

3D TILES SPECIFICATION 1.0

COMMUNITY STANDARD

APPROVED

Version: 1.0

Submission Date: 2018-06-04

Approval Date: 2018-12-14

Publication Date: 2019-01-31

Editor: Patrick Cozzi, Sean Lilley, Gabby Getz

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

Permission is hereby granted by the Open Geospatial Consortium, ("Licensor"), free of charge and subject to the terms set forth below, to any person obtaining a copy of this Intellectual Property and any associated documentation, to deal in the Intellectual Property without restriction (except as set forth below), including without limitation the rights to implement, use, copy, modify, merge, publish, distribute, and/or sublicense copies of the Intellectual Property, and to permit persons to whom the Intellectual Property is furnished to do so, provided that all copyright notices on the intellectual property are retained intact and that each person to whom the Intellectual Property is furnished agrees to the terms of this Agreement.

If you modify the Intellectual Property, all copies of the modified Intellectual Property must include, in addition to the above copyright notice, a notice that the Intellectual Property includes modifications that have not been approved or adopted by LICENSOR.

THIS LICENSE IS A COPYRIGHT LICENSE ONLY, AND DOES NOT CONVEY ANY RIGHTS UNDER ANY PATENTS THAT MAY BE IN FORCE ANYWHERE IN THE WORLD. THE INTELLECTUAL PROPERTY IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE DO NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE INTELLECTUAL PROPERTY WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE INTELLECTUAL PROPERTY WILL BE UNINTERRUPTED OR ERROR FREE. ANY USE OF THE INTELLECTUAL PROPERTY SHALL BE MADE ENTIRELY AT THE USER'S OWN RISK. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR ANY CONTRIBUTOR OF INTELLECTUAL PROPERTY RIGHTS TO THE INTELLECTUAL PROPERTY BE LIABLE FOR ANY CLAIM, OR ANY DIRECT, SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM ANY ALLEGED INFRINGEMENT OR ANY LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR UNDER ANY OTHER LEGAL THEORY, ARISING OUT OF OR IN CONNECTION WITH THE IMPLEMENTATION, USE, COMMERCIALIZATION OR PERFORMANCE OF THIS INTELLECTUAL PROPERTY.

This license is effective until terminated. You may terminate it at any time by destroying the Intellectual Property together with all copies in any form. The license will also terminate if you fail to comply with any term or condition of this Agreement. Except as provided in the following sentence, no such termination of this license shall require the termination of any third party end-user sublicense to the Intellectual Property which is in force as of the date of notice of such termination. In addition, should the Intellectual Property, or the operation of the Intellectual Property, infringe, or in LICENSOR's sole opinion be likely to infringe, any patent, copyright, trademark or other right of a third party, you agree that LICENSOR, in its sole discretion, may terminate this license without any compensation or liability to you, your licensees or any other party. You agree upon termination of any kind to destroy or cause to be destroyed the Intellectual Property together with all copies in any form, whether held by you or by any third party.

Except as contained in this notice, the name of LICENSOR or of any other holder of a copyright in all or part of the Intellectual Property shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Intellectual Property without prior written authorization of LICENSOR or such copyright holder. LICENSOR is and shall at all times be the sole entity that may authorize you or any third party to use certification marks, trademarks or other special designations to indicate compliance with any LICENSOR standards or specifications. This Agreement is governed by the laws of the Commonwealth of Massachusetts. The application to this Agreement of the United Nations Convention on Contracts for the International Sale of Goods is hereby expressly excluded. In the event any provision of this Agreement shall be deemed unenforceable, void or invalid, such provision shall be modified so as to make it valid and enforceable, and as so modified the entire Agreement shall remain in full force and effect. No decision, action or inaction by LICENSOR shall be construed to be a waiver of any rights or remedies available to it.

None of the Intellectual Property or underlying information or technology may be downloaded or otherwise exported or reexported in violation of U.S. export laws and regulations. In addition, you are responsible for complying with any local laws in your jurisdiction which may impact your right to import, export or use the Intellectual Property, and you represent that you have complied with any regulations or registration procedures required by applicable law to make this license enforceable.

Copyright notice

Copyright © 2016-2019 Open Geospatial Consortium
To obtain additional rights of use, visit <http://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. | PREFACE | vi |
| II. | SECURITY CONSIDERATIONS | vii |
| III. | SOURCE OF THE CONTENT FOR THIS OGC DOCUMENT | vii |
| IV. | VALIDITY OF CONTENT | vii |
| V. | FUTURE WORK | vii |
| 1. | SCOPE | 2 |
| 2. | CONFORMANCE | 4 |
| 3. | NORMATIVE REFERENCES | 154 |
| | 3.1. Normative | 154 |
| 4. | TERMS AND DEFINITIONS | 6 |
| 5. | CONVENTIONS | 10 |
| 6. | 3D TILES SPECIFICATION | 12 |
| | 6.1. Overview | 12 |
| | 6.2. File extensions and MIME types | 14 |
| | 6.3. JSON encoding | 14 |
| | 6.4. URLs | 14 |
| | 6.5. Units | 15 |
| | 6.6. Coordinate reference system (CRS) | 15 |
| | 6.7. Tiles | 15 |
| | 6.8. Tileset JSON | 34 |
| | 6.9. Specifying extensions and application specific extras | 41 |
| | 6.10. Tile format specifications | 43 |
| 7. | PROPERTY REFERENCE | 46 |
| | 7.1. Tileset | 46 |
| | 7.2. Asset | 48 |
| | 7.3. Bounding Volume | 49 |
| | 7.4. Extension | 51 |
| | 7.5. Extras | 52 |
| | 7.6. Properties | 52 |

| | |
|--|------------|
| 7.7. Tile | 53 |
| 7.8. Tile Content | 57 |
| 8. FEATURE TABLE | 60 |
| 8.1. Overview | 60 |
| 8.2. Layout | 60 |
| 8.3. Implementation example | 62 |
| 8.4. Property reference | 62 |
| 9. BATCH TABLE | 66 |
| 9.1. Overview | 66 |
| 9.2. Layout | 66 |
| 9.3. Implementation example | 69 |
| 9.4. Property reference | 70 |
| 10. TILE FORMAT SPECIFICATIONS | 75 |
| 10.1. Batched 3D Model | 75 |
| 10.2. Instanced 3D Model | 82 |
| 10.3. Point Cloud | 96 |
| 10.4. Composite | 112 |
| 11. DECLARATIVE STYLING SPECIFICATION | 116 |
| 11.1. Overview | 116 |
| 11.2. Concepts | 116 |
| 11.3. Expressions | 119 |
| 11.4. Point Cloud | 139 |
| 11.5. File extension and MIME type | 140 |
| 11.6. Property reference | 141 |
| ANNEX A (NORMATIVE) CONFORMANCE CLASS ABSTRACT TEST SUITE | 147 |
| ANNEX B (INFORMATIVE) CONTRIBUTOR ACKNOWLEDGEMENTS | 149 |
| ANNEX C (INFORMATIVE) REVISION HISTORY | 151 |

