.NORMAL\_UP\_OCT32P

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-binaryBodyReference
- **Required:** No

## A.18.6. i3dm.featureTable.NORMAL\_RIGHT\_OCT32P

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-binaryBodyReference
- **Required:** No

## A.18.7. i3dm.featureTable.SCALE

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-binaryBodyReference
- **Required:** No

## A.18.8. i3dm.featureTable.SCALE\_NON\_UNIFORM

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-binaryBodyReference
- **Required:** No

## A.18.9. i3dm.featureTable.BATCH\_ID

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-binaryBodyReference
- **Required:** No

## A.18.10. i3dm.featureTable.INSTANCES\_LENGTH

A GlobalPropertyInteger object defining an integer property for all features. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-globalPropertyInteger
- **Required:** ✓ Yes

## A.18.11. i3dm.featureTable.RTC\_CENTER

A GlobalPropertyCartesian3 object defining a 3-component numeric property for all features. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-globalPropertyCartesian3
- **Required:** No

## A.18.12. i3dm.featureTable.QUANTIZED\_VOLUME\_OFFSET

A GlobalPropertyCartesian3 object defining a 3-component numeric property for all features. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-globalPropertyCartesian3
- **Required:** No

## A.18.13. i3dm.featureTable.QUANTIZED\_VOLUME\_SCALE

A GlobalPropertyCartesian3 object defining a 3-component numeric property for all features. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-globalPropertyCartesian3
- **Required:** No

## A.18.14. i3dm.featureTable.EAST\_NORTH\_UP

A GlobalPropertyBoolean object defining a boolean property for all features. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-globalPropertyBoolean
- **Required:** No

## A.18.15. i3dm.featureTable.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

## A.18.16. i3dm.featureTable.extras

Application-specific data.

- **Type:** extras
- **Required:** No

# A.19. Metadata Entity

---

An object containing a reference to a class from a metadata schema, and property values that conform to the properties of that class.

**Table A.19** — Metadata Entity Properties

Type	Description	Required	
class	string	The class that property values conform to. The value shall be a class ID declared in the classes dictionary of the metadata schema.	✓ Yes
properties	object	A dictionary, where each key corresponds to a property ID in the class' properties dictionary and each value contains the property values. The type of the value shall match the property definition: For BOOLEAN use true or false. For STRING use a JSON string. For numeric types use a JSON number. For ENUM use a valid enum name, not an integer value. For ARRAY, VECN, and MATN types use a JSON array containing values matching the componentType. Required properties shall be included in this dictionary.	No
extensions	extension	Dictionary object with extension-specific objects.	No
extras	extras	Application-specific data.	No

- Additional properties are allowed.
- **JSON schema:** `metadataEntity.schema.json`

### A.19.1. `metadataEntity.class`

The class that property values conform to. The value shall be a class ID declared in the classes dictionary of the metadata schema.

- **Type:** string
- **Required:** ✓ Yes

### A.19.2. `metadataEntity.properties`

A dictionary, where each key corresponds to a property ID in the class' properties dictionary and each value contains the property values. The type of the value shall match the property definition: For BOOLEAN use true or false. For STRING use a JSON string. For numeric types use a JSON number. For ENUM use a valid enum name, not an integer value. For ARRAY, VECN, and MATN types use a JSON array containing values matching the componentType. Required properties shall be included in this dictionary.

- **Type:** object
- **Required:** No
- **Minimum number of properties:** 1

- Additional properties are allowed.
- Type of each property: definitions-definitions-anyValue

### A.19.3. metadataEntity.extensions

Dictionary object with extension-specific objects.

- Type: extension
- Required: No
- Additional properties are allowed.
- Type of each property: object

### A.19.4. metadataEntity.extras

Application-specific data.

- Type: extras
- Required: No

## A.20. Point Cloud Feature Table

---

A set of Point Cloud semantics that contains values defining the position and appearance properties for points in a tile.

**Table A.20 — Point Cloud Feature Table Properties**

Type	Description	Required
POSITION	featureTable-definitions-binaryBodyReference A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.	No
POSITION_QUANTIZED	featureTable-definitions-binaryBodyReference A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details	No

Type	Description	Required
RGBA	A <code>BinaryBodyReference</code> object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.	No
RGB	A <code>BinaryBodyReference</code> object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.	No
RGB565	A <code>BinaryBodyReference</code> object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.	No
NORMAL	A <code>BinaryBodyReference</code> object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.	No
NORMAL_OCT16P	A <code>BinaryBodyReference</code> object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.	No
BATCH_ID	A <code>BinaryBodyReference</code> object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.	No
POINTS_LENGTH	A <code>GlobalPropertyInteger</code> object defining an integer property for all points. Details about this property are described in the 3D Tiles specification.	✓ Yes
RTC_CENTER	A <code>GlobalPropertyCartesian3</code> object defining a 3-component numeric property for all points. Details about this property are described in the 3D Tiles specification.	No
QUANTIZED_VOLUME_OFFSET	A <code>GlobalPropertyCartesian3</code> object defining a 3-component numeric property	No

TYPE	DESCRIPTION	REQUIRED
	for all points. Details about this property are described in the 3D Tiles specification.	
QUANTIZED_VOLUME_SCALE	featureTable-definitions-globalPropertyCartesian3 A GlobalPropertyCartesian3 object defining a 3-component numeric property for all points. Details about this property are described in the 3D Tiles specification.	No
CONSTANT_RGB	featureTable-definitions-globalPropertyCartesian4 A GlobalPropertyCartesian4 object defining a 4-component numeric property for all points. Details about this property are described in the 3D Tiles specification.	No
BATCH_LENGTH	featureTable-definitions-globalPropertyInteger A GlobalPropertyInteger object defining an integer property for all points. Details about this property are described in the 3D Tiles specification.	No
extensions	extension Dictionary object with extension-specific objects.	No
extras	extras Application-specific data.	No

- Additional properties are allowed.
- JSON schema: pnts.featureTable.schema.json

## A.20.1. pnts.featureTable.POSITION

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.

- Type: featureTable-definitions-binaryBodyReference
- Required: No

## A.20.2. pnts.featureTable.POSITION\_QUANTIZED

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.

- Type: featureTable-definitions-binaryBodyReference
- Required: No

### A.20.3. pnts.featureTable.RGBA

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-binaryBodyReference
- **Required:** No

### A.20.4. pnts.featureTable.RGB

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-binaryBodyReference
- **Required:** No

### A.20.5. pnts.featureTable.RGB565

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-binaryBodyReference
- **Required:** No

### A.20.6. pnts.featureTable.NORMAL

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-binaryBodyReference
- **Required:** No

## A.20.7. pnts.featureTable.NORMAL\_OCT16P

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-binaryBodyReference
- **Required:** No

## A.20.8. pnts.featureTable.BATCH\_ID

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-binaryBodyReference
- **Required:** No

## A.20.9. pnts.featureTable.POINTS\_LENGTH

A GlobalPropertyInteger object defining an integer property for all points. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-globalPropertyInteger
- **Required:** ✓ Yes

## A.20.10. pnts.featureTable.RTC\_CENTER

A GlobalPropertyCartesian3 object defining a 3-component numeric property for all points. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-globalPropertyCartesian3
- **Required:** No

## A.20.11. pnts.featureTable.QUANTIZED\_VOLUME\_OFFSET

A GlobalPropertyCartesian3 object defining a 3-component numeric property for all points. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-globalPropertyCartesian3
- **Required:** No

## A.20.12. pnts.featureTable.QUANTIZED\_VOLUME\_SCALE

A GlobalPropertyCartesian3 object defining a 3-component numeric property for all points. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-globalPropertyCartesian3
- **Required:** No

## A.20.13. pnts.featureTable.CONSTANT\_RGBA

A GlobalPropertyCartesian4 object defining a 4-component numeric property for all points. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-globalPropertyCartesian4
- **Required:** No

## A.20.14. pnts.featureTable.BATCH\_LENGTH

A GlobalPropertyInteger object defining an integer property for all points. Details about this property are described in the 3D Tiles specification.

- **Type:** featureTable-definitions-globalPropertyInteger
- **Required:** No

## A.20.15. pnts.featureTable.extensions

Dictionary object with extension-specific objects.

- **Type:** extension

- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

## A.20.16. pnts.featureTable.extras

Application-specific data.

- **Type:** extras
- **Required:** No

# A.21. Properties

---

A dictionary object of metadata about per-feature properties.

**Table A.21 — Properties Properties**

TYPE	DESCRIPTION	REQUIRED
maximum	number	The maximum value of this property of all the features in the tileset.
minimum	number	The minimum value of this property of all the features in the tileset.
extensions	extension	Dictionary object with extension-specific objects.
extras	extras	Application-specific data.

- **Additional properties are allowed.**
- **JSON schema:** properties.schema.json

## A.21.1. properties.maximum

The maximum value of this property of all the features in the tileset.

- **Type:** number
- **Required:** ✓ Yes

## A.21.2. properties.minimum

The minimum value of this property of all the features in the tileset.

- **Type:** number
- **Required:** ✓ Yes

## A.21.3. properties.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

## A.21.4. properties.extras

Application-specific data.

- **Type:** extras
- **Required:** No

## A.22. Property Table

---

Properties conforming to a class, organized as property values stored in binary columnar arrays.

**Table A.22 — Property Table Properties**

TYPE	DESCRIPTION	REQUIRED
name	string	The name of the property table, e.g. for display purposes.
class	string	The class that property values conform to. The value shall be a class ID declared in the classes dictionary.

Type	Description	Required
count	integer	The number of elements in each property array. ✓ Yes
properties	object	A dictionary, where each key corresponds to a property ID in the class' properties dictionary and each value is an object describing where property values are stored. Required properties shall be included in this dictionary. No
extensions	extension	Dictionary object with extension-specific objects. No
extras	extras	Application-specific data. No

- Additional properties are allowed.
- JSON schema: propertyTable.schema.json

### A.22.1. propertyTable.name

The name of the property table, e.g. for display purposes.

- Type: string
- Required: No
- Minimum Length: >= 1

### A.22.2. propertyTable.class

The class that property values conform to. The value shall be a class ID declared in the classes dictionary.

- Type: string
- Required: ✓ Yes

### A.22.3. propertyTable.count

The number of elements in each property array.

- Type: integer
- Required: ✓ Yes
- Minimum: >= 1

## A.22.4. `propertyTable.properties`

A dictionary, where each key corresponds to a property ID in the class' properties dictionary and each value is an object describing where property values are stored. Required properties shall be included in this dictionary.

- **Type:** object
- **Required:** No
- **Minimum number of properties:** 1
- **Additional properties are allowed.**
- **Type of each property:** `propertyTable.property`

## A.22.5. `propertyTable.extensions`

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

## A.22.6. `propertyTable.extras`

Application-specific data.

- **Type:** extras
- **Required:** No

# A.23. Property Table Property

---

An array of binary property values. This represents one column of a property table, and contains one value of a certain property for each metadata entity.

**Table A.23 – Property Table Property Properties**

Type	Description	Required
values	The index of the buffer view containing property values. The data type of property values is determined by the property definition: When type is BOOLEAN values are packed into a bitstream. When type is STRING values are stored as byte sequences and decoded as UTF-8 strings. When type is SCALAR, VECN, or MATN the values are stored as the provided componentType and the buffer view byteOffset shall be aligned to a multiple of the componentType size. When type is ENUM values are stored as the enum's valueType and the buffer view byteOffset shall be aligned to a multiple of the valueType size. Each enum value in the array shall match one of the allowed values in the enum definition. arrayOffsets is required for variable-length arrays and stringOffsets is required for strings (for variable-length arrays of strings, both are required).	✓ Yes
arrayOffsets	The index of the buffer view containing offsets for variable-length arrays. The number of offsets is equal to the property table count plus one. The offsets represent the start positions of each array, with the last offset representing the position after the last array. The array length is computed using the difference between the subsequent offset and the current offset. If type is STRING the offsets index into the string offsets array (stored in stringOffsets), otherwise they index into the property array (stored in values). The data type of these offsets is determined by arrayOffsetType. The buffer view byteOffset shall be aligned to a multiple of the arrayOffsetType size.	No
stringOffsets	The index of the buffer view containing offsets for strings. The number of offsets is equal to the number of string elements plus one. The offsets represent the byte offsets of each string in the property array (stored in values), with the last offset representing the byte offset after the last string. The string byte length is computed using the difference between the subsequent offset and the current offset. The data type of these offsets is determined by stringOffsetType. The buffer view byteOffset shall be aligned to a multiple of the stringOffsetType size.	No
arrayOffsetType	The type of values in arrayOffsets.	No, default: UINT32
stringOffset Type	The type of values in stringOffsets.	No, default: UINT32

	TYPE	DESCRIPTION	REQUIRED
offset	definitions-definitions-numericValue	An offset to apply to property values. Only applicable when the component type is FLOAT32 or FLOAT64, or when the property is normalized. Overrides the class property's offset if both are defined.	No
scale	definitions-definitions-numericValue	A scale to apply to property values. Only applicable when the component type is FLOAT32 or FLOAT64, or when the property is normalized. Overrides the class property's scale if both are defined.	No
max	definitions-definitions-numericValue	Maximum value present in the property values. Only applicable to SCALAR, VECN, and MATN types. This is the maximum of all property values, after the transforms based on the normalized, offset, and scale properties have been applied.	No
min	definitions-definitions-numericValue	Minimum value present in the property values. Only applicable to SCALAR, VECN, and MATN types. This is the minimum of all property values, after the transforms based on the normalized, offset, and scale properties have been applied.	No
extensions	extension	Dictionary object with extension-specific objects.	No
extras	extras	Application-specific data.	No

- Additional properties are allowed.
- JSON schema: `propertyTable.property.schema.json`

### A.23.1. `propertyTable.property.values`

The index of the buffer view containing property values. The data type of property values is determined by the property definition: When type is BOOLEAN values are packed into a bitstream. When type is STRING values are stored as byte sequences and decoded as UTF-8 strings. When type is SCALAR, VECN, or MATN the values are stored as the provided componentType and the buffer view byteOffset shall be aligned to a multiple of the componentType size. When type is ENUM values are stored as the enum's valueType and the buffer view byteOffset shall be aligned to a multiple of the valueType size. Each enum value in the array shall match one of the allowed values in the enum definition. arrayOffsets is required for variable-length arrays and stringOffsets is required for strings (for variable-length arrays of strings, both are required).

- Type: integer
- Required: ✓ Yes

- **Minimum:** >= 0

## A.23.2. `propertyTable.property.arrayOffsets`

The index of the buffer view containing offsets for variable-length arrays. The number of offsets is equal to the property table count plus one. The offsets represent the start positions of each array, with the last offset representing the position after the last array. The array length is computed using the difference between the subsequent offset and the current offset. If type is STRING the offsets index into the string offsets array (stored in `stringOffsets`), otherwise they index into the property array (stored in `values`). The data type of these offsets is determined by `arrayOffsetType`. The buffer view `byteOffset` shall be aligned to a multiple of the `arrayOffsetType` size.

- **Type:** integer
- **Required:** No
- **Minimum:** >= 0

## A.23.3. `propertyTable.property.stringOffsets`

The index of the buffer view containing offsets for strings. The number of offsets is equal to the number of string elements plus one. The offsets represent the byte offsets of each string in the property array (stored in `values`), with the last offset representing the byte offset after the last string. The string byte length is computed using the difference between the subsequent offset and the current offset. The data type of these offsets is determined by `stringOffsetType`. The buffer view `byteOffset` shall be aligned to a multiple of the `stringOffsetType` size.

- **Type:** integer
- **Required:** No
- **Minimum:** >= 0

## A.23.4. `propertyTable.property.arrayOffsetType`

The type of values in `arrayOffsets`.

- **Type:** string
- **Required:** No, default: `UINT32`

- **Allowed values:**

- "UINT8"
- "UINT16"
- "UINT32"
- "UINT64"

### A.23.5. `propertyTable.property.stringOffsetType`

The type of values in `stringOffsets`.

- **Type:** string

- **Required:** No, default: UINT32

- **Allowed values:**

- "UINT8"
- "UINT16"
- "UINT32"
- "UINT64"

### A.23.6. `propertyTable.property.offset`

An offset to apply to property values. Only applicable when the component type is FLOAT32 or FLOAT64, or when the property is normalized. Overrides the class property's offset if both are defined.

- **Type:** definitions-definitions-numericValue

- **Required:** No

### A.23.7. `propertyTable.property.scale`

A scale to apply to property values. Only applicable when the component type is FLOAT32 or FLOAT64, or when the property is normalized. Overrides the class property's scale if both are defined.

- **Type:** definitions-definitions-numericValue

- **Required:** No

## A.23.8. propertyTable.property.max

Maximum value present in the property values. Only applicable to SCALAR, VECN, and MATN types. This is the maximum of all property values, after the transforms based on the normalized, offset, and scale properties have been applied.

- **Type:** definitions-definitions-numericValue
- **Required:** No

## A.23.9. propertyTable.property.min

Minimum value present in the property values. Only applicable to SCALAR, VECN, and MATN types. This is the minimum of all property values, after the transforms based on the normalized, offset, and scale properties have been applied.

- **Type:** definitions-definitions-numericValue
- **Required:** No

## A.23.10. propertyTable.property.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

## A.23.11. propertyTable.property.extras

Application-specific data.

- **Type:** extras
- **Required:** No

## A.24. Root Property

---

A basis for storing extensions and extras.

**Table A.24 — Root Property Properties**

	TYPE	DESCRIPTION	REQUIRED
extensions	extension	Dictionary object with extension-specific objects.	No
extras	extras	Application-specific data.	No

- Additional properties are allowed.
- JSON schema: `rootProperty.schema.json`

### A.24.1. rootProperty.extensions

Dictionary object with extension-specific objects.

- Type: extension
- Required: No
- Additional properties are allowed.
- Type of each property: object

### A.24.2. rootProperty.extras

Application-specific data.

- Type: extras
- Required: No

## A.25. Schema

---

An object defining classes and enums.

**Table A.25 – Schema Properties**

TYPE	DESCRIPTION	REQUIRED
<b>id</b>	string Unique identifier for the schema. Schema IDs shall be alphanumeric identifiers matching the regular expression <code>^[a-zA-Z_][a-zA-Z0-9_]*\$</code> .	✓ Yes
<b>name</b>	string The name of the schema, e.g. for display purposes.	No
<b>description</b>	string The description of the schema.	No
<b>version</b>	string Application-specific version of the schema.	No
<b>classes</b>	object A dictionary, where each key is a class ID and each value is an object defining the class. Class IDs shall be alphanumeric identifiers matching the regular expression <code>^[a-zA-Z_][a-zA-Z0-9_]*\$</code> .	No
<b>enums</b>	object A dictionary, where each key is an enum ID and each value is an object defining the values for the enum. Enum IDs shall be alphanumeric identifiers matching the regular expression <code>^[a-zA-Z_][a-zA-Z0-9_]*\$</code> .	No
<b>extensions</b>	extension Dictionary object with extension-specific objects.	No
<b>extras</b>	extras Application-specific data.	No

- Additional properties are allowed.
- **JSON schema:** `schema.schema.json`

## A.25.1. schema.id

Unique identifier for the schema. Schema IDs shall be alphanumeric identifiers matching the regular expression `^[a-zA-Z_][a-zA-Z0-9_]*$`.

- **Type:** string
- **Required:** ✓ Yes
- **Pattern:** `^[a-zA-Z_][a-zA-Z0-9_]*$`

## A.25.2. schema.name

The name of the schema, e.g. for display purposes.

- **Type:** string

- **Required:** No
- **Minimum Length:** >= 1

### A.25.3. schema.description

The description of the schema.

- **Type:** string
- **Required:** No
- **Minimum Length:** >= 1

### A.25.4. schema.version

Application-specific version of the schema.

- **Type:** string
- **Required:** No
- **Minimum Length:** >= 1

### A.25.5. schema.classes

A dictionary, where each key is a class ID and each value is an object defining the class. Class IDs shall be alphanumeric identifiers matching the regular expression `^[a-zA-Z_][a-zA-Z0-9_]*$`.

- **Type:** object
- **Required:** No
- **Minimum number of properties:** 1
- **Additional properties are allowed.**
- **Type of each property:** class

### A.25.6. schema.enums

A dictionary, where each key is an enum ID and each value is an object defining the values for the enum. Enum IDs shall be alphanumeric identifiers matching the regular expression `^[a-zA-Z_][a-zA-Z0-9_]*$`.