LIST OF FIGURES

| | |
|---|----|
| Figure 1 – A sample 3D Tiles bounding volume hierarchy | 13 |
| Figure 2 – A parent tile with replacement refinement | 17 |
| Figure 3 – A refined child tile of a tile with replacement refinement | 17 |
| Figure 4 – A parent tile with additive refinement | 18 |
| Figure 5 – A refined child tile of a tile with additive refinement | 19 |
| Figure 6 – A bounding box | 20 |
| Figure 7 – A bounding sphere | 21 |

| | |
|---|-----|
| Figure 8 – A bounding region | 22 |
| Figure 9 – A bounding region | 23 |
| Figure 10 – A bounding box | 24 |
| Figure 11 – A bounding sphere | 25 |
| Figure 12 – A tileset with transformed children tiles | 29 |
| Figure 13 – Tile JSON properties | 31 |
| Figure 14 – A tile bounding volume in red, and a content bounding volume in blue | 34 |
| Figure 15 – A tileset JSON file with external tileset JSON files | 37 |
| Figure 16 – A tileset with transforms referencing an external tileset with transforms | 38 |
| Figure 17 – A root tile and its four children tiles | 39 |
| Figure 18 – Two sibling tiles with overlapping bounding volumes | 40 |
| Figure 19 – A tileset with an overlapping grid spatial data structure | 41 |
| Figure 20 – Feature Table layout | 60 |
| Figure 21 – Feature Table binary body layout | 62 |
| Figure 22 – Batch Table layout | 66 |
| Figure 23 – Batch Table binary body layout | 68 |
| Figure 24 – Batched 3D Model layout | 76 |
| Figure 25 – Instanced 3D Model layout | 83 |
| Figure 26 – A box in the standard basis | 86 |
| Figure 27 – A box transformed into a rotated basis | 87 |
| Figure 28 – A quantized volume | 88 |
| Figure 29 – Point Cloud layout | 97 |
| Figure 30 – A quantized volume | 101 |
| Figure 31 – Composite layout | 112 |

PREFACE

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.

NOTE: This draft policy does not address the issue of how the OGC migrates from WKT for CRS version 2 –CRS2 – (ISO 19162 and OGC 12-063r5 Geographic information – Well-known text representation of coordinate reference systems) to any subsequent future versions.

If approved, the contents of this document regarding coordinate reference systems will be updated as needed to ensure compatibility.

SECURITY CONSIDERATIONS

No security considerations have been made for this document.

SOURCE OF THE CONTENT FOR THIS OGC DOCUMENT

The majority of the content in this OGC document is a direct copy of the content contained at <https://github.com/AnalyticalGraphicsInc/3d-tiles/releases/tag/1.0> (the 1.0 tag of the 3d-tiles repo). No normative changes have been made to the content. This OGC document does contain content not contained in the 1.0 tag of the 3d-tiles repo.

NOTE: Some elements (such as Vector Data) contained in <https://github.com/AnalyticalGraphicsInc/3d-tiles> (the 3d-tiles repo) have been removed from the OGC document because they are currently under development and not a part of this specification. These elements are identified as future work in this OGC document.

VALIDITY OF CONTENT

The Submission Team has reviewed and certified that the “snapshot” content in this Community Standard is true and accurate.

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.

Additions planned for future inclusion in the 3D Tiles Specification (future work) are described at <https://github.com/AnalyticalGraphicsInc/3d-tiles/issues/247>.

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.

A 3D Tiles data set, called a tileset, contains any combination of tile formats organized into a spatial data structure.

3D Tiles are declarative, extendable, and applicable to various types of 3D data. The following tile formats have been specified as part of this document:

- Batched 3D Model
- Instanced 3D Model
- Point Cloud
- Composite

This document also describes 3D Tile Styles, a declarative styling specification which may be applied to tilesets.

The 3D Tiles specification for tilesets, associated tile formats, and the associated styling specification are open formats that are not dependent on any vendor-specific solution, technology, or products.



2

CONFORMANCE

CONFORMANCE

Sections 7 through 11 of this document describe 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 documents located at <https://github.com/AnalyticalGraphicsInc/3d-tiles/releases/tag/1.0/specification/schema>

All figures, examples, notes, and background information are non-normative.

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. Bounding Volume

A closed volume completely containing the union of a set of geometric objects.¹

4.2. 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.

4.3. 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.

¹https://en.wikipedia.org/wiki/Bounding_volume

4.4. gITF

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

4.5. 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.²

4.6. 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.

4.7. 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, or Composite).

4.8. Tile Format

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

²https://en.wikipedia.org/wiki/Level_of_detail

4.9. 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.

4.10. 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.

4.11. Spatial Coherence

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

4.12. Style

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

5

CONVENTIONS

CONVENTIONS

No conventions are specified in this document.

6

3D TILES SPECIFICATION

3D TILES SPECIFICATION

6.1. Overview

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.

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 of one of the following formats:

| FORMAT | USES |
|------------------------------|---|
| Batched 3D Model (b3dm) | Heterogeneous 3D models. E.g. textured terrain and surfaces, 3D building exteriors and interiors, massive models. |
| Instanced 3D Model (i3dm) | 3D model instances. E.g. trees, windmills, bolts. |
| Point Cloud (pnts) | Massive number of points. |
| Composite (cmpt) | Concatenate tiles of different formats into one tile. |

A tile's *content*, an individual instance of a tile format, is a binary blob with format-specific components including a Feature Table and a Batch Table.

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 stored in the tile's Feature Table, and additional application-specific properties stored in the Batch Table. A client may choose to select features at runtime and retrieve their properties for visualization or analysis.

The Batched 3D Model and Instanced 3D Model formats are built on [glTF](#), an open specification designed for the efficient transmission of 3D content. The tile content of these formats embed a glTF asset, which contains model geometry and texture information, in the binary body. The Point Cloud format does not embed glTF.

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.