- **Type:** object
- **Required:** No
- **Minimum number of properties:** 1
- **Additional properties are allowed.**
- **Type of each property:** enum

## A.25.7. schema.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

## A.25.8. schema.extras

Application-specific data.

- **Type:** extras
- **Required:** No

## A.26. Statistics

---

Statistics about entities.

**Table A.26 – Statistics Properties**

TYPE	DESCRIPTION	REQUIRED
classes	A dictionary, where each key corresponds to a class ID in the classes dictionary and each value is an object containing statistics about entities that conform to the class.	No
extensions	extension Dictionary object with extension-specific objects.	No

TYPE	DESCRIPTION	REQUIRED
extras	extras Application-specific data.	No

- Additional properties are allowed.
- JSON schema: `statistics.schema.json`

## A.26.1. statistics.classes

A dictionary, where each key corresponds to a class ID in the `classes` dictionary and each value is an object containing statistics about entities that conform to the class.

- Type: object
- Required: No
- Minimum number of properties: 1
- Additional properties are allowed.
- Type of each property: `statistics.class`

## A.26.2. statistics.extensions

Dictionary object with extension-specific objects.

- Type: extension
- Required: No
- Additional properties are allowed.
- Type of each property: object

## A.26.3. statistics.extras

Application-specific data.

- Type: extras
- Required: No

## A.27. Class Statistics

---

Statistics about entities that conform to a class.

**Table A.27 – Class Statistics Properties**

Type	Description	Required	
count	integer	The number of entities that conform to the class.	No
properties	object	A dictionary, where each key corresponds to a property ID in the class' properties dictionary and each value is an object containing statistics about property values.	No
extensions	extension	Dictionary object with extension-specific objects.	No
extras	extras	Application-specific data.	No

- Additional properties are allowed.
- JSON schema: `statistics.class.schema.json`

### A.27.1. `statistics.class.count`

The number of entities that conform to the class.

- Type: integer
- Required: No
- Minimum:  $\geq 0$

### A.27.2. `statistics.class.properties`

A dictionary, where each key corresponds to a property ID in the class' properties dictionary and each value is an object containing statistics about property values.

- Type: object
- Required: No
- Minimum number of properties: 1
- Additional properties are allowed.

- **Type of each property:** statistics.class.property

### A.27.3. statistics.class.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

### A.27.4. statistics.class.extras

Application-specific data.

- **Type:** extras
- **Required:** No

## A.28. Property Statistics

---

Statistics about property values.

**Table A.28 — Property Statistics Properties**

	TYPE	DESCRIPTION	REQUIRED
min	definitions-definitions-numericValue	The minimum property value occurring in the tileset. Only applicable to SCALAR, VECN, and MATN types. This is the minimum of all property values, after the transforms based on the normalized, offset, and scale properties have been applied.	No
max	definitions-definitions-numericValue	The maximum property value occurring in the tileset. Only applicable to SCALAR, VECN, and MATN types. This is the maximum of all property values, after the transforms based on the normalized, offset, and scale properties have been applied.	No

	TYPE	DESCRIPTION	REQUIRED
mean	definitions-definitions-numericValue	The arithmetic mean of property values occurring in the tileset. Only applicable to SCALAR, VECN, and MATN types. This is the mean of all property values, after the transforms based on the normalized, offset, and scale properties have been applied.	No
median	definitions-definitions-numericValue	The median of property values occurring in the tileset. Only applicable to SCALAR, VECN, and MATN types. This is the median of all property values, after the transforms based on the normalized, offset, and scale properties have been applied.	No
standard Deviation	definitions-definitions-numericValue	The standard deviation of property values occurring in the tileset. Only applicable to SCALAR, VECN, and MATN types. This is the standard deviation of all property values, after the transforms based on the normalized, offset, and scale properties have been applied.	No
variance	definitions-definitions-numericValue	The variance of property values occurring in the tileset. Only applicable to SCALAR, VECN, and MATN types. This is the variance of all property values, after the transforms based on the normalized, offset, and scale properties have been applied.	No
sum	definitions-definitions-numericValue	The sum of property values occurring in the tileset. Only applicable to SCALAR, VECN, and MATN types. This is the sum of all property values, after the transforms based on the normalized, offset, and scale properties have been applied.	No
occurrences	object	A dictionary, where each key corresponds to an enum name and each value is the number of occurrences of that enum. Only applicable when type is ENUM. For fixed-length arrays, this is an array of component-wise occurrences.	No
extensions	extension	Dictionary object with extension-specific objects.	No
extras	extras	Application-specific data.	No

- Additional properties are allowed.
- JSON schema: `statistics.class.property.schema.json`

## A.28.1. statistics.class.property.min

The minimum property value occurring in the tileset. Only applicable to SCALAR, VECN, and MATN types. This is the minimum of all property values, after the transforms based on the normalized, offset, and scale properties have been applied.

- **Type:** definitions-definitions-numericValue
- **Required:** No

## A.28.2. statistics.class.property.max

The maximum property value occurring in the tileset. Only applicable to SCALAR, VECN, and MATN types. This is the maximum of all property values, after the transforms based on the normalized, offset, and scale properties have been applied.

- **Type:** definitions-definitions-numericValue
- **Required:** No

## A.28.3. statistics.class.property.mean

The arithmetic mean of property values occurring in the tileset. Only applicable to SCALAR, VECN, and MATN types. This is the mean of all property values, after the transforms based on the normalized, offset, and scale properties have been applied.

- **Type:** definitions-definitions-numericValue
- **Required:** No

## A.28.4. statistics.class.property.median

The median of property values occurring in the tileset. Only applicable to SCALAR, VECN, and MATN types. This is the median of all property values, after the transforms based on the normalized, offset, and scale properties have been applied.

- **Type:** definitions-definitions-numericValue
- **Required:** No

## A.28.5. statistics.class.property.standardDeviation

The standard deviation of property values occurring in the tileset. Only applicable to SCALAR, VECN, and MATN types. This is the standard deviation of all property values, after the transforms based on the normalized, offset, and scale properties have been applied.

- **Type:** definitions-definitions-numericValue
- **Required:** No

## A.28.6. statistics.class.property.variance

The variance of property values occurring in the tileset. Only applicable to SCALAR, VECN, and MATN types. This is the variance of all property values, after the transforms based on the normalized, offset, and scale properties have been applied.

- **Type:** definitions-definitions-numericValue
- **Required:** No

## A.28.7. statistics.class.property.sum

The sum of property values occurring in the tileset. Only applicable to SCALAR, VECN, and MATN types. This is the sum of all property values, after the transforms based on the normalized, offset, and scale properties have been applied.

- **Type:** definitions-definitions-numericValue
- **Required:** No

## A.28.8. statistics.class.property.occurrences

A dictionary, where each key corresponds to an enum name and each value is the number of occurrences of that enum. Only applicable when type is ENUM. For fixed-length arrays, this is an array of component-wise occurrences.

- **Type:** object
- **Required:** No
- **Minimum number of properties:** 1
- **Additional properties are allowed.**

- Type of each property: integer

## A.28.9. statistics.class.property.extensions

Dictionary object with extension-specific objects.

- Type: extension
- Required: No
- Additional properties are allowed.
- Type of each property: object

## A.28.10. statistics.class.property.extras

Application-specific data.

- Type: extras
- Required: No

# A.29. Style

---

A 3D Tiles style.

**Table A.29 – Style Properties**

Type	Description	Required
defines	object	A dictionary object of expression strings mapped to a variable name key that may be referenced throughout the style. If an expression references a defined variable, it is replaced with the evaluated result of the corresponding expression.
show	One of style.booleanExpression, style.conditions	A boolean expression or conditions property which determines if a feature should be shown.
color	One of style.colorExpression, style.conditions	A color expression or conditions property which determines the color blended with the feature's intrinsic color.

Type	Description	Required
meta	style.meta	A meta object which determines the values of non-visual properties of the feature.
extensions	extension	Dictionary object with extension-specific objects.
extras	extras	Application-specific data.

- Additional properties are allowed.
- JSON schema: style.schema.json

## A.29.1. style.defines

A dictionary object of expression strings mapped to a variable name key that may be referenced throughout the style. If an expression references a defined variable, it is replaced with the evaluated result of the corresponding expression.

- Type: object
- Required: No
- Additional properties are allowed.
- Type of each property: style.expression

## A.29.2. style.show

A boolean expression or conditions property which determines if a feature should be shown.

- Type: One of style.booleanExpression, style.conditions
- Required: No, default: true

## A.29.3. style.color

A color expression or conditions property which determines the color blended with the feature's intrinsic color.

- Type: One of style.colorExpression, style.conditions
- Required: No, default: color( '#FFFFFF' )

## A.29.4. style.meta

A meta object which determines the values of non-visual properties of the feature.

- **Type:** style.meta
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** style.expression

## A.29.5. style.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

## A.29.6. style.extras

Application-specific data.

- **Type:** extras
- **Required:** No

## A.30. Boolean Expression

---

A boolean or string with a 3D Tiles style expression that evaluates to a boolean. Details are described in the 3D Tiles Styling specification.

- **Additional properties are allowed.**
- **JSON schema:** style.booleanExpression.schema.json

## A.31. Color Expression

---

3D Tiles style expression that evaluates to a Color. Details are described in the 3D Tiles Styling specification.

- Additional properties are allowed.
- **JSON schema:** style.colorExpression.schema.json

## A.32. Conditions

---

A series of conditions evaluated in order, like a series of if...else statements that result in an expression being evaluated.

**Table A.30 — Conditions Properties**

Type	Description	Required
style.conditions	A series of boolean conditions evaluated in order. For the first one that evaluates to true, its value, the 'result' (which is also an expression), is evaluated and returned. Result expressions shall all be the same type. If no condition evaluates to true, the result is undefined. When conditions is undefined, null, or an empty object, the result is undefined.	No
extensions.extension	Dictionary object with extension-specific objects.	No
extras.extras	Application-specific data.	No

- Additional properties are allowed.
- **JSON schema:** style.conditions.schema.json

### A.32.1. style.conditions.conditions

A series of boolean conditions evaluated in order. For the first one that evaluates to true, its value, the 'result' (which is also an expression), is evaluated and returned. Result expressions shall all be the same type. If no condition evaluates to true, the result is undefined. When conditions is undefined, null, or an empty object, the result is undefined.

- **Type:** style.conditions.condition []

- **Required:** No

### A.32.2. style.conditions.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

### A.32.3. style.conditions.extras

Application-specific data.

- **Type:** extras
- **Required:** No

## A.33. Condition

---

An expression evaluated as the result of a condition being true. An array of two expressions. If the first expression is evaluated and the result is true, then the second expression is evaluated and returned as the result of the condition.

- **Additional properties are allowed.**
- **JSON schema:** style.conditions.condition.schema.json

## A.34. Expression

---

A valid 3D Tiles style expression. Details are described in the 3D Tiles Styling specification.

- **Additional properties are allowed.**
- **JSON schema:** style.expression.schema.json

## A.35. Meta

---

A series of property names and the expression to evaluate for the value of that property.

**Table A.31 — Meta Properties**

	TYPE	DESCRIPTION	REQUIRED
extensions	extension	Dictionary object with extension-specific objects.	No
extras	extras	Application-specific data.	No

- Additional properties are allowed.
- Type of additional properties: style.expression
- JSON schema: style.meta.schema.json

### A.35.1. style.meta.extensions

Dictionary object with extension-specific objects.

- Type: extension
- Required: No
- Additional properties are allowed.
- Type of each property: object

### A.35.2. style.meta.extras

Application-specific data.

- Type: extras
- Required: No

## A.36. Subtree

An object describing the availability of tiles and content in a subtree, as well as availability of children subtrees. May also store metadata for available tiles and content.

**Table A.32 – Subtree Properties**

	TYPE	DESCRIPTION	REQUIRED
buffers	buffer [1-*]	An array of buffers.	No
bufferViews	bufferView [1-*]	An array of buffer views.	No
propertyTables	propertyTable [1-*]	An array of property tables.	No
tileAvailability	availability	The availability of tiles in the subtree. The availability bitstream is a 1D boolean array where tiles are ordered by their level in the subtree and Morton index within that level. A tile's availability is determined by a single bit, 1 meaning a tile exists at that spatial index, and 0 meaning it does not. The number of elements in the array is $(N^{\text{subtreeLevels}} - 1)/(N - 1)$ where N is 4 for subdivision scheme QUADTREE and 8 for OCTREE. Availability may be stored in a buffer view or as a constant value that applies to all tiles. If a non-root tile's availability is 1 its parent tile's availability shall also be 1. <code>tileAvailability.constant: 0</code> is disallowed, as subtrees shall have at least one tile.	✓ Yes
contentAvailability	availability [1-*]	An array of content availability objects. If the tile has a single content this array will have one element; if the tile has multiple contents — as supported by 3DTILES_multiple_contents and 3D Tiles 1.1 — this array will have multiple elements.	No
childSubtree Availability	availability	The availability of children subtrees. The availability bitstream is a 1D boolean array where subtrees are ordered by their Morton index in the level of the tree immediately below the bottom row of the subtree. A child subtree's availability is determined by a single bit, 1 meaning a subtree exists at that spatial index, and 0 meaning it does not. The number of elements in the array is $N^{\text{subtreeLevels}}$ where N is 4 for subdivision scheme QUADTREE and 8 for OCTREE. Availability may be stored in a buffer view or as a constant value that applies to all child subtrees. If	✓ Yes

Type	Description	Required	
	availability is 0 for all child subtrees, then the tileset does not subdivide further.		
tileMetadata	integer	Index of the property table containing tile metadata. Tile metadata only exists for available tiles and is tightly packed by increasing tile index. To access individual tile metadata, implementations may create a mapping from tile indices to tile metadata indices.	No
contentMetadata	integer [1-*]	An array of indexes to property tables containing content metadata. If the tile has a single content this array will have one element; if the tile has multiple contents — as supported by 3DTILES_multiple_contents and 3D Tiles 1.1 — this array will have multiple elements. Content metadata only exists for available contents and is tightly packed by increasing tile index. To access individual content metadata, implementations may create a mapping from tile indices to content metadata indices.	No
subtreeMetadata	metadataEntity	Subtree metadata encoded in JSON.	No
extensions	extension	Dictionary object with extension-specific objects.	No
extras	extras	Application-specific data.	No

- Additional properties are allowed.
- JSON schema: `subtree.schema.json`

### A.36.1. subtree.buffers

An array of buffers.

- Type: `buffer [1-*]`
- Required: No

### A.36.2. subtree.bufferViews

An array of buffer views.

- Type: `bufferView [1-*]`
- Required: No

### A.36.3. subtree.propertyTables

An array of property tables.

- **Type:** propertyTable [1-\*]
- **Required:** No

### A.36.4. subtree.tileAvailability

The availability of tiles in the subtree. The availability bitstream is a 1D boolean array where tiles are ordered by their level in the subtree and Morton index within that level. A tile's availability is determined by a single bit, 1 meaning a tile exists at that spatial index, and 0 meaning it does not. The number of elements in the array is  $(N^{\text{subtreeLevels}} - 1)/(N - 1)$  where N is 4 for subdivision scheme QUADTREE and 8 for OCTREE. Availability may be stored in a buffer view or as a constant value that applies to all tiles. If a non-root tile's availability is 1 its parent tile's availability shall also be 1. `tileAvailability.constant: 0` is disallowed, as subtrees shall have at least one tile.

- **Type:** availability
- **Required:** ✓ Yes

### A.36.5. subtree.contentAvailability

An array of content availability objects. If the tile has a single content this array will have one element; if the tile has multiple contents — as supported by `3DTILES_multiple_contents` and `3D Tiles 1.1` — this array will have multiple elements.

- **Type:** availability [1-\*]
- **Required:** No

### A.36.6. subtree.childSubtreeAvailability

The availability of children subtrees. The availability bitstream is a 1D boolean array where subtrees are ordered by their Morton index in the level of the tree immediately below the bottom row of the subtree. A child subtree's availability is determined by a single bit, 1 meaning a subtree exists at that spatial index, and 0 meaning it does not. The number of elements in the array is  $N^{\text{subtreeLevels}}$  where N is 4 for subdivision scheme QUADTREE and 8 for OCTREE. Availability may be stored in a buffer view or as a constant value that applies to all child subtrees. If availability is 0 for all child subtrees, then the tiles set does not subdivide further.

- **Type:** availability
- **Required:** ✓ Yes

### A.36.7. subtree.tileMetadata

Index of the property table containing tile metadata. Tile metadata only exists for available tiles and is tightly packed by increasing tile index. To access individual tile metadata, implementations may create a mapping from tile indices to tile metadata indices.

- **Type:** integer
- **Required:** No
- **Minimum:** >= 0

### A.36.8. subtree.contentMetadata

An array of indexes to property tables containing content metadata. If the tile has a single content this array will have one element; if the tile has multiple contents – as supported by 3DTILES\_multiple\_contents and 3D Tiles 1.1 – this array will have multiple elements. Content metadata only exists for available contents and is tightly packed by increasing tile index. To access individual content metadata, implementations may create a mapping from tile indices to content metadata indices.

- **Type:** integer [1-\*]
  - Each element in the array shall be greater than or equal to 0.
- **Required:** No

### A.36.9. subtree.subtreeMetadata

Subtree metadata encoded in JSON.

- **Type:** metadataEntity
- **Required:** No

### A.36.10. subtree.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

## A.36.11. subtree.extras

Application-specific data.

- **Type:** extras
- **Required:** No

## A.37. Subtrees

---

An object describing the location of subtree files.

**Table A.33 — Subtrees Properties**

TYPE	DESCRIPTION	REQUIRED	
uri	templateUri	A template URI pointing to subtree files. A subtree is a fixed-depth (defined by subtreeLevels) portion of the tree to keep memory use bounded. The URI of each file is substituted with the subtree root's global level, x, and y. For subdivision scheme OCTREE, z shall also be given. Relative paths are relative to the tileset JSON.	✓ Yes
extensions	extension	Dictionary object with extension-specific objects.	No
extras	extras	Application-specific data.	No

- **Additional properties are allowed.**
- **JSON schema:** subtrees.schema.json

### A.37.1. subtrees.uri

A template URI pointing to subtree files. A subtree is a fixed-depth (defined by subtreeLevels) portion of the tree to keep memory use bounded. The URI of each file is substituted with the

subtree root's global level, x, and y. For subdivision scheme OCTREE, z shall also be given. Relative paths are relative to the tileset JSON.

- **Type:** templateUri
- **Required:** ✓ Yes

### A.37.2. subtrees.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

### A.37.3. subtrees.extras

Application-specific data.

- **Type:** extras
- **Required:** No

## A.38. Template URI

---

A URI with embedded expressions that describes the resource that is associated with an implicit tile in an implicit tileset. Allowed expressions are {level}, {x}, {y}, and {z}. {level} is substituted with the level of the node, {x} is substituted with the x index of the node within the level, and {y} is substituted with the y index of the node within the level. {z} may only be given when the subdivision scheme is OCTREE, and it is substituted with the z index of the node within the level.

- **Additional properties are allowed.**
- **JSON schema:** templateUri.schema.json

## A.39. Tile

---

A tile in a 3D Tiles tileset.

**Table A.34 – Tile Properties**

	TYPE	DESCRIPTION	REQUIRED
<b>boundingVolume</b>	boundingVolume	The bounding volume that encloses the tile.	✓ Yes
<b>viewerRequestVolume</b>	boundingVolume	Optional bounding volume that defines the volume the viewer shall be inside of before the tile's content will be requested and before the tile will be refined based on <code>geometricError</code> .	No
<b>geometricError</b>	number	The error, in meters, introduced if this tile is rendered and its children are not. At runtime, the geometric error is used to compute screen space error (SSE), i.e., the error measured in pixels.	✓ Yes
<b>refine</b>	string	Specifies if additive or replacement refinement is used when traversing the tileset for rendering. This property is required for the root tile of a tileset; it is optional for all other tiles. The default is to inherit	No

Type	Description	Required
<code>transform</code>	<p>from the parent tile.</p> <p>A floating-point <math>4 \times 4</math> affine transformation matrix, stored in column-major order, that transforms the tile's content—i.e., its features as well as content. <code>boundingVolume</code>, <code>boundingVolume</code>, and <code>viewerRequestVolume</code>—from the tile's local coordinate system to the parent tile's coordinate system, or, in the case of a root tile, from the tile's local coordinate system to the tilesheet's coordinate system. <code>transform</code> does not apply to any volume property when the volume is a region, defined in EPSG:4979 coordinates. <code>transform</code> scales the <code>geometricError</code> by the maximum scaling factor from the matrix.</p>	No, default: <code>[1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1]</code>
<code>content</code>	<p>Metadata about the tile's content and a link to the content. When this is omitted the tile is just used for culling. When this</p>	No

	TYPE	DESCRIPTION	REQUIRED
		is defined, then contents shall be undefined.	
contents	content [1-*]	An array of contents. When this is defined, then content shall be undefined.	No
metadata	metadataEntity	A metadata entity that is associated with this tile.	No
implicitTiling	tile.implicitTiling	An object that describes the implicit subdivision of this tile.	No
children	tile [1-*]	An array of objects that define child tiles. Each child tile content is fully enclosed by its parent tile's bounding volume and, generally, has a geometricError less than its parent tile's geometric Error. For leaf tiles, the length of this array is zero, and children may not be defined.	No
extensions	extension	Dictionary object with extension-specific objects.	No
extras	extras	Application-specific data.	No

- Additional properties are allowed.
- **JSON schema:** tile.schema.json

### A.39.1. tile.boundingVolume

The bounding volume that encloses the tile.

- **Type:** boundingVolume
- **Required:** ✓ Yes
- **Minimum number of properties:** 1

### A.39.2. tile.viewerRequestVolume

Optional bounding volume that defines the volume the viewer shall be inside of before the tile's content will be requested and before the tile will be refined based on geometricError.

- **Type:** boundingVolume
- **Required:** No
- **Minimum number of properties:** 1

### A.39.3. tile.geometricError

The error, in meters, introduced if this tile is rendered and its children are not. At runtime, the geometric error is used to compute screen space error (SSE), i.e., the error measured in pixels.

- **Type:** number
- **Required:** ✓ Yes
- **Minimum:** >= 0

### A.39.4. tile.refine

Specifies if additive or replacement refinement is used when traversing the tileset for rendering. This property is required for the root tile of a tileset; it is optional for all other tiles. The default is to inherit from the parent tile.

- **Type:** string
- **Required:** No

- **Allowed values:**

- "ADD"
- "REPLACE"

## A.39.5. tile.transform

A floating-point  $4 \times 4$  affine transformation matrix, stored in column-major order, that transforms the tile's content—i.e., its features as well as content.boundingVolume, boundingVolume, and viewerRequestVolume—from the tile's local coordinate system to the parent tile's coordinate system, or, in the case of a root tile, from the tile's local coordinate system to the tileset's coordinate system. transform does not apply to any volume property when the volume is a region, defined in EPSG:4979 coordinates. transform scales the geometricError by the maximum scaling factor from the matrix.

- **Type:** number [16]
- **Required:** No, default: [1,0,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1]

## A.39.6. tile.content

Metadata about the tile's content and a link to the content. When this is omitted the tile is just used for culling. When this is defined, then contents shall be undefined.

- **Type:** content
- **Required:** No

## A.39.7. tile.contents

An array of contents. When this is defined, then content shall be undefined.

- **Type:** content [1-\*]
- **Required:** No

## A.39.8. tile.metadata

A metadata entity that is associated with this tile.

- **Type:** metadataEntity
- **Required:** No

### A.39.9. tile.implicitTiling

An object that describes the implicit subdivision of this tile.

- **Type:** tile.implicitTiling
- **Required:** No

### A.39.10. tile.children

An array of objects that define child tiles. Each child tile content is fully enclosed by its parent tile's bounding volume and, generally, has a geometricError less than its parent tile's geometricError. For leaf tiles, the length of this array is zero, and children may not be defined.

- **Type:** tile [1-\*]
  - Each element in the array shall be unique.
- **Required:** No

### A.39.11. tile.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

### A.39.12. tile.extras

Application-specific data.

- **Type:** extras
- **Required:** No

## A.40. Implicit tiling

---

This object allows a tile to be implicitly subdivided. Tile and content availability and metadata is stored in subtrees which are referenced externally.

**Table A.35 – Implicit tiling Properties**

	TYPE	DESCRIPTION	REQUIRED
subdivisionScheme	string	A string describing the subdivision scheme used within the tileset.	✓ Yes
subtreeLevels	integer	The number of distinct levels in each subtree. For example, a quadtree with subtreeLevels = 2 will have subtrees with 5 nodes (one root and 4 children).	✓ Yes
availableLevels	integer	The numbers of the levels in the tree with available tiles.	✓ Yes
subtrees	subtrees	An object describing the location of subtree files.	✓ Yes
extensions	extension	Dictionary object with extension-specific objects.	No
extras	extras	Application-specific data.	No

- Additional properties are allowed.
- JSON schema: tile.implicitTiling.schema.json

### A.40.1. tile.implicitTiling.subdivisionScheme

A string describing the subdivision scheme used within the tileset.

- Type: string
- Required: ✓ Yes
- Allowed values:
  - "QUADTREE"
  - "OCTREE"

## A.40.2. tile.implicitTiling.subtreeLevels

The number of distinct levels in each subtree. For example, a quadtree with subtreeLevels = 2 will have subtrees with 5 nodes (one root and 4 children).

- **Type:** integer
- **Required:** ✓ Yes
- **Minimum:** >= 1

## A.40.3. tile.implicitTiling.availableLevels

The numbers of the levels in the tree with available tiles.

- **Type:** integer
- **Required:** ✓ Yes
- **Minimum:** >= 1

## A.40.4. tile.implicitTiling.subtrees

An object describing the location of subtree files.

- **Type:** subtrees
- **Required:** ✓ Yes

## A.40.5. tile.implicitTiling.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- **Additional properties are allowed.**
- **Type of each property:** object

## A.40.6. tile.implicitTiling.extras

Application-specific data.

- **Type:** extras
- **Required:** No

## A.41. Tileset

A 3D Tiles tileset.

**Table A.36 – Tileset Properties**

	TYPE	DESCRIPTION	REQUIRED
asset	asset	Metadata about the entire tileset.	✓ Yes
properties	object	A dictionary object of metadata about per-feature properties.	No
schema	schema	An object defining the structure of metadata classes and enums. When this is defined, then schemaUri shall be undefined.	No
schemaUri	string	The URI (or IRI) of the external schema file. When this is defined, then schema shall be undefined.	No
statistics	statistics	An object containing statistics about metadata entities.	No
groups	group [1-*]	An array of groups that tile content may belong to. Each element of this array is a metadata entity that describes the group. The tile content group property is an index into this array.	No
metadata	metadataEntity	A metadata entity that is associated with this tileset.	No
geometricError	number	The error, in meters, introduced if this tileset is not rendered. At runtime, the geometric error is used to compute screen space error (SSE), i.e., the error measured in pixels.	✓ Yes
root	tile	The root tile.	✓ Yes

	TYPE	DESCRIPTION	REQUIRED
extensionsUsed	string [1-*]	Names of 3D Tiles extensions used somewhere in this tileset.	No
extensions Required	string [1-*]	Names of 3D Tiles extensions required to properly load this tileset.	No
extensions	extension	Dictionary object with extension-specific objects.	No
extras	extras	Application-specific data.	No

- Additional properties are allowed.
- JSON schema: tileset.schema.json

### A.41.1. tileset.asset

Metadata about the entire tileset.

- Type: asset
- Required: ✓ Yes

### A.41.2. tileset.properties

A dictionary object of metadata about per-feature properties.

- Type: object
- Required: No
- Additional properties are allowed.
- Type of each property: properties

### A.41.3. tileset.schema

An object defining the structure of metadata classes and enums. When this is defined, then schemaUri shall be undefined.

- Type: schema
- Required: No

#### A.41.4. tileset.schemaUri

The URI (or IRI) of the external schema file. When this is defined, then schema shall be undefined.

- **Type:** string
- **Required:** No
- **Format:** iri-reference

#### A.41.5. tileset.statistics

An object containing statistics about metadata entities.

- **Type:** statistics
- **Required:** No

#### A.41.6. tileset.groups

An array of groups that tile content may belong to. Each element of this array is a metadata entity that describes the group. The tile content group property is an index into this array.

- **Type:** group [1-\*]
- **Required:** No

#### A.41.7. tileset.metadata

A metadata entity that is associated with this tileset.

- **Type:** metadataEntity
- **Required:** No

#### A.41.8. tileset.geometricError

The error, in meters, introduced if this tileset is not rendered. At runtime, the geometric error is used to compute screen space error (SSE), i.e., the error measured in pixels.

- **Type:** number

- **Required:** ✓ Yes
- **Minimum:** >= 0

### A.41.9. tileset.root

The root tile.

- **Type:** tile
- **Required:** ✓ Yes

### A.41.10. tileset.extensionsUsed

Names of 3D Tiles extensions used somewhere in this tileset.

- **Type:** string [1-\*]
  - Each element in the array shall be unique.
- **Required:** No

### A.41.11. tileset.extensionsRequired

Names of 3D Tiles extensions required to properly load this tileset.

- **Type:** string [1-\*]
  - Each element in the array shall be unique.
- **Required:** No

### A.41.12. tileset.extensions

Dictionary object with extension-specific objects.

- **Type:** extension
- **Required:** No
- Additional properties are allowed.
- **Type of each property:** object

### A.41.13. tileset.extras

Application-specific data.

- **Type:** extras
- **Required:** No



B

# ANNEX B (INFORMATIVE) JSON SCHEMA REFERENCE

---

## ANNEX B (INFORMATIVE) JSON SCHEMA REFERENCE

---

### B.1. JSON Schema for Asset

---

```
{  
    "$schema": "https://json-schema.org/draft/2020-12/schema",  
    "$id": "asset.schema.json",  
    "title": "Asset",  
    "$ref": "rootProperty.schema.json",  
    "description": "Metadata about the entire tileset.",  
    "properties": {  
        "version": {  
            "type": "string",  
            "description": "The 3D Tiles version. The version defines the JSON schema for the tileset JSON and the base set of tile formats."  
        },  
        "tilesetVersion": {  
            "type": "string",  
            "description": "Application-specific version of this tileset, e.g., for when an existing tileset is updated."  
        }  
    },  
    "required": [  
        "version"  
    ]  
}
```

### B.2. JSON Schema for Availability

---

```
{  
    "$schema": "https://json-schema.org/draft/2020-12/schema",  
    "$id": "availability.schema.json",  
    "title": "Availability",  
    "$ref": "rootProperty.schema.json",  
    "description": "An object describing the availability of a set of elements.",  
    "properties": {  
        "bitstream": {  
            "description": "Index of a buffer view that indicates whether each element is available. The bitstream conforms to the boolean array encoding"  
        }  
    }  
}
```

```

described in the 3D Metadata specification. If an element is available, its
bit is 1, and if it is unavailable, its bit is 0.",
    "type": "integer",
    "minimum": 0
},
"availableCount": {
    "description": "A number indicating how many 1 bits exist in the
availability bitstream.",
    "type": "integer",
    "minimum": 0
},
"constant": {
    "description": "Integer indicating whether all of the elements are
available (1) or all are unavailable (0).",
    "anyOf": [
        {
            "const": 0,
            "description": "UNAVAILABLE",
            "type": "integer"
        },
        {
            "const": 1,
            "description": "AVAILABLE",
            "type": "integer"
        },
        {
            "type": "integer"
        }
    ]
},
"oneOf": [
    {
        "required": [
            "bitstream"
        ]
    },
    {
        "required": [
            "constant"
        ]
    }
]
}

```

## B.3. JSON Schema for Batched 3D Model Feature Table

---

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "b3dm.featureTable.schema.json",
    "title": "Batched 3D Model Feature Table",
    "$ref": "featureTable.schema.json",
    "description": "A set of Batched 3D Model semantics that contain
additional information about features in a tile.",
    "deprecated": true,
    "properties": {
        "BATCH_LENGTH": {

```

```

        "$ref": "featureTable.schema.json#/definitions/globalPropertyInteger",
      },
      "description": "A `GlobalPropertyInteger` object defining an integer property for all features. Details about this property are described in the 3D Tiles specification."
    },
    "RTC_CENTER": {
      "$ref": "featureTable.schema.json#/definitions/globalPropertyCartesian3",
      "description": "A `GlobalPropertyCartesian3` object defining a 3-component numeric property for all features. Details about this property are described in the 3D Tiles specification."
    }
  },
  "required": [
    "BATCH_LENGTH"
  ]
}

```

## B.4. JSON Schema for Batch Table

---

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "batchTable.schema.json",
  "title": "Batch Table",
  "$ref": "rootProperty.schema.json",
  "description": "A set of properties defining application-specific metadata for features in a tile.",
  "deprecated": true,
  "additionalProperties": {
    "$ref": "#/definitions/property"
  },
  "definitions": {
    "binaryBodyReference": {
      "title": "BinaryBodyReference",
      "$ref": "rootProperty.schema.json",
      "description": "An object defining the reference to a section of the binary body of the batch table where the property values are stored if not defined directly in the JSON.",
      "properties": {
        "byteOffset": {
          "type": "integer",
          "description": "The offset into the buffer in bytes.",
          "minimum": 0
        },
        "componentType": {
          "description": "The datatype of components in the property."
        },
        "anyOf": [
          {
            "const": "BYTE"
          },
          {
            "const": "UNSIGNED_BYTE"
          },
          {
            "const": "SHORT"
          },
          {
            "const": "INT"
          },
          {
            "const": "LONG"
          }
        ]
      }
    }
  }
}
```

```

        {
          "const": "UNSIGNED_SHORT"
        },
        {
          "const": "INT"
        },
        {
          "const": "UNSIGNED_INT"
        },
        {
          "const": "FLOAT"
        },
        {
          "const": "DOUBLE"
        },
        {
          "type": "string"
        }
      ]
    },
    "type": {
      "description": "Specifies if the property is a scalar or vector.",
      "anyOf": [
        {
          "const": "SCALAR"
        },
        {
          "const": "VEC2"
        },
        {
          "const": "VEC3"
        },
        {
          "const": "VEC4"
        },
        {
          "type": "string"
        }
      ]
    }
  },
  "required": [
    "byteOffset",
    "componentType",
    "type"
  ]
},
"property": {
  "title": "Property",
  "description": "A user-defined property which specifies per-feature application-specific metadata in a tile. Values either can be defined directly in the JSON as an array, or can refer to sections in the binary body with a `BinaryBodyReference` object.",
  "oneOf": [
    {
      "$ref": "#/definitions/binaryBodyReference"
    },
    {
      "type": "array"
    }
  ]
}

```

```
    }
}
```

## B.5. JSON Schema for Bounding Volume

---

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "boundingVolume.schema.json",
  "title": "Bounding Volume",
  "$ref": "rootProperty.schema.json",
  "description": "A bounding volume that encloses a tile or its content. At least one bounding volume property is required. Bounding volumes include `box`, `region`, or `sphere`.",
  "minProperties": 1,
  "properties": {
    "box": {
      "type": "array",
      "description": "An array of 12 numbers that define an oriented bounding box. The first three elements define the x, y, and z values for the center of the box. The next three elements (with indices 3, 4, and 5) define the x axis direction and half-length. The next three elements (indices 6, 7, and 8) define the y axis direction and half-length. The last three elements (indices 9, 10, and 11) define the z axis direction and half-length.",
      "items": {
        "type": "number"
      },
      "minItems": 12,
      "maxItems": 12
    },
    "region": {
      "type": "array",
      "description": "An array of six numbers that define a bounding geographic region in EPSG:4979 coordinates with the order [west, south, east, north, minimum height, maximum height]. Longitudes and latitudes are in radians, and heights are in meters above (or below) the WGS84 ellipsoid.",
      "items": {
        "type": "number"
      },
      "minItems": 6,
      "maxItems": 6
    },
    "sphere": {
      "type": "array",
      "description": "An array of four numbers that define a bounding sphere. The first three elements define the x, y, and z values for the center of the sphere. The last element (with index 3) defines the radius in meters.",
      "items": {
        "type": "number"
      },
      "minItems": 4,
      "maxItems": 4
    }
  }
}
```

## B.6. JSON Schema for Buffer

---

```
{  
    "$schema": "https://json-schema.org/draft/2020-12/schema",  
    "$id": "buffer.schema.json",  
    "title": "Buffer",  
    "$ref": "rootProperty.schema.json",  
    "description": "A buffer is a binary blob. It is either the binary chunk  
of the subtree file, or an external buffer referenced by a URI.",  
    "properties": {  
        "uri": {  
            "type": "string",  
            "description": "The URI (or IRI) of the external schema file.  
Relative paths are relative to the file containing the buffer JSON. `uri` is  
required when using the JSON subtree format and not required when using the  
binary subtree format - when omitted the buffer refers to the binary chunk of  
the subtree file. Data URIs are not allowed.",  
            "format": "iri-reference"  
        },  
        "byteLength": {  
            "type": "integer",  
            "minimum": 1,  
            "description": "The length of the buffer in bytes."  
        },  
        "name": {  
            "type": "string",  
            "minLength": 1,  
            "description": "The name of the buffer."  
        }  
    },  
    "required": [  
        "byteLength"  
    ]  
}
```

## B.7. JSON Schema for Buffer View

---

```
{  
    "$schema": "https://json-schema.org/draft/2020-12/schema",  
    "$id": "bufferView.schema.json",  
    "title": "Buffer View",  
    "$ref": "rootProperty.schema.json",  
    "description": "A contiguous subset of a buffer",  
    "properties": {  
        "buffer": {  
            "type": "integer",  
            "minimum": 0,  
            "description": "The index of the buffer."  
        },  
        "byteOffset": {  
            "type": "integer",  
            "minimum": 0,  
            "description": "The offset into the buffer in bytes."  
        },  
        "byteLength": {  
            "type": "integer",  
            "minimum": 0,  
            "description": "The length of the buffer view in bytes."  
        }  
    }  
}
```

```

        "type": "integer",
        "minimum": 1,
        "description": "The total byte length of the buffer view."
    },
    "name": {
        "type": "string",
        "minLength": 1,
        "description": "The name of the `bufferView`."
    }
},
"required": [
    "buffer",
    "byteOffset",
    "byteLength"
]
}

```

## B.8. JSON Schema for Class

---

```

{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "class.schema.json",
    "title": "Class",
    "$ref": "rootProperty.schema.json",
    "description": "A class containing a set of properties.",
    "properties": {
        "name": {
            "type": "string",
            "minLength": 1,
            "description": "The name of the class, e.g. for display purposes."
        },
        "description": {
            "type": "string",
            "minLength": 1,
            "description": "The description of the class."
        },
        "properties": {
            "type": "object",
            "description": "A dictionary, where each key is a property ID and each value is an object defining the property. Property IDs shall be alphanumeric identifiers matching the regular expression `^([a-zA-Z_][a-zA-Z0-9_]*$)`."
            "minProperties": 1,
            "additionalProperties": {
                "$ref": "class.property.schema.json"
            }
        }
    }
}

```

## B.9. JSON Schema for Class Property

---

```
{

```

```

"$schema": "https://json-schema.org/draft/2020-12/schema",
"$id": "class.property.schema.json",
"title": "Class Property",
"$ref": "rootProperty.schema.json",
"description": "A single property of a metadata class.",
"properties": {
    "name": {
        "type": "string",
        "minLength": 1,
        "description": "The name of the property, e.g. for display purposes."
    },
    "description": {
        "type": "string",
        "minLength": 1,
        "description": "The description of the property."
    },
    "type": {
        "description": "The element type.",
        "anyOf": [
            {
                "const": "SCALAR"
            },
            {
                "const": "VEC2"
            },
            {
                "const": "VEC3"
            },
            {
                "const": "VEC4"
            },
            {
                "const": "MAT2"
            },
            {
                "const": "MAT3"
            },
            {
                "const": "MAT4"
            },
            {
                "const": "STRING"
            },
            {
                "const": "BOOLEAN"
            },
            {
                "const": "ENUM"
            },
            {
                "type": "string"
            }
        ]
    },
    "componentType": {
        "description": "The datatype of the element's components. Only applicable to `SCALAR`, `VECN`, and `MATN` types.",
        "anyOf": [
            {
                "const": "INT8"
            },
            {

```