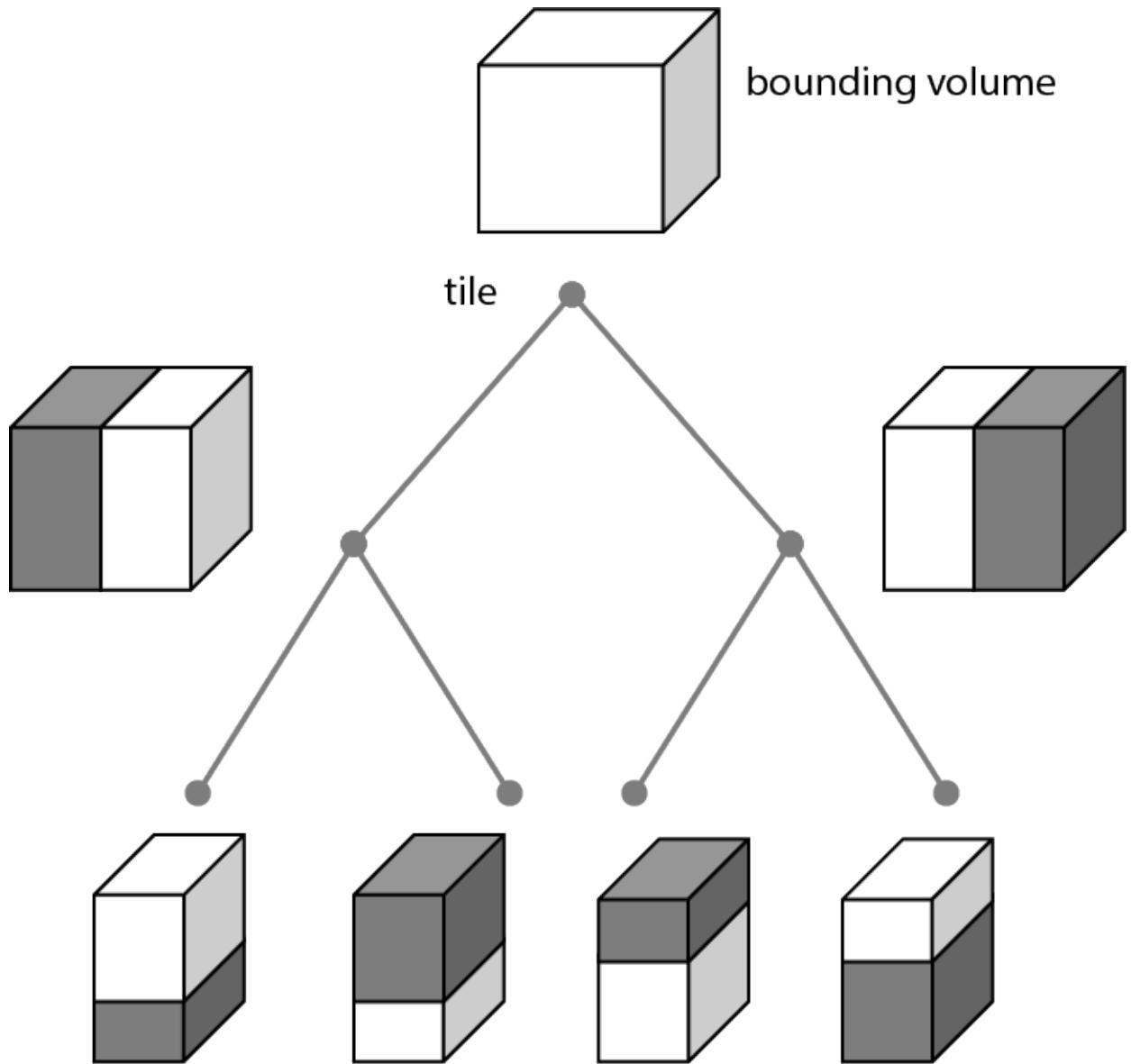


Figure 1 – A sample 3D Tiles bounding volume hierarchy

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.

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 MIME types

3D Tiles uses the following file extensions and MIME types.

- Tileset files use the .json extension and the application/json MIME type.
- Tile content files use the file type and MIME format specific to their tile format specification, see Tile format specifications.
- Tileset style files use the .json extension and the application/json MIME 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 must use UTF-8 encoding without BOM.
2. All strings defined in this spec (properties names, enums) use only ASCII charset and must be written as plain text.
3. Names (keys) within JSON objects must be unique, i.e., duplicate keys aren't allowed.

6.4. URIs

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” URI 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 must 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 4979](#)), 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.

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](#).

6.7. 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. 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. The root tile, being the most simplified version of the source geometry, will have the greatest geometric error. Then each successive level of children will have a lower 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, tile size in meters, or another factor specific to that tilesheet. In general, a higher geometric error means a tile will be refined more aggressively, and children tiles will be loaded and rendered sooner.

6.7.2. 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 tilesheet 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 tilesheet; it is optional for all other tiles. When omitted, a tile inherits the refinement type of its parent.