```

        "const": "UINT8"
    },
    {
        "const": "INT16"
    },
    {
        "const": "UINT16"
    },
    {
        "const": "INT32"
    },
    {
        "const": "UINT32"
    },
    {
        "const": "INT64"
    },
    {
        "const": "UINT64"
    },
    {
        "const": "FLOAT32"
    },
    {
        "const": "FLOAT64"
    },
    {
        "type": "string"
    }
]
},
"enumType": {
    "type": "string",
    "description": "Enum ID as declared in the `enums` dictionary.
Required when `type` is `ENUM`."
},
"array": {
    "type": "boolean",
    "default": false,
    "description": "Whether the property is an array. When `count` is defined the property is a fixed-length array. Otherwise the property is a variable-length array."
},
"count": {
    "type": "integer",
    "minimum": 2,
    "description": "The number of array elements. May only be defined when `array` is `true`."
},
"normalized": {
    "type": "boolean",
    "description": "Specifies whether integer values are normalized. Only applicable to `SCALAR`, `VECN`, and `MATN` types with integer component types. For unsigned integer component types, values are normalized between `[0, 1.0]`. For signed integer component types, values are normalized between `[-1.0, 1.0]`. For all other component types, this property shall be false.",
    "default": false
},
"offset": {
    "$ref": "definitions.schema.json#/definitions/numericValue",
    "description": "An offset to apply to property values. Only applicable to `SCALAR`, `VECN`, and `MATN` types when the component type is `FLOAT32` or `FLOAT64`, or when the property is `normalized`."
}

```

```

},
  "scale": {
    "$ref": "definitions.schema.json#/definitions/numericValue",
    "description": "A scale to apply to property values. Only applicable to `SCALAR`, `VECN`, and `MATN` types when the component type is `FLOAT32` or `FLOAT64`, or when the property is `normalized`."
  },
  "max": {
    "$ref": "definitions.schema.json#/definitions/numericValue",
    "description": "Maximum allowed value for the property. Only applicable to `SCALAR`, `VECN`, and `MATN` types. This is the maximum of all property values, after the transforms based on the `normalized`, `offset`, and `scale` properties have been applied."
  },
  "min": {
    "$ref": "definitions.schema.json#/definitions/numericValue",
    "description": "Minimum allowed value for the property. Only applicable to `SCALAR`, `VECN`, and `MATN` types. This is the minimum of all property values, after the transforms based on the `normalized`, `offset`, and `scale` properties have been applied."
  },
  "required": {
    "type": "boolean",
    "description": "If required, the property shall be present in every entity conforming to the class. If not required, individual entities may include `noData` values, or the entire property may be omitted. As a result, `noData` has no effect on a required property. Client implementations may use required properties to make performance optimizations.",
    "default": false
  },
  "noData": {
    "$ref": "definitions.schema.json#/definitions/noDataValue",
    "description": "A `noData` value represents missing data – also known as a sentinel value – wherever it appears. `BOOLEAN` properties may not specify `noData` values. This is given as the plain property value, without the transforms from the `normalized`, `offset`, and `scale` properties. Shall not be defined if `required` is true."
  },
  "default": {
    "$ref": "definitions.schema.json#/definitions/anyValue",
    "description": "A default value to use when encountering a `noData` value or an omitted property. The value is given in its final form, taking the effect of `normalized`, `offset`, and `scale` properties into account. Shall not be defined if `required` is true."
  },
  "semantic": {
    "type": "string",
    "minLength": 1,
    "description": "An identifier that describes how this property should be interpreted. The semantic cannot be used by other properties in the class."
  }
},
  "required": [
    "type"
  ]
}

```

## B.10. JSON Schema for Content

---

```
{  
    "$schema": "https://json-schema.org/draft/2020-12/schema",  
    "$id": "content.schema.json",  
    "title": "Content",  
    "$ref": "rootProperty.schema.json",  
    "description": "Metadata about the tile's content and a link to the  
content.",  
    "properties": {  
        "boundingVolume": {  
            "description": "An optional bounding volume that tightly encloses  
tile content. tile.boundingVolume provides spatial coherence and tile.content.  
boundingVolume enables tight view frustum culling. When this is omitted, tile.  
boundingVolume is used.",  
            "$ref": "boundingVolume.schema.json"  
        },  
        "uri": {  
            "type": "string",  
            "description": "A uri that points to tile content. When the uri is  
relative, it is relative to the referring tileset JSON file."  
        },  
        "metadata": {  
            "$ref": "metadataEntity.schema.json",  
            "description": "Metadata that is associated with this content."  
        },  
        "group": {  
            "type": "integer",  
            "minimum": 0,  
            "description": "The group this content belongs to. The value is an  
index into the array of `groups` that is defined for the containing tileset."  
        }  
    },  
    "required": [  
        "uri"  
    ]  
}
```

## B.11. JSON Schema for Definitions

---

```
{  
    "$schema": "https://json-schema.org/draft/2020-12/schema",  
    "$id": "definitions.schema.json",  
    "title": "Definitions",  
    "description": "Common definitions used in schema files.",  
    "definitions": {  
        "numericArray1D": {  
            "title": "Numeric 1D Array",  
            "type": "array",  
            "items": {  
                "type": "number"  
            },  
            "minItems": 1,  
            "description": "An array of numeric values"  
        },  
    },  
}
```

```

"numericArray2D": {
    "title": "Numeric 2D Array",
    "type": "array",
    "items": {
        "$ref": "#/definitions/numericArray1D"
    },
    "minItems": 1,
    "description": "An array of arrays of numeric values"
},
"booleanArray1D": {
    "title": "Boolean 1D Array",
    "type": "array",
    "items": {
        "type": "boolean"
    },
    "minItems": 1,
    "description": "An array of boolean values"
},
"stringArray1D": {
    "title": "String 1D Array",
    "type": "array",
    "items": {
        "type": "string"
    },
    "minItems": 1,
    "description": "An array of string values"
},
"numericValue": {
    "title": "Numeric Value",
    "oneOf": [
        {
            "type": "number"
        },
        {
            "$ref": "#/definitions/numericArray1D"
        },
        {
            "$ref": "#/definitions/numericArray2D"
        }
    ],
    "description": "For `SCALAR` this is a number. For `VECN` this is an array of `N` numbers. For `MATN` this is an array of `N2` numbers. For fixed-length arrays this is an array of `count` elements of the given `type`."
},
"noDataValue": {
    "title": "No Data Value",
    "oneOf": [
        {
            "$ref": "#/definitions/numericValue"
        },
        {
            "type": "string"
        },
        {
            "$ref": "#/definitions/stringArray1D"
        }
    ],
    "description": "For `SCALAR` this is a number. For `STRING` this is a string. For `ENUM` this is a string that shall be a valid enum `name`, not an integer value. For `VECN` this is an array of `N` numbers. For `MATN` this is an array of `N2` numbers. For fixed-length arrays this is an array of `count` elements of the given `type`."
}

```

```

    "anyValue": {
        "title": "Any Value",
        "oneOf": [
            {
                "$ref": "#/definitions/numericValue"
            },
            {
                "type": "string"
            },
            {
                "$ref": "#/definitions/stringArray1D"
            },
            {
                "type": "boolean"
            },
            {
                "$ref": "#/definitions/booleanArray1D"
            }
        ],
        "description": "For `SCALAR` this is a number. For `STRING` this is a string. For `ENUM` this is a string that shall be a valid enum `name`, not an integer value. For `BOOLEAN` this is a boolean. For `VECN` this is an array of `N` numbers. For `MATN` this is an array of `N2` numbers. For fixed-length array this is an array of `count` elements of the given `type`. For variable-length arrays this is an array of any length of the given `type`."
    }
}

```

## B.12. JSON Schema for Enum

---

```

{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "enum.schema.json",
    "title": "Enum",
    "$ref": "rootProperty.schema.json",
    "description": "An object defining the values of an enum.",
    "properties": {
        "name": {
            "type": "string",
            "minLength": 1,
            "description": "The name of the enum, e.g. for display purposes."
        },
        "description": {
            "type": "string",
            "minLength": 1,
            "description": "The description of the enum."
        },
        "valueType": {
            "default": "UINT16",
            "description": "The type of the integer enum value.",
            "anyOf": [
                {
                    "const": "INT8"
                },
                {
                    "const": "UINT8"
                },
                {
                    "const": "INT16"
                },
                {
                    "const": "UINT16"
                }
            ]
        }
    }
}

```

```

        {
            "const": "INT16"
        },
        {
            "const": "UINT16"
        },
        {
            "const": "INT32"
        },
        {
            "const": "UINT32"
        },
        {
            "const": "INT64"
        },
        {
            "const": "UINT64"
        },
        {
            "type": "string"
        }
    ]
},
"values": {
    "type": "array",
    "description": "An array of enum values. Duplicate names or
duplicate integer values are not allowed.",
    "items": {
        "$ref": "enum.value.schema.json"
    },
    "minItems": 1
}
},
"required": [
    "values"
]
}

```

## B.13. JSON Schema for Enum value

---

```

{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "enum.value.schema.json",
    "title": "Enum value",
    "$ref": "rootProperty.schema.json",
    "description": "An enum value.",
    "properties": {
        "name": {
            "type": "string",
            "minLength": 1,
            "description": "The name of the enum value."
        },
        "description": {
            "type": "string",
            "minLength": 1,
            "description": "The description of the enum value."
        },
        "value": {

```

```
        "type": "integer",
        "description": "The integer enum value."
    }
},
"required": [
    "name",
    "value"
]
}
```

## B.14. JSON Schema for Extension

---

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "extension.schema.json",
    "title": "Extension",
    "type": "object",
    "description": "Dictionary object with extension-specific objects.",
    "additionalProperties": {
        "type": "object"
    }
}
```

## B.15. JSON Schema for Extras

---

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "extras.schema.json",
    "title": "Extras",
    "description": "Application-specific data."
}
```

## B.16. JSON Schema for Feature Table

---

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "featureTable.schema.json",
    "title": "Feature Table",
    "$ref": "rootProperty.schema.json",
    "description": "A set of semantics containing per-tile and per-feature values defining the position and appearance properties for features in a tile.",
    "deprecated": true,
    "additionalProperties": {
        "$ref": "#/definitions/property"
    },
    "definitions": {
        "binaryBodyOffset": {
```

```

        "title": "BinaryBodyOffset",
        "$ref": "rootProperty.schema.json",
        "description": "An object defining the offset into a section of
the binary body of the features table where the property values are stored if
not defined directly in the JSON.",
        "properties": {
            "byteOffset": {
                "type": "integer",
                "description": "The offset into the buffer in bytes.",
                "minimum": 0
            }
        },
        "required": [
            "byteOffset"
        ]
    },
    "binaryBodyReference": {
        "title": "BinaryBodyReference",
        "$ref": "#/definitions/binaryBodyOffset",
        "description": "An object defining the reference to a section of
the binary body of the features table where the property values are stored if
not defined directly in the JSON.",
        "properties": {
            "componentType": {
                "description": "The datatype of components in the property.
This is defined only if the semantic allows for overriding the implicit
component type. These cases are specified in each tile format."
            }
        },
        "anyOf": [
            {
                "const": "BYTE"
            },
            {
                "const": "UNSIGNED_BYTE"
            },
            {
                "const": "SHORT"
            },
            {
                "const": "UNSIGNED_SHORT"
            },
            {
                "const": "INT"
            },
            {
                "const": "UNSIGNED_INT"
            },
            {
                "const": "FLOAT"
            },
            {
                "const": "DOUBLE"
            },
            {
                "type": "string"
            }
        ]
    }
},
"property": {
    "title": "Property",
    "description": "A user-defined property which specifies
application-specific metadata in a tile. Values can refer to sections in the

```

```

binary body with a `BinaryBodyReference` object. Global values can be also be
defined directly in the JSON.",
    "oneOf": [
        {
            "$ref": "#/definitions/binaryBodyReference"
        },
        {
            "$ref": "#/definitions/globalPropertyBoolean"
        },
        {
            "$ref": "#/definitions/globalPropertyInteger"
        },
        {
            "$ref": "#/definitions/globalPropertyNumber"
        },
        {
            "$ref": "#/definitions/globalPropertyCartesian3"
        },
        {
            "$ref": "#/definitions/globalPropertyCartesian4"
        }
    ]
},
"globalPropertyBoolean": {
    "title": "GlobalPropertyBoolean",
    "description": "An object defining a global boolean property value
for all features.",
    "type": "boolean"
},
"globalPropertyInteger": {
    "title": "GlobalPropertyInteger",
    "description": "An object defining a global integer property value
for all features.",
    "oneOf": [
        {
            "$ref": "#/definitions/binaryBodyOffset"
        },
        {
            "type": "integer",
            "minimum": 0
        }
    ]
},
"globalPropertyNumber": {
    "title": "GlobalPropertyName",
    "description": "An object defining a global numeric property value
for all features.",
    "oneOf": [
        {
            "$ref": "#/definitions/binaryBodyOffset"
        },
        {
            "type": "number",
            "minimum": 0
        }
    ]
},
"globalPropertyCartesian3": {
    "title": "GlobalPropertyCartesian3",
    "description": "An object defining a global 3-component numeric
property values for all features.",
    "oneOf": [
        {

```

```

        "$ref": "#/definitions/binaryBodyOffset"
    },
    {
        "type": "array",
        "items": {
            "type": "number"
        },
        "minItems": 3,
        "maxItems": 3
    }
]
},
"globalPropertyCartesian4": {
    "title": "GlobalPropertyCartesian4",
    "description": "An object defining a global 4-component numeric property values for all features.",
    "oneOf": [
        {
            "$ref": "#/definitions/binaryBodyOffset"
        },
        {
            "type": "array",
            "items": {
                "type": "number"
            },
            "minItems": 4,
            "maxItems": 4
        }
    ]
}
}

```

## B.17. JSON Schema for Group Metadata

---

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "group.schema.json",
    "title": "Group Metadata",
    "$ref": "metadataEntity.schema.json",
    "description": "An object containing metadata about a group."
}
```

## B.18. JSON Schema for Instanced 3D Model Feature Table

---

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "i3dm.featureTable.schema.json",
    "title": "Instanced 3D Model Feature Table",
    "$ref": "featureTable.schema.json",
}
```

```

    "description": "A set of Instanced 3D Model semantics that contains values defining the position and appearance properties for instanced models in a tile.",
    "deprecated": true,
    "properties": {
        "POSITION": {
            "description": "A `BinaryBodyReference` object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.",
            "$ref": "featureTable.schema.json#/definitions/binaryBodyReference"
        },
        "POSITION_QUANTIZED": {
            "description": "A `BinaryBodyReference` object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.",
            "$ref": "featureTable.schema.json#/definitions/binaryBodyReference"
        },
        "NORMAL_UP": {
            "description": "A `BinaryBodyReference` object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.",
            "$ref": "featureTable.schema.json#/definitions/binaryBodyReference"
        },
        "NORMAL_RIGHT": {
            "description": "A `BinaryBodyReference` object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.",
            "$ref": "featureTable.schema.json#/definitions/binaryBodyReference"
        },
        "NORMAL_UP_OCT32P": {
            "description": "A `BinaryBodyReference` object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.",
            "$ref": "featureTable.schema.json#/definitions/binaryBodyReference"
        },
        "NORMAL_RIGHT_OCT32P": {
            "description": "A `BinaryBodyReference` object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.",
            "$ref": "featureTable.schema.json#/definitions/binaryBodyReference"
        },
        "SCALE": {
            "description": "A `BinaryBodyReference` object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.",
            "$ref": "featureTable.schema.json#/definitions/binaryBodyReference"
        },
        "SCALE_NON_UNIFORM": {
            "description": "A `BinaryBodyReference` object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.",
            "$ref": "featureTable.schema.json#/definitions/binaryBodyReference"
        },
        "BATCH_ID": {
            "description": "A `BinaryBodyReference` object defining the reference to a section of the binary body where the property values are stored. Details about this property are described in the 3D Tiles specification.",
            "$ref": "featureTable.schema.json#/definitions/binaryBodyReference"
        },
        "INSTANCES_LENGTH": {
            "description": "A `GlobalPropertyInteger` object defining an integer property for all features. Details about this property are described in the 3D Tiles specification."
        }
    }
}

```

```

        "$ref": "featureTable.schema.json#/definitions/globalPropertyInteger"
    },
    "RTC_CENTER": {
        "description": "A `GlobalPropertyCartesian3` object defining a 3-component numeric property for all features. Details about this property are described in the 3D Tiles specification.",
        "$ref": "featureTable.schema.json#/definitions/globalPropertyCartesian3"
    },
    "QUANTIZED_VOLUME_OFFSET": {
        "description": "A `GlobalPropertyCartesian3` object defining a 3-component numeric property for all features. Details about this property are described in the 3D Tiles specification.",
        "$ref": "featureTable.schema.json#/definitions/globalPropertyCartesian3"
    },
    "QUANTIZED_VOLUME_SCALE": {
        "description": "A `GlobalPropertyCartesian3` object defining a 3-component numeric property for all features. Details about this property are described in the 3D Tiles specification.",
        "$ref": "featureTable.schema.json#/definitions/globalPropertyCartesian3"
    },
    "EAST_NORTH_UP": {
        "description": "A `GlobalPropertyBoolean` object defining a boolean property for all features. Details about this property are described in the 3D Tiles specification.",
        "$ref": "featureTable.schema.json#/definitions/globalPropertyBoolean"
    }
},
"oneOf": [
{
    "required": [
        "POSITION"
    ]
},
{
    "required": [
        "POSITION_QUANTIZED"
    ]
}
],
"dependencies": {
    "POSITION_QUANTIZED": [
        "QUANTIZED_VOLUME_OFFSET",
        "QUANTIZED_VOLUME_SCALE"
    ],
    "NORMAL_UP": [
        "NORMAL_RIGHT"
    ],
    "NORMAL_RIGHT": [
        "NORMAL_UP"
    ],
    "NORMAL_UP_OCT32P": [
        "NORMAL_RIGHT_OCT32P"
    ],
    "NORMAL_RIGHT_OCT32P": [
        "NORMAL_UP_OCT32P"
    ]
},
"required": [

```

```
        "INSTANCES_LENGTH"
    ]
}
```

## B.19. JSON Schema for Metadata Entity

---

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "metadataEntity.schema.json",
    "title": "Metadata Entity",
    "$ref": "rootProperty.schema.json",
    "description": "An object containing a reference to a class from a metadata schema, and property values that conform to the properties of that class.",
    "properties": {
        "class": {
            "type": "string",
            "description": "The class that property values conform to. The value shall be a class ID declared in the `classes` dictionary of the metadata schema."
        },
        "properties": {
            "type": "object",
            "description": "A dictionary, where each key corresponds to a property ID in the class' `properties` dictionary and each value contains the property values. The type of the value shall match the property definition: For `BOOLEAN` use `true` or `false`. For `STRING` use a JSON string. For numeric types use a JSON number. For `ENUM` use a valid enum `name`, not an integer value. For `ARRAY`, `VECN`, and `MATN` types use a JSON array containing values matching the `componentType`. Required properties shall be included in this dictionary."
            "minProperties": 1,
            "additionalProperties": {
                "$ref": "definitions.schema.json#/definitions/anyValue"
            }
        }
    },
    "required": [
        "class"
    ]
}
```

## B.20. JSON Schema for Point Cloud Feature Table

---

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "pnts.featureTable.schema.json",
    "title": "Point Cloud Feature Table",
    "$ref": "featureTable.schema.json",
    "description": "A set of Point Cloud semantics that contains values defining the position and appearance properties for points in a tile.",
    "deprecated": true,
    "properties": {
```

```

    "POSITION": {
        "description": "A `BinaryBodyReference` object defining the
reference to a section of the binary body where the property values are stored.
Details about this property are described in the 3D Tiles specification.",
        "$ref": "featureTable.schema.json#/definitions/binaryBodyReference"
    },
    "POSITION_QUANTIZED": {
        "description": "A `BinaryBodyReference` object defining the
reference to a section of the binary body where the property values are stored.
Details about this property are described in the 3D Tiles specification.",
        "$ref": "featureTable.schema.json#/definitions/binaryBodyReference"
    },
    "RGBA": {
        "description": "A `BinaryBodyReference` object defining the
reference to a section of the binary body where the property values are stored.
Details about this property are described in the 3D Tiles specification.",
        "$ref": "featureTable.schema.json#/definitions/binaryBodyReference"
    },
    "RGB": {
        "description": "A `BinaryBodyReference` object defining the
reference to a section of the binary body where the property values are stored.
Details about this property are described in the 3D Tiles specification.",
        "$ref": "featureTable.schema.json#/definitions/binaryBodyReference"
    },
    "RGB565": {
        "description": "A `BinaryBodyReference` object defining the
reference to a section of the binary body where the property values are stored.
Details about this property are described in the 3D Tiles specification.",
        "$ref": "featureTable.schema.json#/definitions/binaryBodyReference"
    },
    "NORMAL": {
        "description": "A `BinaryBodyReference` object defining the
reference to a section of the binary body where the property values are stored.
Details about this property are described in the 3D Tiles specification.",
        "$ref": "featureTable.schema.json#/definitions/binaryBodyReference"
    },
    "NORMAL_OCT16P": {
        "description": "A `BinaryBodyReference` object defining the
reference to a section of the binary body where the property values are stored.
Details about this property are described in the 3D Tiles specification.",
        "$ref": "featureTable.schema.json#/definitions/binaryBodyReference"
    },
    "BATCH_ID": {
        "description": "A `BinaryBodyReference` object defining the
reference to a section of the binary body where the property values are stored.
Details about this property are described in the 3D Tiles specification.",
        "$ref": "featureTable.schema.json#/definitions/binaryBodyReference"
    },
    "POINTS_LENGTH": {
        "description": "A `GlobalPropertyInteger` object defining an
integer property for all points. Details about this property are described in
the 3D Tiles specification.",
        "$ref": "featureTable.schema.json#/definitions/globalPropertyInteger"
    },
    "RTC_CENTER": {
        "description": "A `GlobalPropertyCartesian3` object defining a
3-component numeric property for all points. Details about this property are
described in the 3D Tiles specification.",
        "$ref": "featureTable.schema.json#/definitions/globalPropertyCartesian3"
    },
    "QUANTIZED_VOLUME_OFFSET": {