6.7.2.1. Replacement

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

Parent Tile

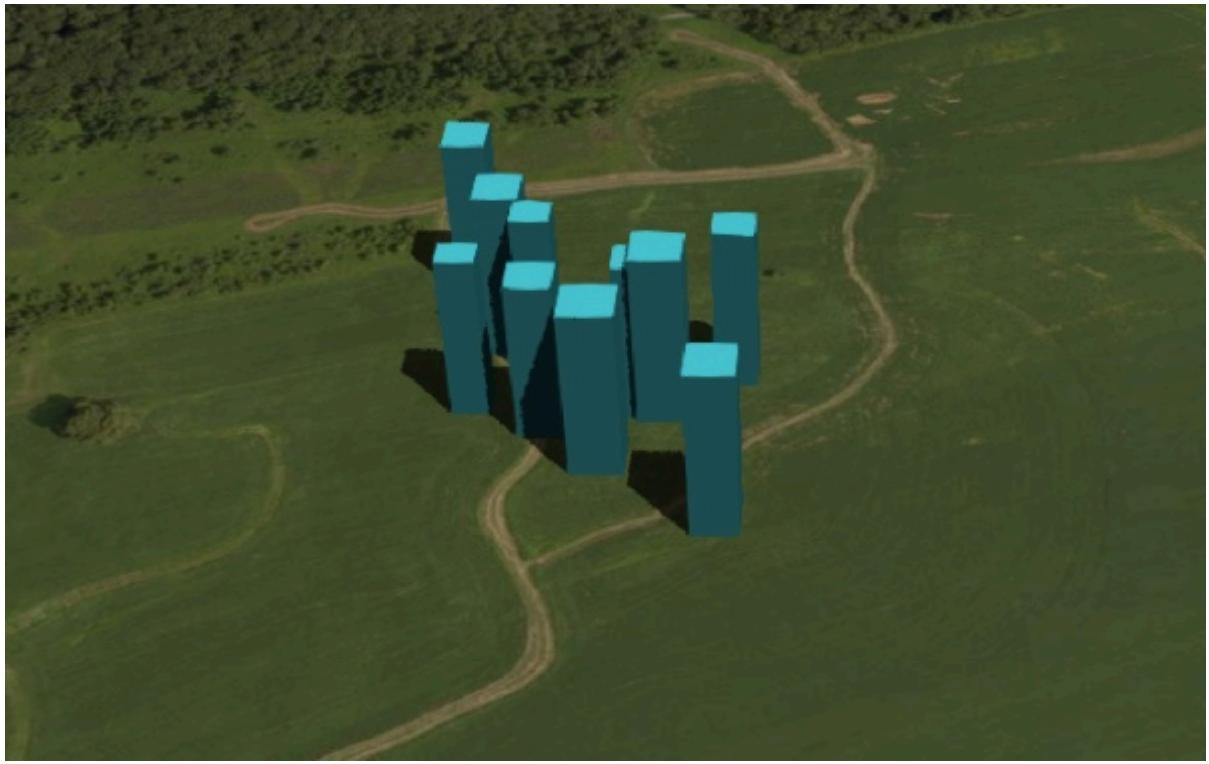


Figure 2 – A parent tile with replacement refinement

Refined



Figure 3 – A refined child tile of a tile with replacement refinement

6.7.2.2. Additive

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

Parent Tile

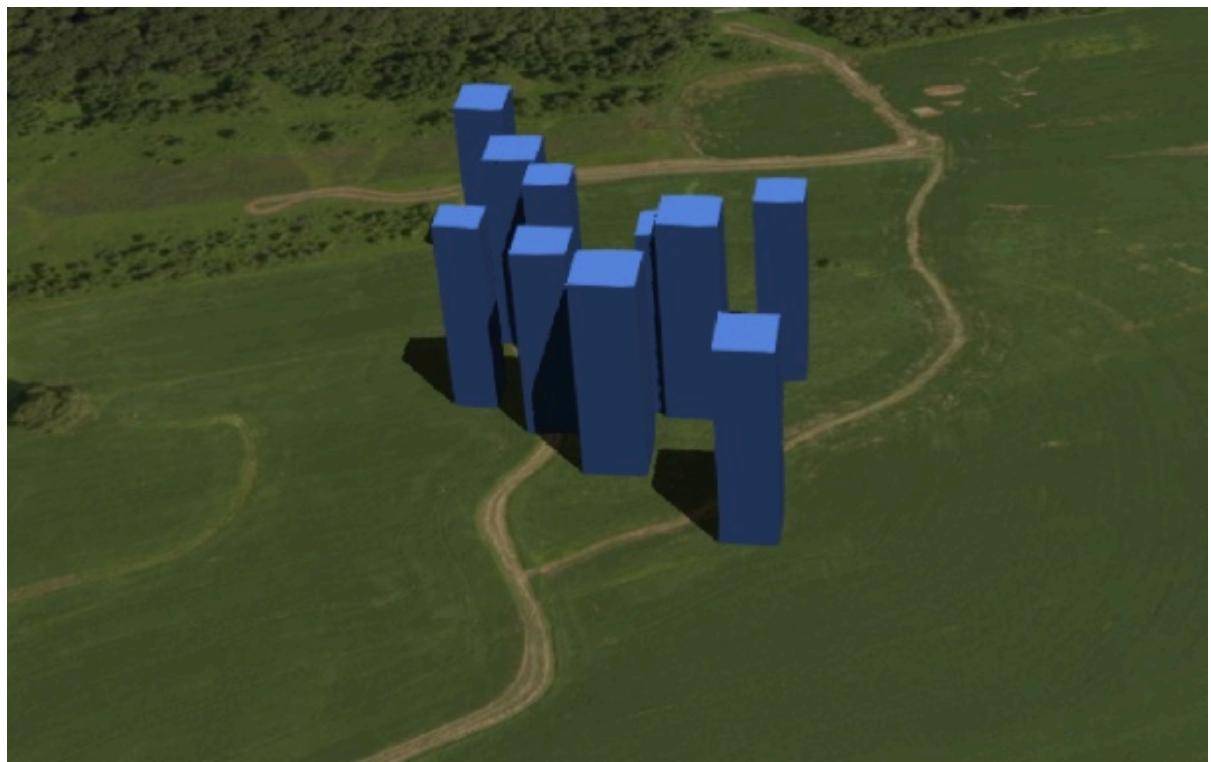


Figure 4 – A parent tile with additive refinement

Refined

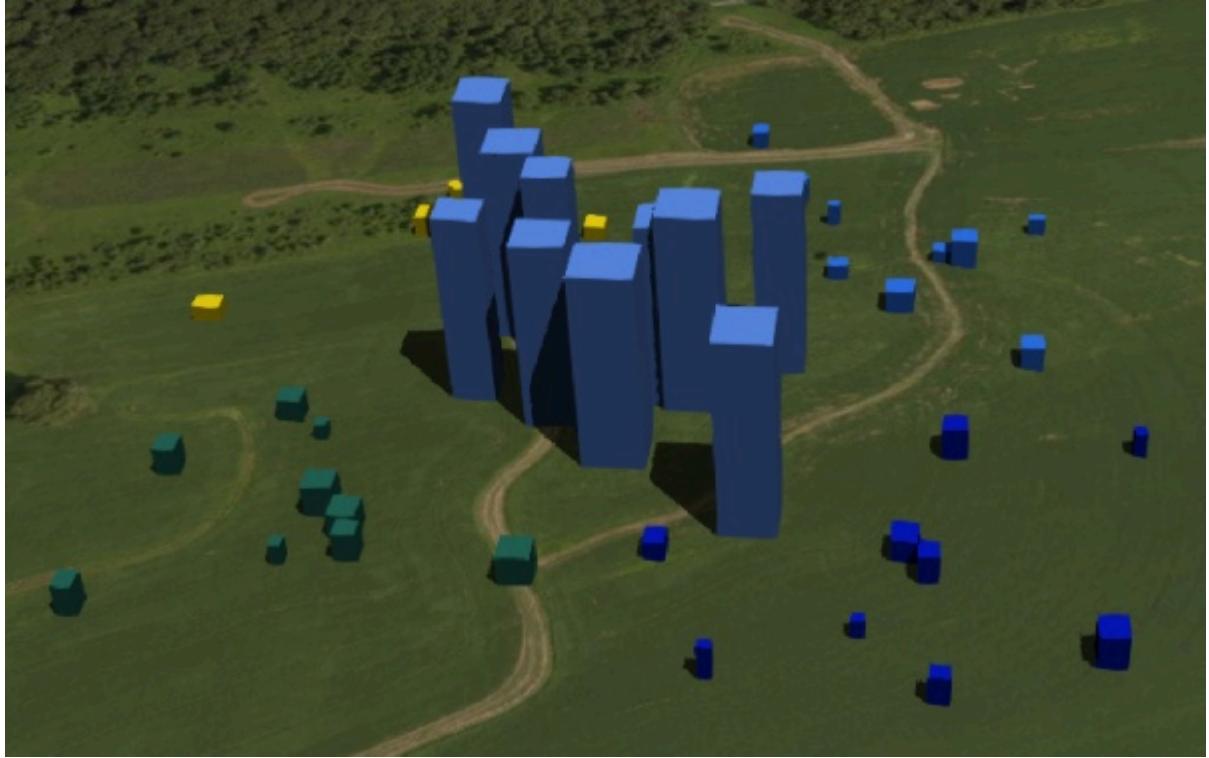


Figure 5 – A refined child tile of a tile with additive refinement

6.7.3. 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.

Bounding box

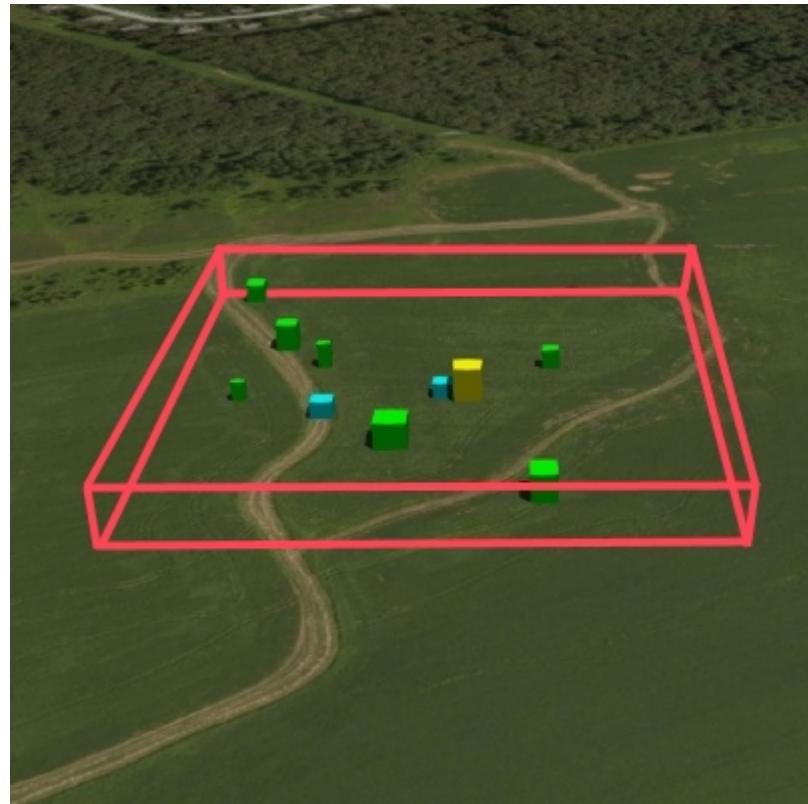


Figure 6 – A bounding box

Bounding sphere



Figure 7 – A bounding sphere

Bounding region



Figure 8 – A bounding region

6.7.3.1. 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](#).



Figure 9 – A bounding region

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

6.7.3.2. 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.

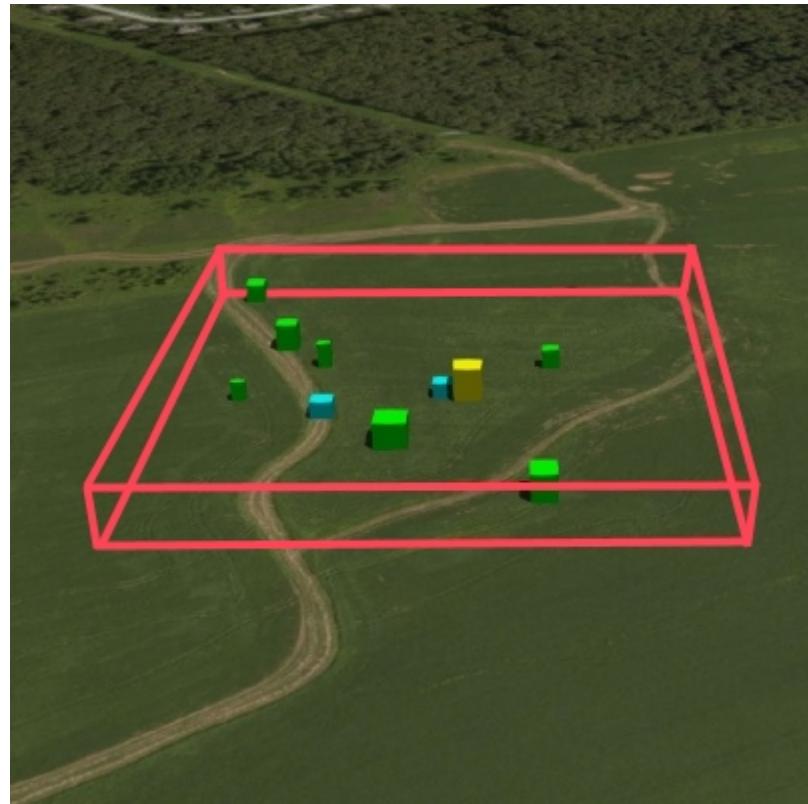


Figure 10 – A bounding box

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

6.7.3.3. 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.

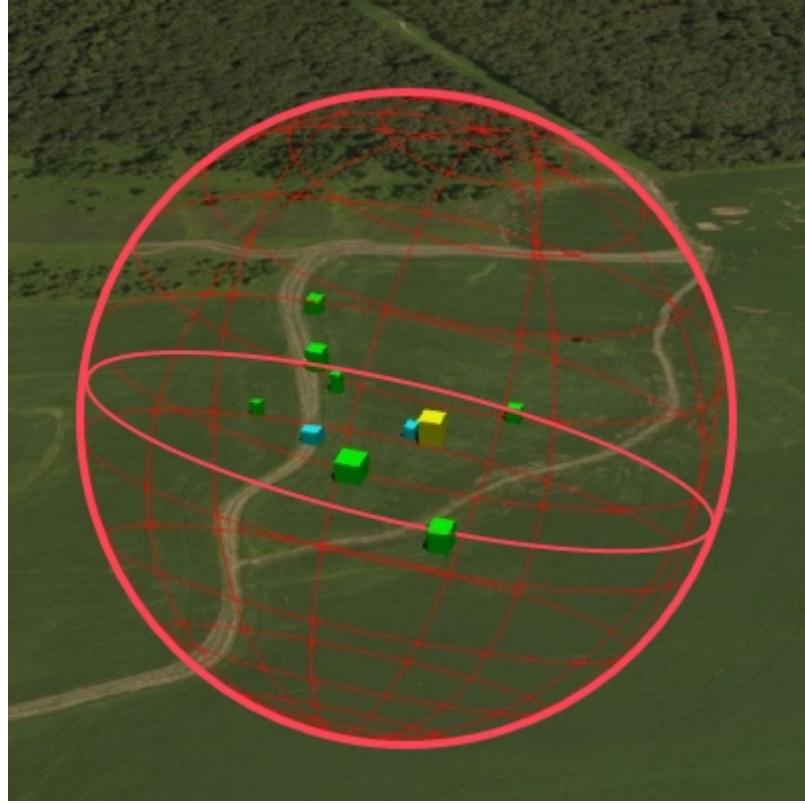


Figure 11 – A bounding sphere

```

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

6.7.4. 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 building in a b3dm tile and a point cloud inside the building in a pnts tile. 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.b3dm"
}
},
{
  "transform": [
    0.968635634376879, 0.24848542777253732, 0, 0,
    -0.15986650990768783, 0.6231850279035362, 0.7655606573007809, 0,
    0.19023066741520941, -0.7415493329385225, 0.6433637229384295, 0,
    1215002.0371330238, -4736270.772726648, 4081651.6414821907, 1
],
"viewerRequestVolume": {
  "sphere": [0, 0, 0, 15]
},
"boundingVolume": {
  "sphere": [0, 0, 0, 1.25]
},
"geometricError": 0,
"content": {
  "uri": "points.pnts"
}
}
}
]
}

```

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

6.7.5. Transforms

6.7.5.1. 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 4×4 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.

The `transform` property applies to:

- `tile.content`
- Each feature's position.
- Each feature's normal should be transformed by the top-left 3×3 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 does not apply to `geometricError` —i.e., the scale defined by `transform` does not scale the geometric error—the geometric error is always defined in meters.

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.

6.7.5.2. glTF transforms

Batched 3D Model and Instanced 3D Model tiles embed glTF, which 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.

6.7.5.2.1. glTF node hierarchy

First, glTF node hierarchy transforms are applied according to the [glTF specification](#).

6.7.5.2.2. y-up to z-up

Next, for consistency with the *z*-up coordinate system of 3D Tiles, glTFS must 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. glTF node hierarchy transformations
2. glTF y-up to z-up transform
3. Any tile format specific transforms.
 - Batched 3D Model Feature Table may define RTC_CENTER which is used to translate model vertices.
 - Instanced 3D Model Feature Table defines per-instance position, normals, and scales. These are used to create per-instance 4x4 affine transform matrices that are applied to each instance.
4. Tile transform

Implementation 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 glTF.
- At runtime the glTF is transformed back from *y-up* to *z-up* with the matrix above. Effectively the transforms cancel out.