```

```

        "description": "A `GlobalPropertyCartesian3` object defining a
3-component numeric property for all points. Details about this property are
described in the 3D Tiles specification.",
        "$ref": "featureTable.schema.json#/definitions/globalPropertyCartes
ian3"
    },
    "QUANTIZED_VOLUME_SCALE": {
        "description": "A `GlobalPropertyCartesian3` object defining a
3-component numeric property for all points. Details about this property are
described in the 3D Tiles specification.",
        "$ref": "featureTable.schema.json#/definitions/globalPropertyCartes
ian3"
    },
    "CONSTANT_RGBA": {
        "description": "A `GlobalPropertyCartesian4` object defining a
4-component numeric property for all points. Details about this property are
described in the 3D Tiles specification.",
        "$ref": "featureTable.schema.json#/definitions/globalPropertyCartes
ian4"
    },
    "BATCH_LENGTH": {
        "description": "A `GlobalPropertyInteger` object defining an
integer property for all points. Details about this property are described in
the 3D Tiles specification.",
        "$ref": "featureTable.schema.json#/definitions/globalPropertyIntege
r"
    }
},
"anyOf": [
{
    "required": [
        "POSITION"
    ]
},
{
    "required": [
        "POSITION_QUANTIZED"
    ]
}
],
"dependencies": {
    "POSITION_QUANTIZED": [
        "QUANTIZED_VOLUME_OFFSET",
        "QUANTIZED_VOLUME_SCALE"
    ],
    "BATCH_ID": [
        "BATCH_LENGTH"
    ]
},
"required": [
    "POINTS_LENGTH"
]
}
}
```

## B.21. JSON Schema for Properties

---

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
```

```

"$id": "properties.schema.json",
"title": "Properties",
"$ref": "rootProperty.schema.json",
"description": "A dictionary object of metadata about per-feature properties.",
"properties": {
    "maximum": {
        "type": "number",
        "description": "The maximum value of this property of all the features in the tileset."
    },
    "minimum": {
        "type": "number",
        "description": "The minimum value of this property of all the features in the tileset."
    }
},
"required": [
    "maximum",
    "minimum"
]
}

```

## B.22. JSON Schema for Property Table

---

```

{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "propertyTable.schema.json",
    "title": "Property Table",
    "$ref": "rootProperty.schema.json",
    "description": "Properties conforming to a class, organized as property values stored in binary columnar arrays.",
    "properties": {
        "name": {
            "type": "string",
            "minLength": 1,
            "description": "The name of the property table, e.g. for display purposes."
        },
        "class": {
            "type": "string",
            "description": "The class that property values conform to. The value shall be a class ID declared in the `classes` dictionary."
        },
        "count": {
            "type": "integer",
            "minimum": 1,
            "description": "The number of elements in each property array."
        },
        "properties": {
            "type": "object",
            "description": "A dictionary, where each key corresponds to a property ID in the class' `properties` dictionary and each value is an object describing where property values are stored. Required properties shall be included in this dictionary."
        },
        "minProperties": 1,
        "additionalProperties": {
            "$ref": "propertyTable.property.schema.json"
        }
    }
}

```

```

        }
    },
    "required": [
        "class",
        "count"
    ]
}

```

## B.23. JSON Schema for Property Table Property

---

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "propertyTable.property.schema.json",
    "title": "Property Table Property",
    "$ref": "rootProperty.schema.json",
    "description": "An array of binary property values. This represents one column of a property table, and contains one value of a certain property for each metadata entity.",
    "properties": {
        "values": {
            "type": "integer",
            "minimum": 0,
            "description": "The index of the buffer view containing property values. The data type of property values is determined by the property definition: When `type` is `BOOLEAN` values are packed into a bitstream. When `type` is `STRING` values are stored as byte sequences and decoded as UTF-8 strings. When `type` is `SCALAR`, `VECN`, or `MATN` the values are stored as the provided `componentType` and the buffer view `byteOffset` shall be aligned to a multiple of the `componentType` size. When `type` is `ENUM` values are stored as the enum's `valueType` and the buffer view `byteOffset` shall be aligned to a multiple of the `valueType` size. Each enum value in the array shall match one of the allowed values in the enum definition. `arrayOffsets` is required for variable-length arrays and `stringOffsets` is required for strings (for variable-length arrays of strings, both are required)."
        },
        "arrayOffsets": {
            "type": "integer",
            "minimum": 0,
            "description": "The index of the buffer view containing offsets for variable-length arrays. The number of offsets is equal to the property table `count` plus one. The offsets represent the start positions of each array, with the last offset representing the position after the last array. The array length is computed using the difference between the subsequent offset and the current offset. If `type` is `STRING` the offsets index into the string offsets array (stored in `stringOffsets`), otherwise they index into the property array (stored in `values`). The data type of these offsets is determined by `arrayOffsetType`. The buffer view `byteOffset` shall be aligned to a multiple of the `arrayOffsetType` size."
        },
        "stringOffsets": {
            "type": "integer",
            "minimum": 0,
            "description": "The index of the buffer view containing offsets for strings. The number of offsets is equal to the number of string elements plus one. The offsets represent the byte offsets of each string in the property array (stored in `values`), with the last offset representing the byte offset after the last string. The string byte length is computed using"
        }
    }
}
```

the difference between the subsequent offset and the current offset. The data type of these offsets is determined by `stringOffsetType`. The buffer view `byteOffset` shall be aligned to a multiple of the `stringOffsetType` size."

```

        },
        "arrayOffsetType": {
            "description": "The type of values in `arrayOffsets`.",
            "default": "UINT32",
            "anyOf": [
                {
                    "const": "UINT8"
                },
                {
                    "const": "UINT16"
                },
                {
                    "const": "UINT32"
                },
                {
                    "const": "UINT64"
                },
                {
                    "type": "string"
                }
            ]
        },
        "stringOffsetType": {
            "description": "The type of values in `stringOffsets`.",
            "default": "UINT32",
            "anyOf": [
                {
                    "const": "UINT8"
                },
                {
                    "const": "UINT16"
                },
                {
                    "const": "UINT32"
                },
                {
                    "const": "UINT64"
                },
                {
                    "type": "string"
                }
            ]
        },
        "offset": {
            "$ref": "definitions.schema.json#/definitions/numericValue",
            "description": "An offset to apply to property values. Only applicable when the component type is `FLOAT32` or `FLOAT64`, or when the property is `normalized`. Overrides the class property's `offset` if both are defined."
        },
        "scale": {
            "$ref": "definitions.schema.json#/definitions/numericValue",
            "description": "A scale to apply to property values. Only applicable when the component type is `FLOAT32` or `FLOAT64`, or when the property is `normalized`. Overrides the class property's `scale` if both are defined."
        },
        "max": {
            "$ref": "definitions.schema.json#/definitions/numericValue",

```

```

        "description": "Maximum value present in the property values. Only applicable to `SCALAR`, `VECN`, and `MATN` types. This is the maximum of all property values, after the transforms based on the `normalized`, `offset`, and `scale` properties have been applied."
    },
    "min": {
        "$ref": "definitions.schema.json#/definitions/numericValue",
        "description": "Minimum value present in the property values. Only applicable to `SCALAR`, `VECN`, and `MATN` types. This is the minimum of all property values, after the transforms based on the `normalized`, `offset`, and `scale` properties have been applied."
    }
},
"required": [
    "values"
]
}

```

## B.24. JSON Schema for Root Property

---

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "rootProperty.schema.json",
    "title": "Root Property",
    "type": "object",
    "description": "A basis for storing extensions and extras.",
    "properties": {
        "extensions": {
            "$ref": "extension.schema.json"
        },
        "extras": {
            "$ref": "extras.schema.json"
        }
    }
}
```

## B.25. JSON Schema for Schema

---

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "schema.schema.json",
    "title": "Schema",
    "$ref": "rootProperty.schema.json",
    "description": "An object defining classes and enums.",
    "properties": {
        "id": {
            "type": "string",
            "pattern": "^[a-zA-Z_][a-zA-Z0-9_]*$",
            "description": "Unique identifier for the schema. Schema IDs shall be alphanumeric identifiers matching the regular expression `^([a-zA-Z_][a-zA-Z0-9_])*`."
        },
        "name": {

```

```

        "type": "string",
        "minLength": 1,
        "description": "The name of the schema, e.g. for display purposes."
    },
    "description": {
        "type": "string",
        "minLength": 1,
        "description": "The description of the schema."
    },
    "version": {
        "type": "string",
        "minLength": 1,
        "description": "Application-specific version of the schema."
    },
    "classes": {
        "type": "object",
        "description": "A dictionary, where each key is a class ID and each value is an object defining the class. Class IDs shall be alphanumeric identifiers matching the regular expression `^[a-zA-Z_][a-zA-Z0-9_]*$`.",
        "minProperties": 1,
        "additionalProperties": {
            "$ref": "class.schema.json"
        }
    },
    "enums": {
        "type": "object",
        "description": "A dictionary, where each key is an enum ID and each value is an object defining the values for the enum. Enum IDs shall be alphanumeric identifiers matching the regular expression `^[a-zA-Z_][a-zA-Z0-9_]*$`.",
        "minProperties": 1,
        "additionalProperties": {
            "$ref": "enum.schema.json"
        }
    }
},
"required": [
    "id"
]
}

```

## B.26. JSON Schema for Statistics

---

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "statistics.schema.json",
    "title": "Statistics",
    "$ref": "rootProperty.schema.json",
    "description": "Statistics about entities.",
    "properties": {
        "classes": {
            "type": "object",
            "description": "A dictionary, where each key corresponds to a class ID in the `classes` dictionary and each value is an object containing statistics about entities that conform to the class.",
            "minProperties": 1,
            "additionalProperties": {
                "$ref": "statistics.class.schema.json"
            }
        }
    }
}
```

```
        }
    }
}
```

## B.27. JSON Schema for Class Statistics

---

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "statistics.class.schema.json",
  "title": "Class Statistics",
  "$ref": "rootProperty.schema.json",
  "description": "Statistics about entities that conform to a class.",
  "properties": {
    "count": {
      "type": "integer",
      "description": "The number of entities that conform to the class.",
      "minimum": 0
    },
    "properties": {
      "type": "object",
      "description": "A dictionary, where each key corresponds to a property ID in the class' `properties` dictionary and each value is an object containing statistics about property values.",
      "minProperties": 1,
      "additionalProperties": {
        "$ref": "statistics.class.property.schema.json"
      }
    }
  }
}
```

## B.28. JSON Schema for Property Statistics

---

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "statistics.class.property.schema.json",
  "title": "Property Statistics",
  "$ref": "rootProperty.schema.json",
  "description": "Statistics about property values.",
  "properties": {
    "min": {
      "$ref": "definitions.schema.json#/definitions/numericValue",
      "description": "The minimum property value occurring in the tileset. Only applicable to `SCALAR`, `VECN`, and `MATN` types. This is the minimum of all property values, after the transforms based on the `normalized`, `offset`, and `scale` properties have been applied."
    },
    "max": {
      "$ref": "definitions.schema.json#/definitions/numericValue",
      "description": "The maximum property value occurring in the tileset. Only applicable to `SCALAR`, `VECN`, and `MATN` types. This is the
```

```
maximum of all property values, after the transforms based on the `normalized`,  
`offset`, and `scale` properties have been applied."  
},  
    "mean": {  
        "$ref": "definitions.schema.json#/definitions/numericValue",  
        "description": "The arithmetic mean of property values occurring  
in the tileset. Only applicable to `SCALAR`, `VECN`, and `MATN` types.  
This is the mean of all property values, after the transforms based on the  
`normalized`, `offset`, and `scale` properties have been applied."  
},  
    "median": {  
        "$ref": "definitions.schema.json#/definitions/numericValue",  
        "description": "The median of property values occurring in the  
tileset. Only applicable to `SCALAR`, `VECN`, and `MATN` types. This is the  
median of all property values, after the transforms based on the `normalized`,  
`offset`, and `scale` properties have been applied."  
},  
    "standardDeviation": {  
        "$ref": "definitions.schema.json#/definitions/numericValue",  
        "description": "The standard deviation of property values  
occurring in the tileset. Only applicable to `SCALAR`, `VECN`, and `MATN`  
types. This is the standard deviation of all property values, after the  
transforms based on the `normalized`, `offset`, and `scale` properties have  
been applied."  
},  
    "variance": {  
        "$ref": "definitions.schema.json#/definitions/numericValue",  
        "description": "The variance of property values occurring in  
the tileset. Only applicable to `SCALAR`, `VECN`, and `MATN` types. This  
is the variance of all property values, after the transforms based on the  
`normalized`, `offset`, and `scale` properties have been applied."  
},  
    "sum": {  
        "$ref": "definitions.schema.json#/definitions/numericValue",  
        "description": "The sum of property values occurring in the  
tileset. Only applicable to `SCALAR`, `VECN`, and `MATN` types. This is the  
sum of all property values, after the transforms based on the `normalized`,  
`offset`, and `scale` properties have been applied."  
},  
    "occurrences": {  
        "type": "object",  
        "description": "A dictionary, where each key corresponds to an  
enum `name` and each value is the number of occurrences of that enum. Only  
applicable when `type` is `ENUM`. For fixed-length arrays, this is an array of  
component-wise occurrences.",  
        "minProperties": 1,  
        "additionalProperties": {  
            "oneOf": [  
                {  
                    "type": "integer"  
                },  
                {  
                    "type": "array",  
                    "items": {  
                        "type": "integer"  
                    },  
                    "minItems": 1  
                }  
            ]  
        }  
    }  
}
```

```
}
```

## B.29. JSON Schema for Style

---

```
{  
    "$schema": "https://json-schema.org/draft/2020-12/schema",  
    "$id": "style.schema.json",  
    "title": "Style",  
    "$ref": "rootProperty.schema.json",  
    "description": "A 3D Tiles style.",  
    "properties": {  
        "defines": {  
            "type": "object",  
            "additionalProperties": {  
                "$ref": "style.expression.schema.json"  
            },  
            "description": "A dictionary object of `expression` strings mapped  
to a variable name key that may be referenced throughout the style. If an  
expression references a defined variable, it is replaced with the evaluated  
result of the corresponding expression."  
        },  
        "show": {  
            "oneOf": [  
                {  
                    "$ref": "style.booleanExpression.schema.json"  
                },  
                {  
                    "$ref": "style.conditions.schema.json"  
                }  
            ],  
            "description": "A `boolean expression` or `conditions` property  
which determines if a feature should be shown.",  
            "default": "true"  
        },  
        "color": {  
            "oneOf": [  
                {  
                    "$ref": "style.colorExpression.schema.json"  
                },  
                {  
                    "$ref": "style.conditions.schema.json"  
                }  
            ],  
            "description": "A `color expression` or `conditions` property  
which determines the color blended with the feature's intrinsic color.",  
            "default": "color('#FFFFFF')"  
        },  
        "meta": {  
            "$ref": "style.meta.schema.json",  
            "description": "A `meta` object which determines the values of non-  
visual properties of the feature."  
        }  
    }  
}
```

## B.30. JSON Schema for Boolean Expression

---

```
{  
    "$schema": "https://json-schema.org/draft/2020-12/schema",  
    "$id": "style.booleanExpression.schema.json",  
    "title": "Boolean Expression",  
    "type": [  
        "boolean",  
        "string"  
    ],  
    "description": "A boolean or string with a 3D Tiles style expression  
that evaluates to a boolean. Details are described in the 3D Tiles Styling  
specification."  
}
```

## B.31. JSON Schema for Color Expression

---

```
{  
    "$schema": "https://json-schema.org/draft/2020-12/schema",  
    "$id": "style.colorExpression.schema.json",  
    "title": "Color Expression",  
    "type": "string",  
    "description": "3D Tiles style `expression` that evaluates to a Color.  
Details are described in the 3D Tiles Styling specification."  
}
```

## B.32. JSON Schema for Conditions

---

```
{  
    "$schema": "https://json-schema.org/draft/2020-12/schema",  
    "$id": "style.conditions.schema.json",  
    "title": "Conditions",  
    "$ref": "rootProperty.schema.json",  
    "description": "A series of conditions evaluated in order, like a series  
of if...else statements that result in an expression being evaluated.",  
    "properties": {  
        "conditions": {  
            "type": "array",  
            "description": "A series of boolean conditions evaluated in order.  
For the first one that evaluates to true, its value, the 'result' (which is  
also an expression), is evaluated and returned. Result expressions shall all  
be the same type. If no condition evaluates to true, the result is `undefined`.  
When conditions is `undefined`, `null`, or an empty object, the result is  
`undefined`. ",  
            "items": {  
                "$ref": "style.conditions.condition.schema.json"  
            }  
        }  
    }  
}
```

```
}
```

## B.33. JSON Schema for Condition

---

```
{  
    "$schema": "https://json-schema.org/draft/2020-12/schema",  
    "$id": "style.conditions.condition.schema.json",  
    "title": "Condition",  
    "type": "array",  
    "description": "An `expression` evaluated as the result of a condition being true. An array of two expressions. If the first expression is evaluated and the result is `true`, then the second expression is evaluated and returned as the result of the condition.",  
    "items": {  
        "$ref": "style.expression.schema.json"  
    },  
    "minItems": 2,  
    "maxItems": 2  
}
```

## B.34. JSON Schema for Expression

---

```
{  
    "$schema": "https://json-schema.org/draft/2020-12/schema",  
    "$id": "style.expression.schema.json",  
    "title": "Expression",  
    "type": "string",  
    "description": "A valid 3D Tiles style expression. Details are described in the 3D Tiles Styling specification."  
}
```

## B.35. JSON Schema for Meta

---

```
{  
    "$schema": "https://json-schema.org/draft/2020-12/schema",  
    "$id": "style.meta.schema.json",  
    "title": "Meta",  
    "$ref": "rootProperty.schema.json",  
    "description": "A series of property names and the `expression` to evaluate for the value of that property.",  
    "additionalProperties": {  
        "$ref": "style.expression.schema.json"  
    }  
}
```

## B.36. JSON Schema for Subtree

---

```
{  
    "$schema": "https://json-schema.org/draft/2020-12/schema",  
    "$id": "subtree.schema.json",  
    "title": "Subtree",  
    "$ref": "rootProperty.schema.json",  
    "description": "An object describing the availability of tiles and content  
in a subtree, as well as availability of children subtrees. May also store  
metadata for available tiles and content.",  
    "properties": {  
        "buffers": {  
            "type": "array",  
            "items": {  
                "$ref": "buffer.schema.json"  
            },  
            "minItems": 1,  
            "description": "An array of buffers."  
        },  
        "bufferViews": {  
            "type": "array",  
            "items": {  
                "$ref": "bufferView.schema.json"  
            },  
            "minItems": 1,  
            "description": "An array of buffer views."  
        },  
        "propertyTables": {  
            "type": "array",  
            "items": {  
                "$ref": "propertyTable.schema.json"  
            },  
            "minItems": 1,  
            "description": "An array of property tables."  
        },  
        "tileAvailability": {  
            "$ref": "availability.schema.json",  
            "description": "The availability of tiles in the subtree. The  
availability bitstream is a 1D boolean array where tiles are ordered by their  
level in the subtree and Morton index within that level. A tile's availability  
is determined by a single bit, 1 meaning a tile exists at that spatial  
index, and 0 meaning it does not. The number of elements in the array is  
` $(N^{\text{subtreeLevels}} - 1)/(N - 1)$ ` where N is 4 for subdivision scheme `QUADTREE`  
and 8 for `OCTREE`. Availability may be stored in a buffer view or as a  
constant value that applies to all tiles. If a non-root tile's availability is  
1 its parent tile's availability shall also be 1. `tileAvailability.constant:  
0` is disallowed, as subtrees shall have at least one tile."  
        },  
        "contentAvailability": {  
            "type": "array",  
            "items": {  
                "$ref": "availability.schema.json"  
            },  
            "minItems": 1,  
            "description": "An array of content availability objects. If the  
tile has a single content this array will have one element; if the tile has  
multiple contents - as supported by 3DTILES_multiple_contents and 3D Tiles 1.1  
- this array will have multiple elements."  
        },  
    }  
}
```

```

    "childSubtreeAvailability": {
        "$ref": "availability.schema.json",
        "description": "The availability of children subtrees. The availability bitstream is a 1D boolean array where subtrees are ordered by their Morton index in the level of the tree immediately below the bottom row of the subtree. A child subtree's availability is determined by a single bit, 1 meaning a subtree exists at that spatial index, and 0 meaning it does not. The number of elements in the array is `N^subtreeLevels` where N is 4 for subdivision scheme `QUADTREE` and 8 for `OCTREE`. Availability may be stored in a buffer view or as a constant value that applies to all child subtrees. If availability is 0 for all child subtrees, then the tileset does not subdivide further."
    },
    "tileMetadata": {
        "type": "integer",
        "minimum": 0,
        "description": "Index of the property table containing tile metadata. Tile metadata only exists for available tiles and is tightly packed by increasing tile index. To access individual tile metadata, implementations may create a mapping from tile indices to tile metadata indices."
    },
    "contentMetadata": {
        "type": "array",
        "items": {
            "type": "integer",
            "minimum": 0
        },
        "minItems": 1,
        "description": "An array of indexes to property tables containing content metadata. If the tile has a single content this array will have one element; if the tile has multiple contents - as supported by 3DTILES_multiple_contents and 3D Tiles 1.1 - this array will have multiple elements. Content metadata only exists for available contents and is tightly packed by increasing tile index. To access individual content metadata, implementations may create a mapping from tile indices to content metadata indices."
    },
    "subtreeMetadata": {
        "$ref": "metadataEntity.schema.json",
        "description": "Subtree metadata encoded in JSON."
    }
},
"required": [
    "tileAvailability",
    "childSubtreeAvailability"
]
}

```

## B.37. JSON Schema for Subtrees

---

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "subtrees.schema.json",
    "title": "Subtrees",
    "$ref": "rootProperty.schema.json",
    "description": "An object describing the location of subtree files.",
    "properties": {
        "uri": {
            "$ref": "templateUri.schema.json",

```

```

        "description": "A template URI pointing to subtree files. A
subtree is a fixed-depth (defined by `subtreeLevels`) portion of the tree to
keep memory use bounded. The URI of each file is substituted with the subtree
root's global level, x, and y. For subdivision scheme `OCTREE`, z shall also
be given. Relative paths are relative to the tileset JSON."
    }
},
"required": [
    "uri"
]
}

```

## B.38. JSON Schema for Template URI

---

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "templateUri.schema.json",
  "title": "Template URI",
  "type": "string",
  "description": "A URI with embedded expressions that describes the
resource that is associated with an implicit tile in an implicit tileset.
Allowed expressions are `'{level}'`, `'{x}'`, `'{y}'`, and `'{z}'`. `'{level}'` is
substituted with the level of the node, `'{x}'` is substituted with the x index
of the node within the level, and `'{y}'` is substituted with the y index of the
node within the level. `'{z}'` may only be given when the subdivision scheme is
`'OCTREE'`, and it is substituted with the z index of the node within the level."
}
```

## B.39. JSON Schema for Tile

---

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "tile.schema.json",
  "title": "Tile",
  "$ref": "rootProperty.schema.json",
  "description": "A tile in a 3D Tiles tileset.",
  "properties": {
    "boundingVolume": {
      "description": "The bounding volume that encloses the tile.",
      "$ref": "boundingVolume.schema.json"
    },
    "viewerRequestVolume": {
      "description": "Optional bounding volume that defines the volume
the viewer shall be inside of before the tile's content will be requested and
before the tile will be refined based on geometricError.",
      "$ref": "boundingVolume.schema.json"
    },
    "geometricError": {
      "type": "number",
      "description": "The error, in meters, introduced if this tile is
rendered and its children are not. At runtime, the geometric error is used to
compute screen space error (SSE), i.e., the error measured in pixels.",
      "minimum": 0
    }
  }
}
```

```

},
"refine": {
  "description": "Specifies if additive or replacement refinement is used when traversing the tileset for rendering. This property is required for the root tile of a tileset; it is optional for all other tiles. The default is to inherit from the parent tile.",
  "anyOf": [
    {
      "const": "ADD"
    },
    {
      "const": "REPLACE"
    },
    {
      "type": "string"
    }
  ]
},
"transform": {
  "type": "array",
  "description": "A floating-point 4x4 affine transformation matrix, stored in column-major order, that transforms the tile's content--i.e., its features as well as content.boundingVolume, boundingVolume, and viewerRequestVolume--from the tile's local coordinate system to the parent tile's coordinate system, or, in the case of a root tile, from the tile's local coordinate system to the tileset's coordinate system. `transform` does not apply to any volume property when the volume is a region, defined in EPSG:4979 coordinates. `transform` scales the `geometricError` by the maximum scaling factor from the matrix.",
  "items": {
    "type": "number"
  },
  "minItems": 16,
  "maxItems": 16,
  "default": [
    1.0,
    0.0,
    0.0,
    0.0,
    0.0,
    1.0,
    0.0,
    0.0,
    0.0,
    0.0,
    0.0,
    1.0,
    0.0,
    0.0,
    0.0,
    0.0,
    1.0
  ]
},
"content": {
  "description": "Metadata about the tile's content and a link to the content. When this is omitted the tile is just used for culling. When this is defined, then `contents` shall be undefined."
},
"$ref": "content.schema.json"
},
"contents": {
  "type": "array",
  "description": "An array of contents. When this is defined, then `content` shall be undefined."
}

```

```

        "items": {
            "$ref": "content.schema.json"
        },
        "minItems": 1
    },
    "metadata": {
        "$ref": "metadataEntity.schema.json",
        "description": "A metadata entity that is associated with this tile."
    },
    "implicitTiling": {
        "$ref": "tile.implicitTiling.schema.json",
        "description": "An object that describes the implicit subdivision of this tile."
    },
    "children": {
        "type": "array",
        "description": "An array of objects that define child tiles. Each child tile content is fully enclosed by its parent tile's bounding volume and, generally, has a geometricError less than its parent tile's geometricError. For leaf tiles, the length of this array is zero, and children may not be defined.",
        "items": {
            "$ref": "tile.schema.json"
        },
        "uniqueItems": true,
        "minItems": 1
    }
},
"required": [
    "boundingVolume",
    "geometricError"
],
"not": {
    "required": [
        "content",
        "contents"
    ]
}
}

```

## B.40. JSON Schema for Implicit tiling

---

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "tile.implicitTiling.schema.json",
    "title": "Implicit tiling",
    "$ref": "rootProperty.schema.json",
    "description": "This object allows a tile to be implicitly subdivided. Tile and content availability and metadata is stored in subtrees which are referenced externally.",
    "properties": {
        "subdivisionScheme": {
            "description": "A string describing the subdivision scheme used within the tileset."
        },
        "anyOf": [
            {
                "const": "QUADTREE"
            }
        ]
    }
}
```

```

        },
        {
            "const": "OCTREE"
        },
        {
            "type": "string"
        }
    ]
},
"subtreeLevels": {
    "type": "integer",
    "description": "The number of distinct levels in each subtree. For example, a quadtree with `subtreeLevels = 2` will have subtrees with 5 nodes (one root and 4 children).",
    "minimum": 1
},
"availableLevels": {
    "type": "integer",
    "description": "The numbers of the levels in the tree with available tiles.",
    "minimum": 1
},
"subtrees": {
    "$ref": "subtrees.schema.json",
    "description": "An object describing the location of subtree files."
}
},
"required": [
    "subdivisionScheme",
    "subtreeLevels",
    "availableLevels",
    "subtrees"
]
}

```

## B.41. JSON Schema for Tileset

---

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "tileset.schema.json",
    "title": "Tileset",
    "$ref": "rootProperty.schema.json",
    "description": "A 3D Tiles tileset.",
    "properties": {
        "asset": {
            "description": "Metadata about the entire tileset.",
            "$ref": "asset.schema.json"
        },
        "properties": {
            "type": "object",
            "description": "A dictionary object of metadata about per-feature properties."
        },
        "additionalProperties": {
            "$ref": "properties.schema.json"
        },
        "deprecated": true
    }
}
```

```

    "schema": {
        "$ref": "schema.schema.json",
        "description": "An object defining the structure of metadata classes and enums. When this is defined, then `schemaUri` shall be undefined."
    },
    "schemaUri": {
        "type": "string",
        "description": "The URI (or IRI) of the external schema file. When this is defined, then `schema` shall be undefined.",
        "format": "iri-reference"
    },
    "statistics": {
        "$ref": "statistics.schema.json",
        "description": "An object containing statistics about metadata entities."
    },
    "groups": {
        "type": "array",
        "description": "An array of groups that tile content may belong to. Each element of this array is a metadata entity that describes the group. The tile content `group` property is an index into this array.",
        "items": {
            "$ref": "group.schema.json"
        },
        "minItems": 1
    },
    "metadata": {
        "$ref": "metadataEntity.schema.json",
        "description": "A metadata entity that is associated with this tileset."
    },
    "geometricError": {
        "type": "number",
        "description": "The error, in meters, introduced if this tileset is not rendered. At runtime, the geometric error is used to compute screen space error (SSE), i.e., the error measured in pixels.",
        "minimum": 0
    },
    "root": {
        "description": "The root tile.",
        "$ref": "tile.schema.json"
    },
    "extensionsUsed": {
        "type": "array",
        "description": "Names of 3D Tiles extensions used somewhere in this tileset.",
        "items": {
            "type": "string"
        },
        "uniqueItems": true,
        "minItems": 1
    },
    "extensionsRequired": {
        "type": "array",
        "description": "Names of 3D Tiles extensions required to properly load this tileset.",
        "items": {
            "type": "string"
        },
        "uniqueItems": true,
        "minItems": 1
    }
}

```

```
"required": [
    "asset",
    "geometricError",
    "root"
],
"not": {
    "required": [
        "schema",
        "schemaUri"
    ]
}
}
```

C

## ANNEX C (INFORMATIVE) MIGRATION FROM LEGACY TILE FORMATS

---

# ANNEX C (INFORMATIVE)

## MIGRATION FROM LEGACY TILE FORMATS

---

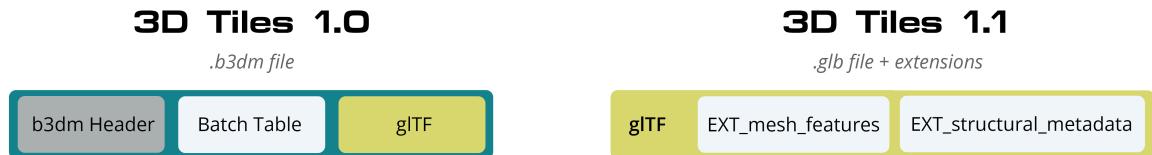
This section describes how legacy tile formats can be converted into equivalent glTF content.

### C.1. Batched 3D Model (b3dm)

---

Batched 3D Model is a wrapper around a binary glTF that includes additional information in its Feature Table and Batch Table. Batched 3D Model content can be converted into glTF content with the following changes:

- The RTC\_CENTER can be added to the translation component of the root node of the glTF asset.
- Batch IDs and Batch Tables can be represented using `EXT_mesh_features` and `EXT_structural_metadata`.



**Figure C.1** – Batched 3D Models in 3D Tiles 1.0, and the corresponding representation in 3D Tiles 1.1

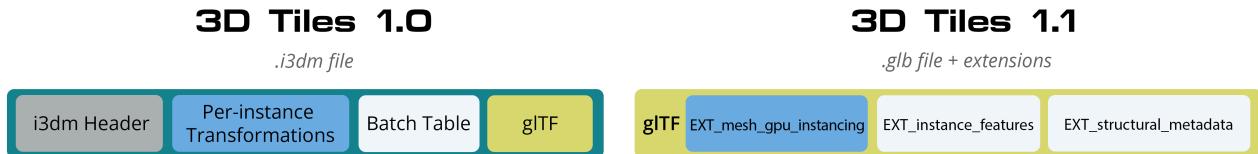
### C.2. Instanced 3D Model (i3dm)

---

Instanced 3D Model instances a glTF asset (embedded or external) and provides per-instance transforms and batch IDs.

- The RTC\_CENTER can be added to the translation component of the root node of the glTF asset.
- glTF can leverage GPU instancing with the `EXT_mesh_gpu_instancing` extension.

- Batch IDs and Batch Tables can be represented using `EXT_instance_features` and `EXT_structural_metadata`.
- `EAST_NORTH_UP` is not directly supported, but can be represented using per-instance rotations.



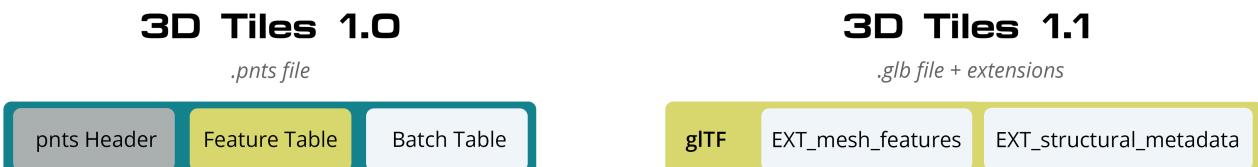
**Figure C.2 – Instanced 3D Models in 3D Tiles 1.0, and the corresponding representation in 3D Tiles 1.1**

### C.3. Point Cloud (pnts)

---

Point Cloud can be represented as a glTF using the primitive mode 0 (POINTS).

- The `RTC_CENTER` can be added to the translation component of the root node of the glTF asset.
- Feature table properties like POSITION, COLOR, and NORMAL may be stored as glTF attributes.
- `EXT_meshopt_compression` and `KHR_mesh_quantization` may be used for point cloud compression. `3DTILES_draco_point_compression` is not directly supported in glTF because `KHR_draco_mesh_compression` only supports triangle meshes.
- Batch IDs and Batch Tables can be represented using `EXT_mesh_features` and `EXT_structural_metadata`.
- `CONSTANT_RGBA` is not directly supported in glTF, but can be achieved with materials or per-point colors.



**Figure C.3 – Point Clouds in 3D Tiles 1.0, and the corresponding representation in 3D Tiles 1.1**

## C.4. Composite (cmpt)

---

All inner contents of a Composite may be combined into the same glTF as separate nodes, meshes, or primitives, at the tilesset author's discretion. Alternatively, a tile may have multiple contents.

D

# ANNEX D (INFORMATIVE) AVAILABILITY INDEXING

---

## ANNEX D (INFORMATIVE) AVAILABILITY INDEXING

### D.1. Converting from Tile Coordinates to Morton Index

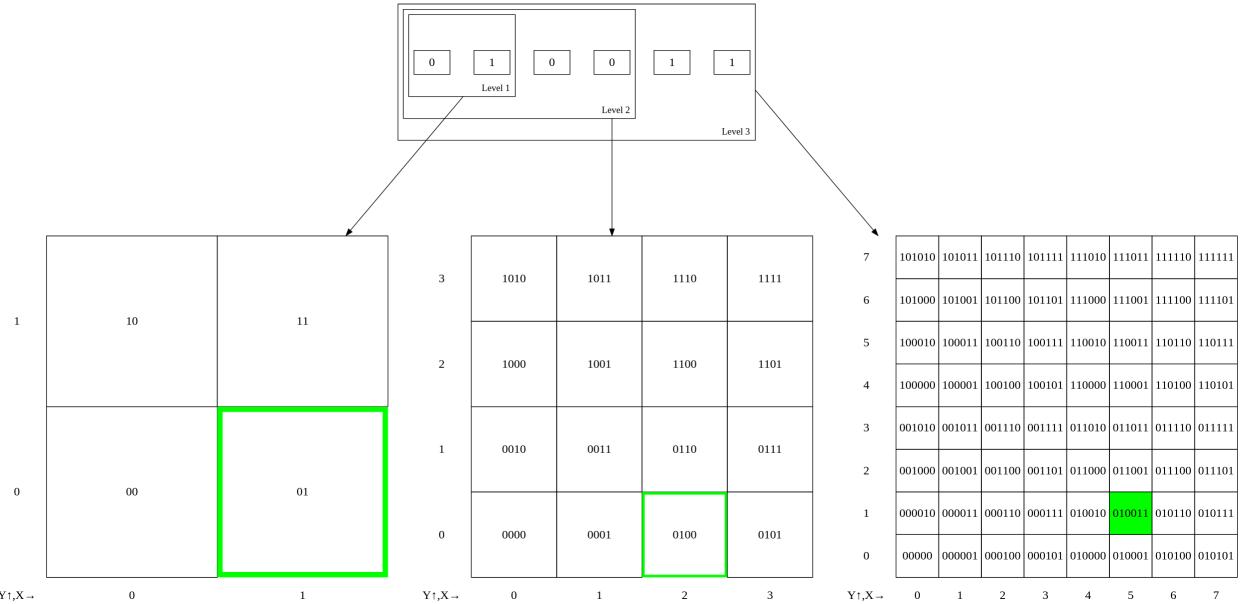
A Morton index is computed by interleaving the bits of the (x, y) or (x, y, z) coordinates of a tile. Specifically:

```
quadtreeMortonIndex = interleaveBits(x, y)
octreeMortonIndex = interleaveBits(x, y, z)
```

For example:

```
// Quadtree
interleaveBits(0b11, 0b00) = 0b0101
interleaveBits(0b1010, 0b0011) = 0b01001110
interleaveBits(0b0110, 0b0101) = 0b00110110

// Octree
interleaveBits(0b001, 0b010, 0b100) = 0b100010001
interleaveBits(0b111, 0b000, 0b111) = 0b101101101
```



**Figure D.1** – An example for the computation of the Morton index based on the tile coordinates on three levels of a quadtree

## D.2. Availability Bitstream Lengths

---

**Table D.1** – Availability bitstream lengths

AVAILABILITY TYPE	LENGTH (BITS)	DESCRIPTION
Tile availability	$(N^{\text{subtreeLevels}} - 1)/(N - 1)$	Total number of nodes in the subtree
Content availability	$(N^{\text{subtreeLevels}} - 1)/(N - 1)$	Since there is at most one content per tile, this is the same length as tile availability
Child subtree availability	$N^{\text{subtreeLevels}}$	Number of nodes one level deeper than the deepest level of the subtree

Where N is 4 for quadtrees and 8 for octrees.