Example glTF root node:

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

6.7.5.2.3. Example

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

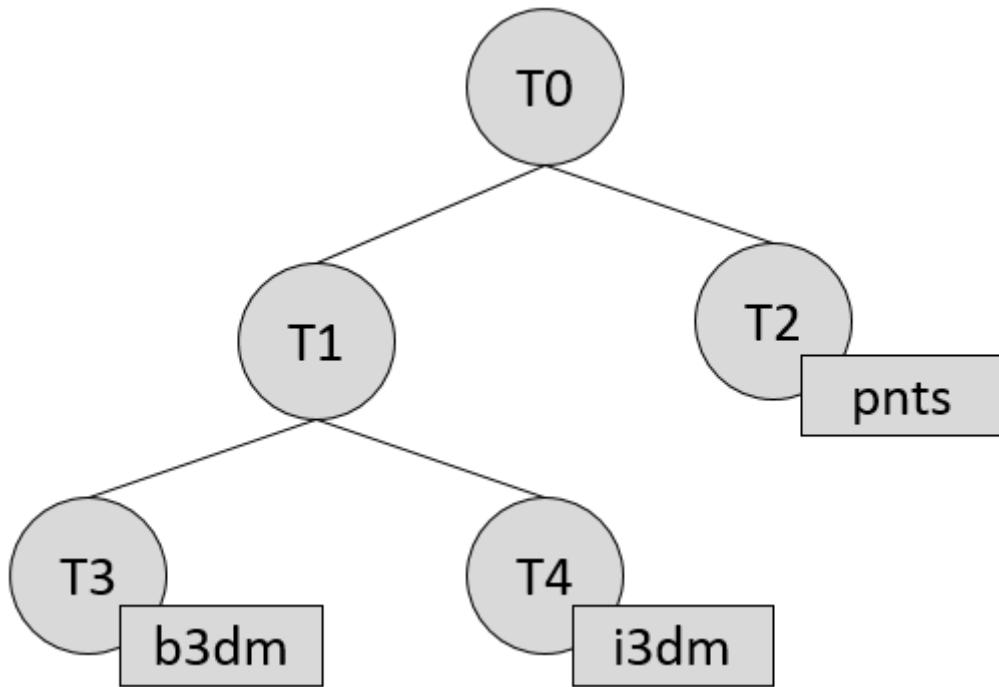


Figure 12 – A tileset with transformed children tiles

The computed transform for each tile is:

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

The positions and normals in a tile's content may also have tile-specific transformations applied to them *before* the tile's transform (before indicates post-multiplying for affine transformations). Some examples are:

- b3dm and i3dm tiles embed glTF, which defines its own node hierarchy and coordinate system. `tile.transform` is applied after these transforms are resolved. See glTF transforms.
- i3dm's Feature Table defines per-instance position, normals, and scales. These are used to create per-instance 4×4 affine transform matrices that are applied to each instance before `tile.transform`.
- Compressed attributes, such as POSITION_QUANTIZED in the Feature Tables for i3dm and pnts, and NORMAL_OCT16P in pnts should be decompressed before any other transforms.

Therefore, the full computed transforms for the above example are:

- T0: [T0]
- T1: [T0][T1]
- T2: [T0][T2][pnts-specific transform, including RTC_CENTER (if defined)]
- T3: [T0][T1][T3][b3dm-specific transform, including RTC_CENTER (if defined), coordinate system transform, and glTF node hierarchy]
- T4: [T0][T1][T4][i3dm-specific transform, including per-instance transform, coordinate system transform, and glTF node hierarchy]

6.7.5.2.4. Implementation example

This section is non-normative

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

```
function computeTransforms(tileset) {  
    var t = tileset.root;  
    var transformToRoot =defined(t.transform) ?Matrix4.fromArray(t.transform) :  
    Matrix4.IDENTITY;  
  
    computeTransform(t, transformToRoot);  
}  
  
function computeTransform(tile, transformToRoot) {  
    // Apply 4x4 transformToRoot to this tile's positions and bounding volumes  
  
    var inverseTransform =Matrix4.inverse(transformToRoot,*new*Matrix4());  
    var normalTransform =Matrix4.getRotation(inverseTransform,*new*Matrix3());  
    normalTransform =Matrix3.transpose(normalTransform, normalTransform);  
    // Apply 3x3 normalTransform to this tile's normals_  
  
    var children =tile.children;  
    var length =children.length;  
    for (var i =0; i < length; ++i) {  
        var child = children[i];  
        var childToRoot =defined(child.transform) ?Matrix4.fromArray(child.  
transform) : Matrix4.clone(Matrix4.IDENTITY);  
        childToRoot =Matrix4.multiplyTransformation(transformToRoot,  
childToRoot, childToRoot);  
        computeTransform(child, childToRoot);  
    }  
}
```

6.7.6. Tile JSON

A tile JSON object consists of the following properties.

tile

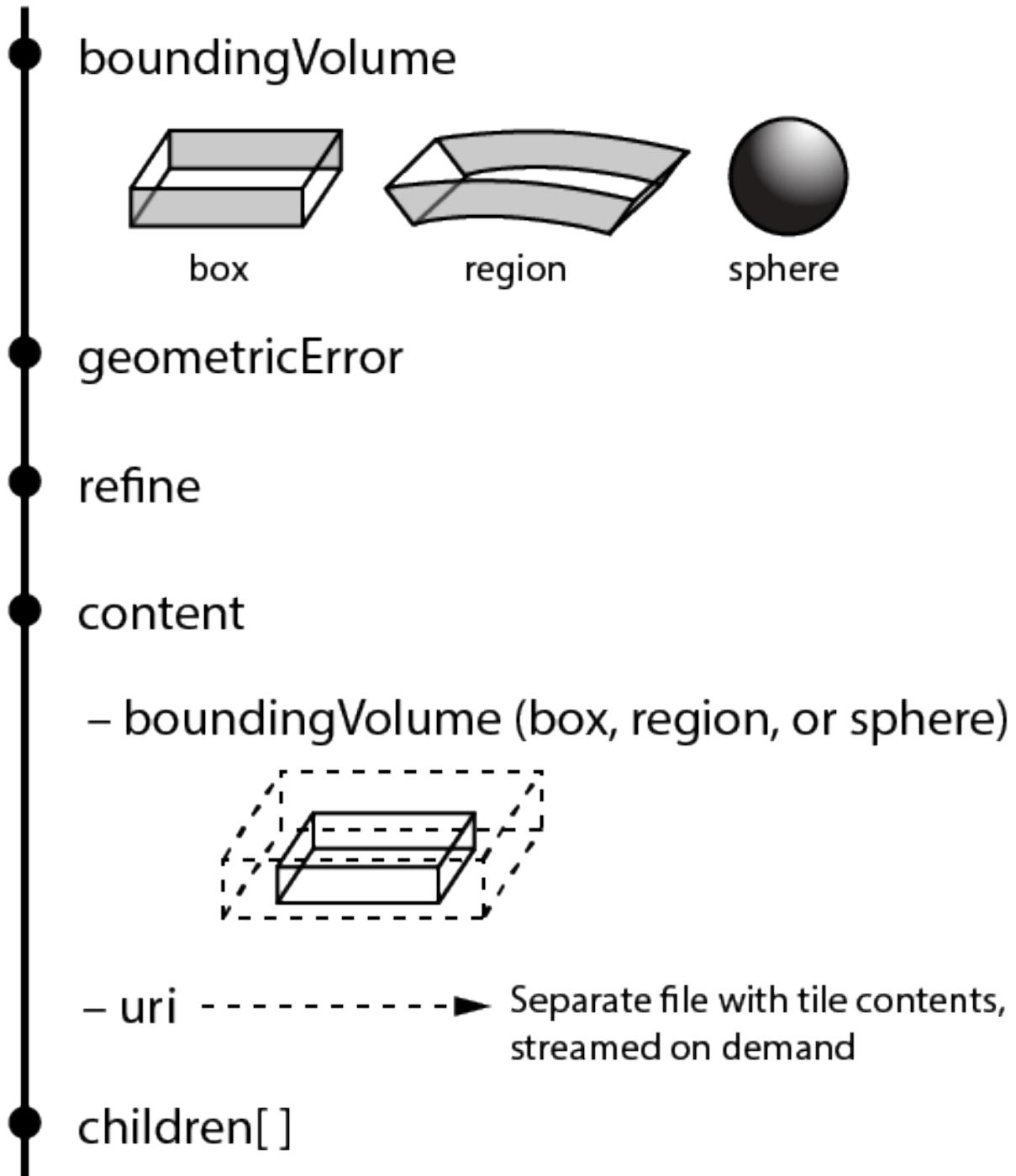


Figure 13 – Tile JSON properties

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.b3dm"
},
"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 [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](#).