These lengths are in number of bits in a bitstream. To compute the length of the bitstream in bytes, the following formula is used:

```
lengthBytes = ceil(lengthBits / 8)
```

## D.3. Accessing Availability Bits

---

For tile availability and content availability, the Morton index only determines the ordering within a single level of the subtree. Since the availability bitstream stores bits for every level of the subtree, a level offset shall be computed.

Given the `(level, mortonIndex)` of a tile relative to the subtree root, the index of the corresponding bit can be computed with the following formulas:

**Table D.2 – Availability offsets**

QUANTITY	FORMULA	DESCRIPTION
levelOffset	$(N^{\text{level}} - 1) / (N - 1)$	This is the number of nodes at levels 1, 2, ... (level - 1)
tileAvailabilityIndex	levelOffset + mortonIndex	The index into the buffer view is the offset for the tile's level plus the morton index for the tile

Where N is 4 for quadtrees and 8 for octrees.

Since child subtree availability stores bits for a single level, no levelOffset is needed, i.e. `childSubtreeAvailabilityIndex = mortonIndex`, where the `mortonIndex` is the Morton index of the desired child subtree relative to the root of the current subtree.

## D.4. Global and Local Tile Coordinates

---

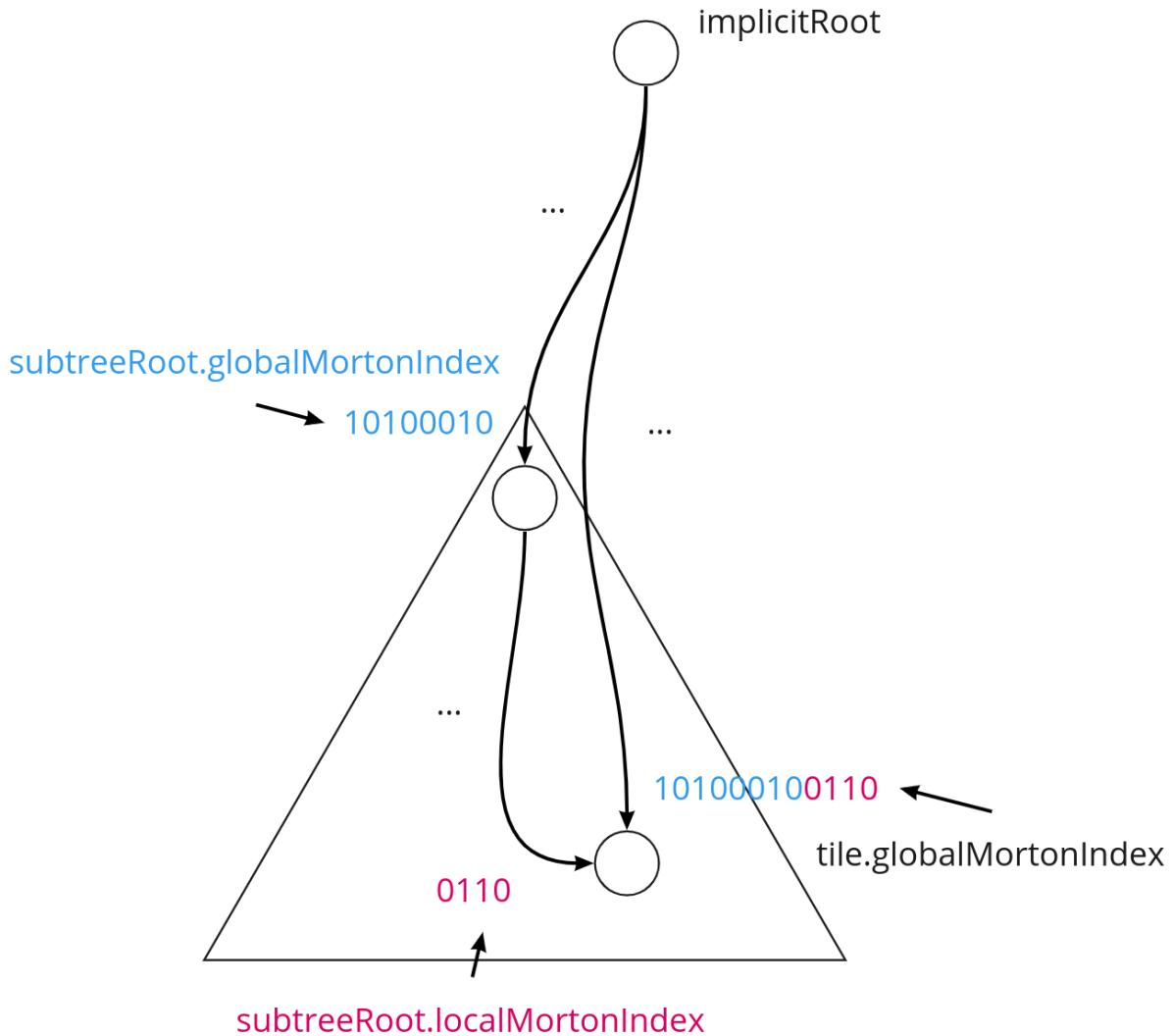
When working with tile coordinates, it is important to consider which tile the coordinates are relative to. There are two main types used in implicit tiling:

- **global coordinates** – coordinates relative to the implicit root tile.
- **local coordinates** – coordinates relative to the root of a specific subtree.

Global coordinates are used for locating any tile in the entire implicit tileset. For example, template URIs use global coordinates to locate content files and subtrees. Meanwhile, local coordinates are used for locating data within a single subtree file.

In binary, a tile's global Morton index is the complete path from the implicit root tile to the tile. This is the concatenation of the path from the implicit root tile to the subtree root tile, followed by the path from the subtree root tile to the tile. This can be expressed with the following equation:

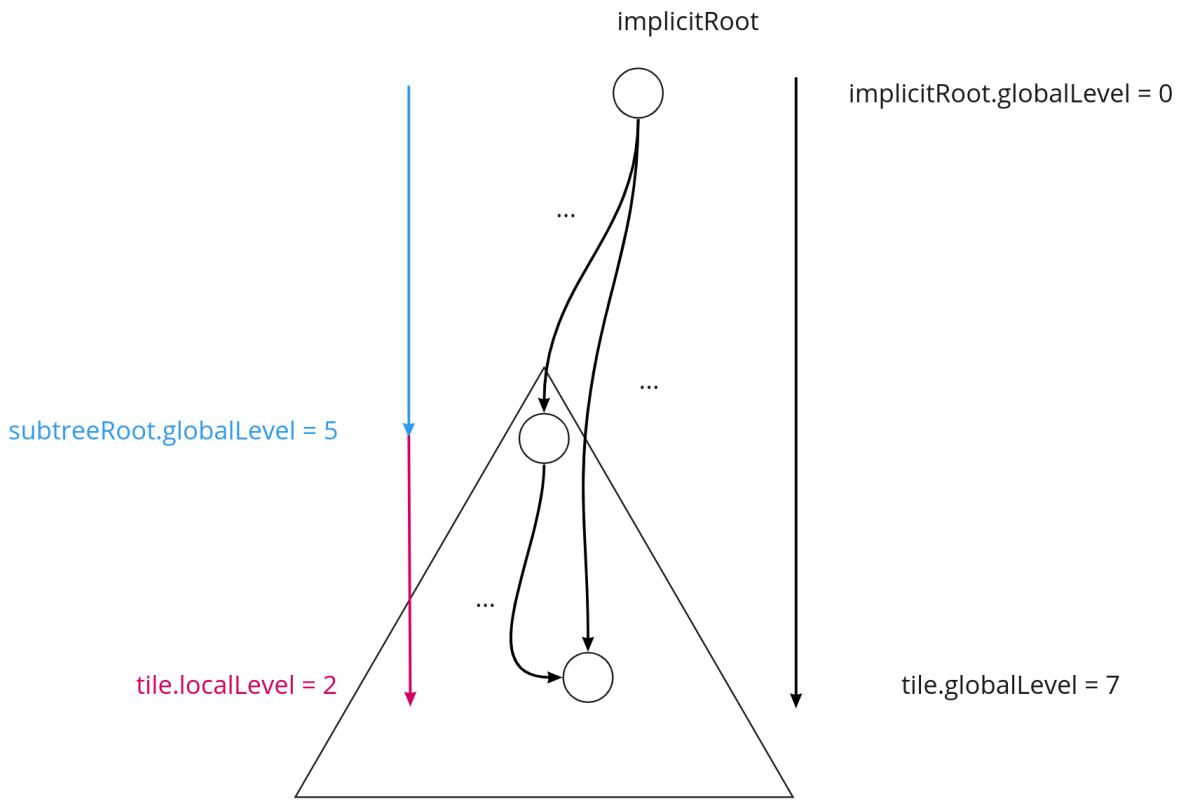
```
tile.globalMortonIndex = concatBits(subtreeRoot.globalMortonIndex, tile.localMortonIndex)
```



**Figure D.2** – Illustration of how to compute the global Morton index of a tile, from the global Morton index of the root of the containing subtree, and the local Morton index of the tile in this subtree

Similarly, the global level of a tile is the length of the path from the implicit root tile to the tile. This is the sum of the subtree root tile's global level and the tile's local level relative to the subtree root tile:

```
tile.globalLevel = subtreeRoot.globalLevel + tile.localLevel
```



**Figure D.3** — Illustration of how to compute the global level of a tile, from the global level of the root of the containing subtree, and the local level of the tile in this subtree

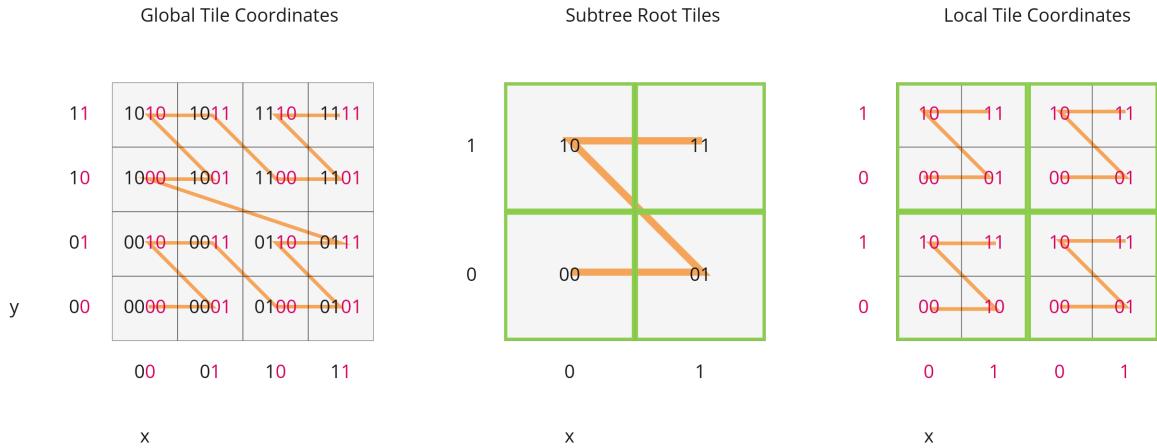
$(x, y, z)$  coordinates follow the same pattern as Morton indices. The only difference is that the concatenation of bits happens component-wise. That is:

```

tile.globalX = concatBits(subtreeRoot.globalX, tile.localX)
tile.globalY = concatBits(subtreeRoot.globalY, tile.localY)

// Octrees only
tile.globalZ = concatBits(subtreeRoot.globalZ, tile.localZ)

```



**Figure D.4** – Illustration of the computation of the global tile coordinates, from the global coordinates of the containing subtree, and the local coordinates of the tile in this subtree.

## D.5. Finding Parent and Child Tiles

---

The coordinates of a parent or child tile can also be computed with bitwise operations on the Morton index. The following formulas apply for both local and global coordinates.

```

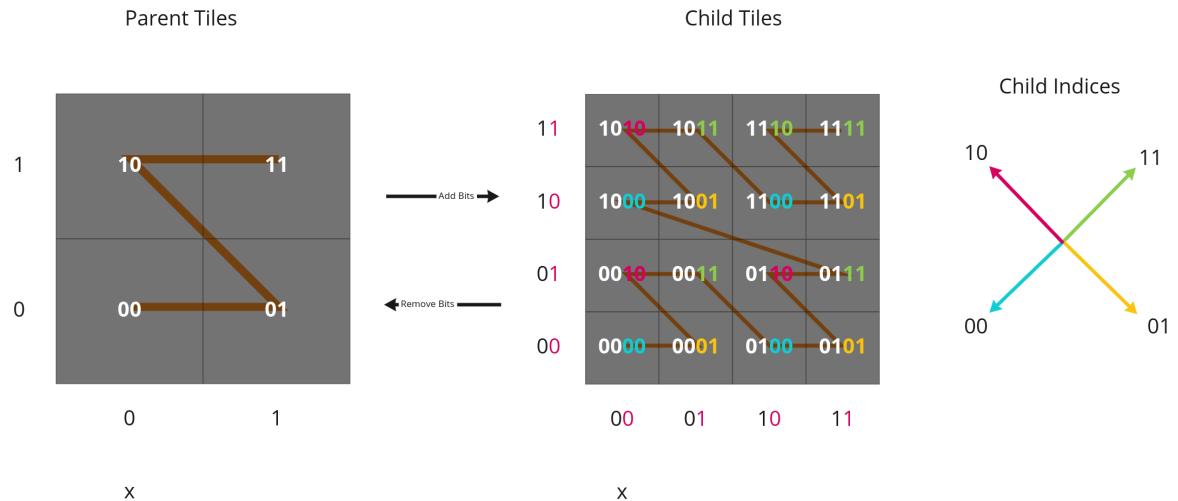
childTile.level = parentTile.level + 1
childTile.mortonIndex = concatBits(parentTile.mortonIndex, childIndex)
childTile.x = concatBits(parentTile.x, childX)
childTile.y = concatBits(parentTile.y, childY)

// Octrees only
childTile.z = concatBits(parentTile.z, childZ)

```

Where:

- `childIndex` is an integer in the range  $[0, N]$  that is the index of the child tile relative to the parent.
- `childX`, `childY`, and `childZ` are single bits that represent which half of the parent's bounding volume the child is in in each direction.



**Figure D.5 – Illustration of the computation of the coordinates of parent- and child tiles**



E

# ANNEX E (INFORMATIVE) 3D METADATA REFERENCE IMPLEMENTATION

---

## ANNEX E

### (INFORMATIVE)

## 3D METADATA REFERENCE IMPLEMENTATION

---

This document defines a reference implementation of the concepts defined in the 3D Metadata Specification. The 3D Metadata Specification itself defines a standard format for structured metadata in 3D content in a way that is language- and format agnostic. The reference implementation described here is an implementation of these concepts:

- The Schema is a JSON-based representation of 3D Metadata Schemas that describe the structure and types of metadata
- The PropertyTable is one form of a Binary Table Format. It is a JSON-based description of how large amounts of metadata can be stored compactly in a binary form.

These serialization formats are used as a common basis for different implementations of the 3D Metadata Specification:

- 3D Tiles Metadata – Assigns metadata to tilesets, tiles, and contents in 3D Tiles 1.1
- 3DTILES\_metadata – An extension for 3D Tiles 1.0 that assigns metadata to tilesets, tiles, and contents
- EXT\_structural\_metadata (gltf 2.0) – Assigns metadata to vertices, texels, and features in a glTF asset

### E.1. Property Table Implementation

---

The 3D Metadata Specification defines schemas as a description of the structure of metadata, and different storage formats for the metadata. One form of storing metadata is that of a Binary Table Format, where the data is stored in a binary representation of a table. Each column of such a table represents one of the properties of a class. Each row represents a single entity conforming to the class. The following is the description of such a binary table format, referred to as **property table**. It is used as the basis for defining the metadata storage in the following implementations:

- 3D Tiles Metadata Implicit Tilesets – Assigns metadata to tilesets, tiles, groups, and contents in a 3D Tiles tileset. A property table is defined for subtrees of an implicit tile hierarchy, and stores metadata that is associated with the nodes of such a subtree.
- `EXT_structural_metadata` – Assigns metadata to vertices, texels, and features in a glTF asset. A property table is defined in the top-level extension object. The property values are stored in standard glTF buffer views.

The full JSON schema definition for this implementation can be found in [the PropertyTable directory of the specification](#).

### E.1.1. Property Tables

*Defined in `propertyTable.schema.json`.*

A property table shall specify the following elements:

- Its class (`class`), which refers to a class ID in a Schema.
- A dictionary of properties (`properties`), where each entry describes the binary storage of the data for the corresponding class property.
- A count (`count`) for the number of elements (`rows`) in the property table.

The property table may provide value arrays for only a subset of the properties of its class, but class properties that are marked required: `true` in the schema shall not be omitted.

**NOTE** A `tree_survey_2021-09-29` property table, implementing the `tree` class defined in the Schema examples. The table contains observations for 10 trees. Details about the class properties will be given in later examples.

```
"schema": { ... },
"propertyTables": [
  {
    "name": "tree_survey_2021-09-29",
    "class": "tree",
    "count": 10,
    "properties": {
      "species": { ... },
      "age": { ... },
      "height": { ... },
      // "diameter" is not required and has been omitted from this table.
    }
  }
]
```

### E.1.2. Property Table Properties

*Defined in `propertyTable.schema.json`.*

Each property definition in a property table represents one column of the table. This column data is stored in binary form, using the encoding defined in the Binary Table Format section of

the 3D Metadata Specification. The actual data is stored in a binary buffer, and the property refers to a section of this buffer that is called a *buffer view*.

- In the 3D Tiles Metadata implementation, a buffer view is defined as part of subtrees in implicit tilesets.
- In the `3DTILES_metadata` extension for 3D Tiles 1.0, a buffer view is defined as part of subtrees in the `3DTILES_implicit_tiling` extension.
- In the `EXT_structural_metadata`, a buffer view is a standard glTF buffer view.

The exact structure of each property entry depends on the property type:

- Every property definition shall define the values that store the raw data of the actual values.
- Properties that have the STRING component type shall define the `stringOffsets`, as defined in Strings.
- Properties that are variable-length arrays shall define the `arrayOffsets`, as defined in Arrays.

For variable-length arrays of strings, both the `stringOffsets` and the `arrayOffsets` are required.

**NOTE 1**The property table from the previous example, with details about the binary storage of the property values

```
{  
  "propertyTables": [{  
    "name": "tree_survey_2021-09-29",  
    "class": "tree",  
    "count": 10,  
    "properties": {  
      "species": {  
        "values": 2,  
        "stringOffsets": 3  
      },  
      "age": {  
        "values": 1  
      },  
      "height": {  
        "values": 0  
      },  
      // "diameter" is not required and has been omitted from this table.  
    }  
  }]  
}
```

Each buffer view `byteOffset` shall be aligned to a multiple of the size of the `componentType` of the respective property.

**NOTE 2**Authoring tools may choose to align all buffer views to 8-byte boundaries for consistency, but client implementations should only depend on 8-byte alignment for buffer views containing 64-bit component types.

For the arrayOffsets and stringOffsets buffer views, the property can also define the arrayOffsetType and stringOffsetType, which describe the storage type for array- and string offsets, respectively. Allowed types are UINT8, UINT16, UINT32, and UINT64. The default is UINT32.

A property may override the minimum and maximum values and the offset and scale from the property definition in the class, to account for the actual range of values that is stored in the property table.

## E.2. Schema Implementation

---

The 3D Metadata Specification defines schemas as a description of the structure of metadata, consisting of classes with different properties, and enum types. The following is the description of a JSON-based representation of such a schema and its elements. It is used as the basis for defining the metadata structure in the following implementations:

- 3D Tiles Metadata Implicit Tilesets – Assigns metadata to tilesets, tiles, groups, and contents in a 3D Tiles tileset. A property table is defined for subtrees of an implicit tile hierarchy, and stores metadata that is associated with the nodes of such a subtree.
- EXT\_structural\_metadata – Assigns metadata to vertices, texels, and features in a glTF asset. A property table is defined in the top-level extension object. The property values are stored in standard glTF buffer views.

The full JSON schema definition for this implementation can be found in [the Schema directory of the specification](#).

### E.2.1. Schema

Defined in `schema.schema.json`.

A schema defines a set of classes and enums. Classes serve as templates for entities. They provide a list of properties and the type information for those properties. Enums define the allowable values for enum properties.