Geometric error.

The optional `viewerRequestVolume` property (not shown above) defines a volume, using the same schema as `boundingVolume`, which the viewer must 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 [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. The root tile, being the most simplified version of the source geometry, will have the greatest geometric error. Then each successive level of children will have a lower 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, tile size in meters, 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.

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 contains metadata about the tile's renderable content. `content.uri` is a uri that points to the tile's binary content (see Tile format specifications), or another tileset JSON to create a tileset of tileset (see External tilesets).

A file extension is not required for `content.uri`. A content's tile format (see Tile format specifications) 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 `boundingVolume` property. But unlike the top-level `boundingVolume` property, `content.boundingVolume` is a tightly fit bounding volume enclosing just the tile's content. `boundingVolume` provides spatial coherence and `content.boundingVolume` 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](#). `boundingVolume`, shown in red, encloses the entire area of the tileset; `content.boundingVolume` shown in blue, encloses just the four features (models) in the root tile.

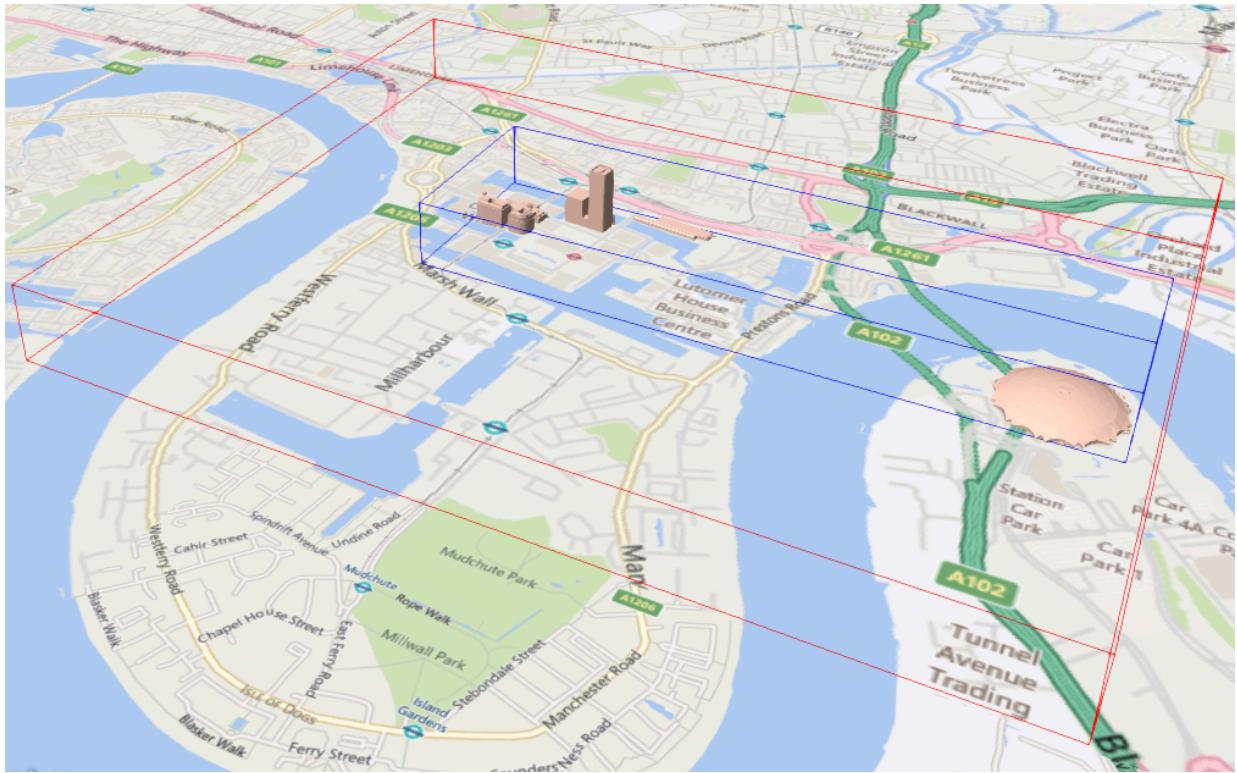


Figure 14 – A tile bounding volume in red, and a content bounding volume in blue

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

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.