**NOTE** Schema with a tree class, and a speciesEnum enum that defines different species of trees. Later examples show how these structures in more detail.

```
{  
  "schema": {  
    "classes": {  
      "tree": { ... },  
      "enums": {  
        "speciesEnum": { ... }  
      }  
    }  
  }  
}
```

### E.2.1.1. Class

Defined in *class.schema.json*.

A class is a template for metadata entities. Classes provide a list of property definitions. Every entity shall be associated with a class, and the entity's properties shall conform to the class's property definitions. Entities whose properties conform to a class are considered instances of that class.

Classes are defined as entries in the `schema.classes` dictionary, indexed by class ID. Class IDs shall be identifiers as defined in the 3D Metadata Specification.

**NOTE** A “Tree” class, which might describe a table of tree measurements taken in a park. Property definitions are abbreviated here, and introduced in the next section.

```
{  
  "schema": {  
    "classes": {  
      "tree": {  
        "name": "Tree",  
        "description": "Woody, perennial plant.",  
        "properties": {  
          "species": { ... },  
          "age": { ... },  
          "height": { ... },  
          "diameter": { ... }  
        }  
      }  
    }  
  }  
}
```

### E.2.1.2. Class Property

Defined in *class.property.schema.json*.

Class properties are defined abstractly in a class. The class is instantiated with specific values conforming to these properties. Class properties support a rich variety of data types. Details about the supported types can be found in the 3D Metadata Specification.

Class properties are defined as entries in the `class.properties` dictionary, indexed by property ID. Property IDs shall be identifiers as defined in the 3D Metadata Specification.

**NOTE** A “Tree” class, which might describe a table of tree measurements taken in a park. Properties include species, age, height, and diameter of each tree.

```
{  
  "schema": {  
    "classes": {  
      "tree": {  
        "name": "Tree",  
        "description": "Woody, perennial plant.",  
        "properties": {  
          "species": {  
            "description": "Type of tree.",  
            "type": "ENUM",  
            "enumType": "speciesEnum",  
            "required": true  
          }  
        }  
      }  
    }  
  }  
}
```

```
        },
        "age": {
            "description": "The age of the tree, in years",
            "type": "SCALAR",
            "componentType": "UINT8",
            "required": true
        },
        "height": {
            "description": "Height of tree measured from ground level, in meters.",
            "type": "SCALAR",
            "componentType": "FLOAT32"
        },
        "diameter": {
            "description": "Diameter at trunk base, in meters.",
            "type": "SCALAR",
            "componentType": "FLOAT32"
        }
    }
}
```

### E.2.1.3. Enum

Defined in *enum.schema.json*.

A set of categorical types, defined as (name, value) pairs. Enum properties use an enum as their type.

Enums are defined as entries in the schema.`.enums` dictionary, indexed by an enum ID. Enum IDs shall be identifiers as defined in the 3D Metadata Specification.

**NOTE** A “Species” enum defining types of trees. An “Unspecified” enum value is optional, but when provided as the `noData` value for a property (see: 3D Metadata – No Data Values) may be helpful to identify missing data.

```
{  
  "schema": {  
    "enums": {  
      "speciesEnum": {  
        "name": "Species",  
        "description": "An example enum for tree species.",  
        "values": [  
          {"name": "Unspecified", "value": 0},  
          {"name": "Oak", "value": 1},  
          {"name": "Pine", "value": 2},  
          {"name": "Maple", "value": 3}  
        ]  
      }  
    }  
  }  
}
```

#### E.2.1.4. Enum Value

Defined in *enum.value.schema.json*.

Pairs of (name, value) entries representing possible values of an enum property.

Enum values are defined as entries in the enum.values array. Duplicate names or duplicate integer values are not allowed.



F

# ANNEX F (INFORMATIVE) 3D METADATA SEMANTIC REFERENCE

---

# ANNEX F (INFORMATIVE) 3D METADATA SEMANTIC REFERENCE

## F.1. Overview

This document provides common definitions of meaning (“semantics”) used by metadata properties in 3D Tiles and glTF. Tileset authors may define their own application- or domain-specific semantics separately.

Semantics describe how properties should be interpreted. For example, an application that encounters the ID or NAME semantics while parsing a dataset may use these values as unique identifiers or human-readable labels, respectively.

Each semantic is defined in terms of its meaning, and the datatypes it may assume. Datatype specifications include “type” as defined by the 3D Metadata Specification. When applicable they may also include “component type”, “array”, and “count”.

For use of semantics, see:

- 3D Tiles Metadata – Assigns metadata to tilesets, tiles, and contents in 3D Tiles 1.1
- 3DTILES\_metadata – An extension for 3D Tiles 1.0 that assigns metadata to tilesets, tiles, and contents
- EXT\_structural\_metadata (glTF 2.0) – Assigns metadata to vertices, texels, and features in a glTF asset

## F.2. General

### F.2.1. Overview

Throughout this section, the term “entity” refers to any conceptual object with which a property value (as defined in the 3D Metadata Specification) may be associated. Examples of entities

include tilesets, tiles, and tile contents in 3D Tiles, or groups of vertices and texels in glTF 2.0 assets. Additional types of entities may be defined by other specifications or applications.

## F.2.2. General Semantics

**Table F.1 – General metadata semantics**

SEMANTIC	TYPE	DESCRIPTION
ID	<ul style="list-style-type: none"><li>Type: STRING</li></ul>	The unique identifier for the entity.
NAME	<ul style="list-style-type: none"><li>Type: STRING</li></ul>	The name of the entity. Names should be human-readable, and do not have to be unique.
DESCRIPTION	<ul style="list-style-type: none"><li>Type: STRING</li></ul>	Description of the entity. Typically at least a phrase, and possibly several sentences or paragraphs.
ATTRIBUTION_IDS	<ul style="list-style-type: none"><li>Type: SCALAR</li><li>Component type: UINT8, UINT16, UINT32, or UINT64</li><li>Array: true</li></ul>	List of attribution IDs that index into a global list of attribution strings. This semantic may be assigned to metadata at any level of granularity including tileset, group, subtree, tile, content, feature, vertex, and texel granularity. The global list of attribution strings is located in a tileset or subtree with the property semantic ATTRIBUTION_STRINGS. The following precedence order is used to locate the attribution strings: first the containing subtree (if applicable), then the containing external tileset (if applicable), and finally the root tileset.
ATTRIBUTION_STRINGS	<ul style="list-style-type: none"><li>Type: STRING</li><li>Array: true</li></ul>	List of attribution strings. Each string contains information about a data provider or copyright text. Text may include embedded markup languages such as HTML. This semantic may be assigned to metadata at any granularity (wherever STRING property values can be encoded). When used in combination with ATTRIBUTION_IDS it is assigned to subtrees and tilesets.

## F.3. 3D Tiles

---

### F.3.1. Overview

Semantics for 3D Tiles are assigned in relationship to a tileset, subtree, tile, or tile content, as defined by the 3D Tiles specification. When associated with other types of entities, these semantics may have invalid or undefined meanings.

Units for all linear distances are meters, and all angles are radians.

## F.3.2. Tileset

### F.3.2.1. Overview

TILESET\_\* semantics provide meaning for properties associated with a tileset.

### F.3.2.2. Tileset Semantics

**Table F.2 – Tileset metadata semantics**

SEMANTIC	TYPE	DESCRIPTION
TILESET_FEATURE_ID_LABELS	<ul style="list-style-type: none"><li>Type: STRING</li><li>Array: true</li></ul>	The union of all the feature ID labels in glTF content using the <a href="#">EXT_mesh_features</a> and <a href="#">EXT_instance_features</a> extensions.
TILESET_CRS_GEOCENTRIC	<ul style="list-style-type: none"><li>Type: STRING</li></ul>	<p>The geocentric coordinate reference system (CRS) of the tileset. Values include, but are not limited to:</p> <ul style="list-style-type: none"><li>"EPSG:4978" – WGS 84</li><li>"EPSG:7656" – WGS 84 (G730)</li><li>"EPSG:7658" – WGS 84 (G873)</li><li>"EPSG:7660" – WGS 84 (G1150)</li><li>"EPSG:7662" – WGS 84 (G1674)</li><li>"EPSG:7664" – WGS 84 (G1762)</li><li>"EPSG:9753" – WGS 84 (G2139)</li><li>"EPSG:7842" – GDA2020</li><li>"UNKNOWN" – CRS is unknown</li></ul> <p>region bounding volumes are assumed to use the same reference ellipsoid as the geocentric coordinate reference system specified here. For more details on coordinate reference systems in 3D Tiles, see Coordinate Reference System (CRS).</p>
TILESET_CRS_COORDINATE_EPOCH	<ul style="list-style-type: none"><li>Type: STRING</li></ul>	The coordinate epoch for coordinates that are referenced to a dynamic CRS such as WGS 84. Coordinates include glTF vertex positions after transforms have been applied – see glTF transforms. Expressed as a decimal year (e.g. "2019.81"). See <a href="#">WKT representation of coordinate epoch</a> and <a href="#">coordinate metadata</a> for more details.

### F.3.3. Tile

#### F.3.3.1. Overview

TILE\_\* semantics provide meaning for properties associated with a particular tile, and should take precedence over equivalent metadata on parent objects, as well as over values derived from subdivision schemes in Implicit Tiling.

If property values are missing, either because the property is omitted or the property table contains noData values, the original tile properties are used, such as those explicitly defined in tileset JSON or implicitly computed from subdivision schemes in Implicit Tiling.

In particular, TILE\_BOUNDING\_BOX, TILE\_BOUNDING\_REGION, and TILE\_BOUNDING\_SPHERE semantics each define a more specific bounding volume for a tile than is implicitly calculated from Implicit Tiling. If more than one of these semantics are available for a tile, clients may select the most appropriate option based on use case and performance requirements.

#### F.3.3.2. Tile Semantics

**Table F.3 – Tile metadata semantics**

SEMANTIC	TYPE	DESCRIPTION
TILE_BOUNDING_BOX	<ul style="list-style-type: none"><li>Type: SCALAR</li><li>Component type: FLOAT32 or FLOAT64</li><li>Array: true</li><li>Count: 12</li></ul>	The bounding volume of the tile, expressed as a box. Equivalent to <code>tile.boundingVolume.box</code> .
TILE_BOUNDING_REGION	<ul style="list-style-type: none"><li>Type: SCALAR</li><li>Component type: FLOAT64</li><li>Array: true</li><li>Count: 6</li></ul>	The bounding volume of the tile, expressed as a region. Equivalent to <code>tile.boundingVolume.region</code> .
TILE_BOUNDING_SPHERE	<ul style="list-style-type: none"><li>Type: SCALAR</li><li>Component type: FLOAT32 or FLOAT64</li><li>Array: true</li><li>Count: 4</li></ul>	The bounding volume of the tile, expressed as a sphere. Equivalent to <code>tile.boundingVolume.sphere</code> .
TILE_BOUNDING_S2_CELL	<ul style="list-style-type: none"><li>Type: SCALAR</li><li>Component type: UINT64</li></ul>	The bounding volume of the tile, expressed as an <a href="#">S2 Cell ID</a> using the 64-bit representation instead of the hexadecimal representation. Only applicable to <a href="#">3DTILES_bounding_volume_S2</a> .

SEMANTIC	TYPE	DESCRIPTION
TILE_MINIMUM_HEIGHT	<ul style="list-style-type: none"> <li>Type: SCALAR</li> <li>Component type: FLOAT32 or FLOAT64</li> </ul>	The minimum height of the tile above (or below) the WGS84 ellipsoid. Equivalent to minimum height component of TILE_BOUNDING_REGION and tile.boundingVolume.region. Also equivalent to minimum height component of <u>3DTILES_bounding_volume_S2</u> .
TILE_MAXIMUM_HEIGHT	<ul style="list-style-type: none"> <li>Type: SCALAR</li> <li>Component type: FLOAT32 or FLOAT64</li> </ul>	The maximum height of the tile above (or below) the WGS84 ellipsoid. Equivalent to maximum height component of TILE_BOUNDING_REGION and tile.boundingVolume.region. Also equivalent to maximum height component of <u>3DTILES_bounding_volume_S2</u> .
TILE_HORIZON_OCCLUSION_POINT <sup>1</sup>	<ul style="list-style-type: none"> <li>Type: VEC3</li> <li>Component type: FLOAT32 or FLOAT64</li> </ul>	The horizon occlusion point of the tile expressed in an ellipsoid-scaled fixed frame. If this point is below the horizon, the entire tile is below the horizon. See <u>Horizon Culling</u> for more information.
TILE_GEOMETRIC_ERROR	<ul style="list-style-type: none"> <li>Type: SCALAR</li> <li>Component type: FLOAT32 or FLOAT64</li> </ul>	The geometric error of the tile. Equivalent to tile.geometricError.
TILE_REFINE	<ul style="list-style-type: none"> <li>Type: SCALAR</li> <li>Component type: UINT8</li> </ul>	The tile refinement type. Valid values are 0 ("ADD") and 1 ("REPLACE"). Equivalent to tile.refine.
TILE_TRANSFORM	<ul style="list-style-type: none"> <li>Type: MAT4</li> <li>Component type: FLOAT32 or FLOAT64</li> </ul>	The tile transform. Equivalent to tile.transform.

<sup>1</sup> TILE\_HORIZON\_OCCLUSION\_POINT should account for all content in a tile and its descendants, whereas CONTENT\_HORIZON\_OCCLUSION\_POINT should only account for content in a tile. When the two values are equivalent, only TILE\_HORIZON\_OCCLUSION\_POINT should be specified.

## F.3.4. Content

### F.3.4.1. Overview

CONTENT\_\* semantics provide meaning for properties associated with the content of a tile, and may be more specific to that content than properties of the containing tile.

CONTENT\_BOUNDING\_BOX, CONTENT\_BOUNDING\_REGION, and CONTENT\_BOUNDING\_SPHERE semantics each define a more specific bounding volume for tile contents than the bounding volume of the tile. If more than one of these semantics are available for the same content, clients may select the most appropriate option based on use case and performance requirements.

### F.3.4.2. Content Semantics

**Table F.4 – Content metadata semantics**

SEMANTIC	TYPE	DESCRIPTION
CONTENT_BOUNDING_BOX	<ul style="list-style-type: none"> <li>Type: SCALAR</li> <li>Component type: FLOAT32 or FLOAT64</li> <li>Array: true</li> <li>Count: 12</li> </ul>	The bounding volume of the content of a tile, expressed as a box. Equivalent to <code>tile.content.boundingVolume.box</code> .
CONTENT_BOUNDING_REGION	<ul style="list-style-type: none"> <li>Type: SCALAR</li> <li>Component type: FLOAT64</li> <li>Array: true</li> <li>Count: 6</li> </ul>	The bounding volume of the content of a tile, expressed as a region. Equivalent to <code>tile.content.boundingVolume.region</code> .
CONTENT_BOUNDING_SPHERE	<ul style="list-style-type: none"> <li>Type: SCALAR</li> <li>Component type: FLOAT32 or FLOAT64</li> <li>Array: true</li> <li>Count: 4</li> </ul>	The bounding volume of the content of a tile, expressed as a sphere. Equivalent to <code>tile.content.boundingVolume.sphere</code> .
CONTENT_BOUNDING_S2_CELL	<ul style="list-style-type: none"> <li>Type: SCALAR</li> <li>Component type: UINT64</li> </ul>	The bounding volume of the content of a tile, expressed as an <a href="#">S2 Cell ID</a> using the 64-bit representation instead of the hexadecimal representation. Only applicable to <a href="#">3DTILES_bounding_volume_S2</a> .
CONTENT_MINIMUM_HEIGHT	<ul style="list-style-type: none"> <li>Type: SCALAR</li> <li>Component type: FLOAT32 or FLOAT64</li> </ul>	The minimum height of the content of a tile above (or below) the WGS84 ellipsoid. Equivalent to minimum height component of <code>CONTENT_BOUNDING_REGION</code> and <code>tile.content.boundingVolume.region</code> . Also equivalent to minimum height component of <a href="#">3DTILES_bounding_volume_S2</a> .
CONTENT_MAXIMUM_HEIGHT	<ul style="list-style-type: none"> <li>Type: SCALAR</li> <li>Component type: FLOAT32 or FLOAT64</li> </ul>	The maximum height of the content of a tile above (or below) the WGS84 ellipsoid. Equivalent to maximum height component of <code>CONTENT_BOUNDING_REGION</code> and <code>tile.content.boundingVolume.region</code> . Also equivalent to maximum height component of <a href="#">3DTILES_bounding_volume_S2</a> .
CONTENT_HORIZON_OCCLUSION_POINT <sup>1</sup>	<ul style="list-style-type: none"> <li>Type: VEC3</li> <li>Component type: FLOAT32 or FLOAT64</li> </ul>	The horizon occlusion point of the content of a tile expressed in an ellipsoid-scaled fixed frame. If this point is below the horizon, the entire content is below the horizon. See <a href="#">Horizon Culling</a> for more information.
CONTENT_URI	<ul style="list-style-type: none"> <li>Type: STRING</li> </ul>	The content uri. Overrides the implicit tile's generated content uri. Equivalent to <code>tile.content.uri</code> .

SEMANTIC	TYPE	DESCRIPTION
CONTENT_GROUP_ID	<ul style="list-style-type: none"> <li>Type: SCALAR</li> <li>Component type: UINT8, UINT16, UINT32, or UINT64</li> </ul>	The content group ID. Equivalent to <code>tile.content.group</code> .

<sup>1</sup>`TILE_HORIZON_OCCLUSION_POINT` should account for all content in a tile and its descendants, whereas `CONTENT_HORIZON_OCCLUSION_POINT` should only account for content in a tile. When the two values are equivalent, only `TILE_HORIZON_OCCLUSION_POINT` should be specified.



G

# ANNEX G (INFORMATIVE) CONTRIBUTORS

---

## ANNEX G (INFORMATIVE) CONTRIBUTORS

---

Editors:

- Patrick Cozzi, [@pjcozzi](#), [patrick@cesium.com](mailto:patrick@cesium.com)
- Sean Lilley, [@lilleyse](#), [sean@cesium.com](mailto:sean@cesium.com)

Acknowledgements:

- Gabby Getz, [@gabbygetz](#)
- Matt Amato, [@matt\\_amato](#)
- Erik Andersson, [@e-andersson](#)
- Dan Bagnell, [@bagnell](#)
- Ray Bentley
- Jannes Bolling, [@jbo023](#)
- Johan Borg
- Dylan Brown, [@Dylan-Brown](#)
- Sarah Chow, [cesium.com/team/SarahChow](#)
- Paul Connelly
- Volker Coors
- Erik Dahlström, [@erikdahlstrom](#)
- Tom Fili, [@CesiumFili](#)
- Leesa Fini, [@LeesaFini](#)
- Peter Gagliardi, [@ptrgags](#)
- Ralf Gutbell
- Frederic Houbie

- Marco Hutter, [@javagl](#)
- Daniel Krupka, [@krupkad](#)
- Ian Lilley, [@IanLilleyT](#)
- Don McCurdy, [@donmccurdy](#)
- Christopher Mitchell, Ph.D., [@KermMartian](#)
- Shehzan Mohammed, [@shehzan10](#)
- Claus Nagel
- Jean-Philippe Pons
- Carl Reed
- Kevin Ring, [www.kotachrome.com/kevin](http://www.kotachrome.com/kevin)
- Scott Simmons
- Sam Suhag, [@sanjeetsuhag](#)
- Rob Taglang, [@lasalvavida](#)
- Stan Tillman
- Piero Toffanin, [@pierotofy](#)
- Bao Tran, [@baothientran](#)
- Samuel Vargas, [@samulus](#)
- Pano Voudouris
- Dave Wesloh



H

# ANNEX H (INFORMATIVE) REVISION HISTORY

---

**H**

## ANNEX H (INFORMATIVE) REVISION HISTORY

---

DATE	RELEASE	EDITOR	PRIMARY CLAUSES MODIFIED	DESCRIPTION
2022-07-26	1.1.0	Sean Lilley	all	This is the first normative version of this document.
2022-08-17	1.1.0	Sean Lilley		Added explanatory notes about choice of radians, bounding box representation, and column-major order for matrices. Fixed formatting issues and typos.
2022-10-07	1.1.0	Sean Lilley	Terms and Definitions	Added note that in the context of 3D Tiles the term “metadata” is equivalent to the concept of attribution in other OGC standards.
2023-01-09	1.1.0	Sean Lilley	Document Metadata	Updated approval date and public date.