6.8. 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](#) (also see the complete file, `tileset.json`):

```
{
  "asset": {
    "version": "1.0",
    "tilesetVersion": "e575c6f1-a45b-420a-b172-6449fa6e0a59",
  },
  "properties": {
    "Height": {
      "min": 0,
      "max": 100
    }
  }
}
```

```

        "minimum":1,
        "maximum":241.6
    }
},
"geometricError":494.50961650991815,
"root":{
    "boundingVolume":{
        "region":[
            -0.0005682966577418737,
            0.8987233516605286,
            0.00011646582098558159,
            0.8990603398325034,
            0,
            241.6
        ]
    },
    "geometricError":268.37878244706053,
    "refine":"ADD",
    "content":{
        "uri":"0/0/0.b3dm",
        "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.

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, when the tileset is not rendered. See 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.

Geometric error for how this value is used to drive refinement.

root is an object that defines the root tile using the JSON described in the above section. root.geometricError is not the same as the tileset's top-level geometricError. The tileset's geometricError is the error when the entire tileset is not rendered; root.geometricError is the error when only the root tile is rendered.

`root.children` 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.

6.8.1. 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.

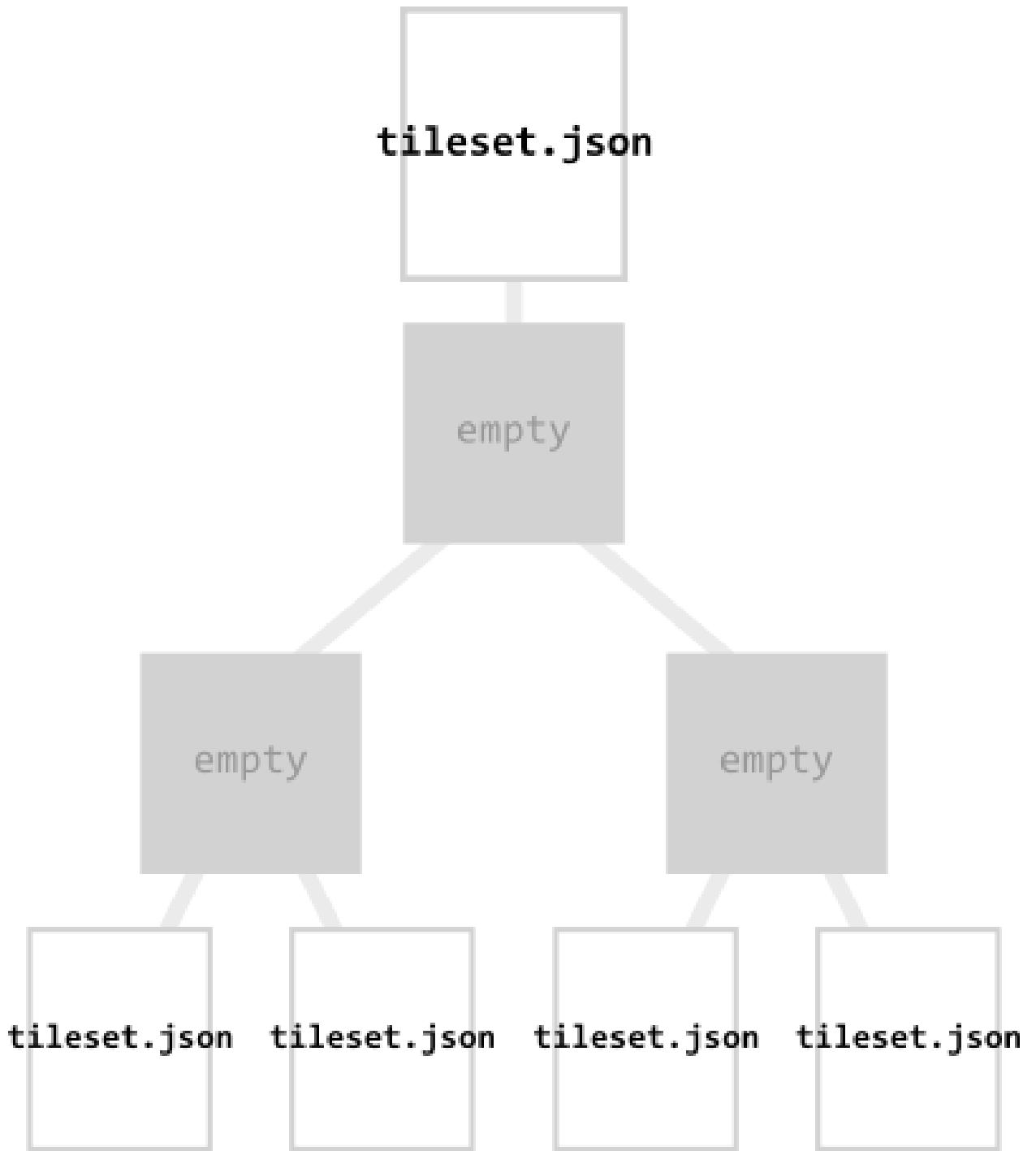


Figure 15 – A tileset JSON file with external tileset JSON files

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

- Cannot have any children; `tile.children` must be `undefined` or an empty array.
- 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 [T0][T1][T2][T3].

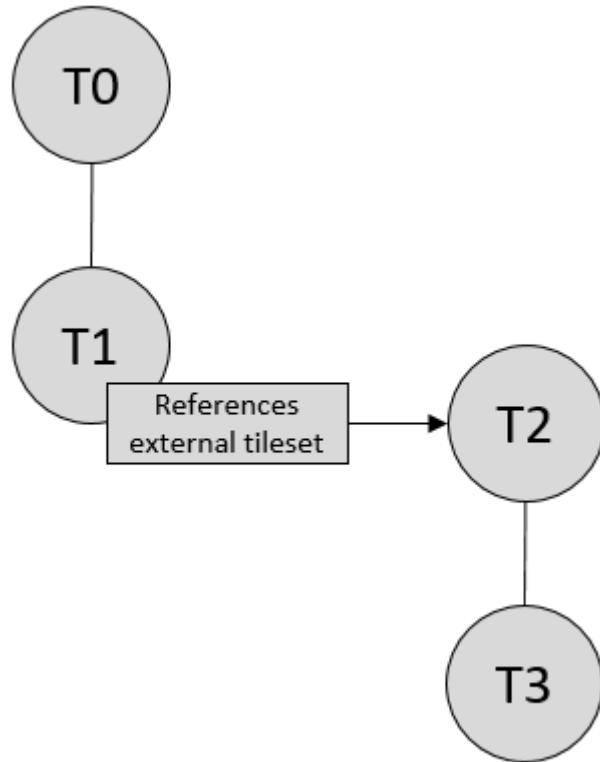


Figure 16 – A tileset with transforms referencing an external tileset with transforms

6.8.2. 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:

6.8.3. 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.

6.8.3.1. 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.

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).

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:

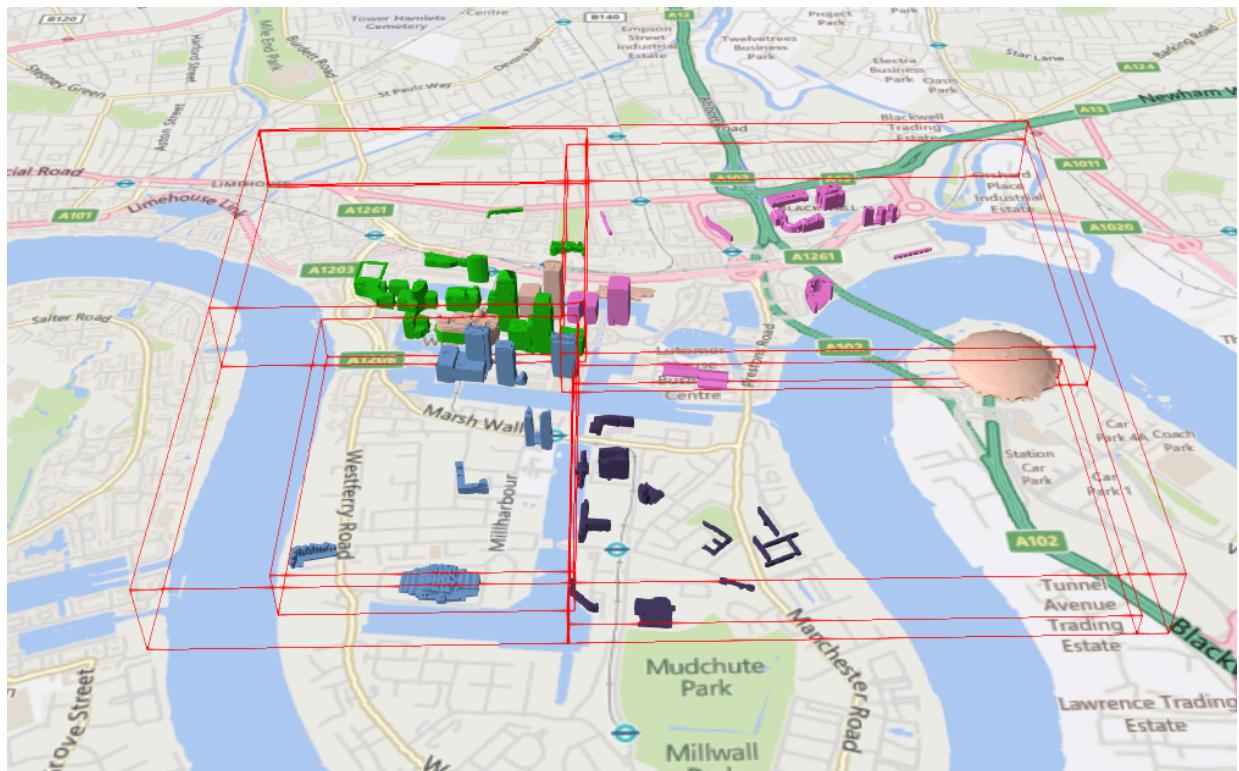


Figure 17 – A root tile and its four children tiles

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.

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.

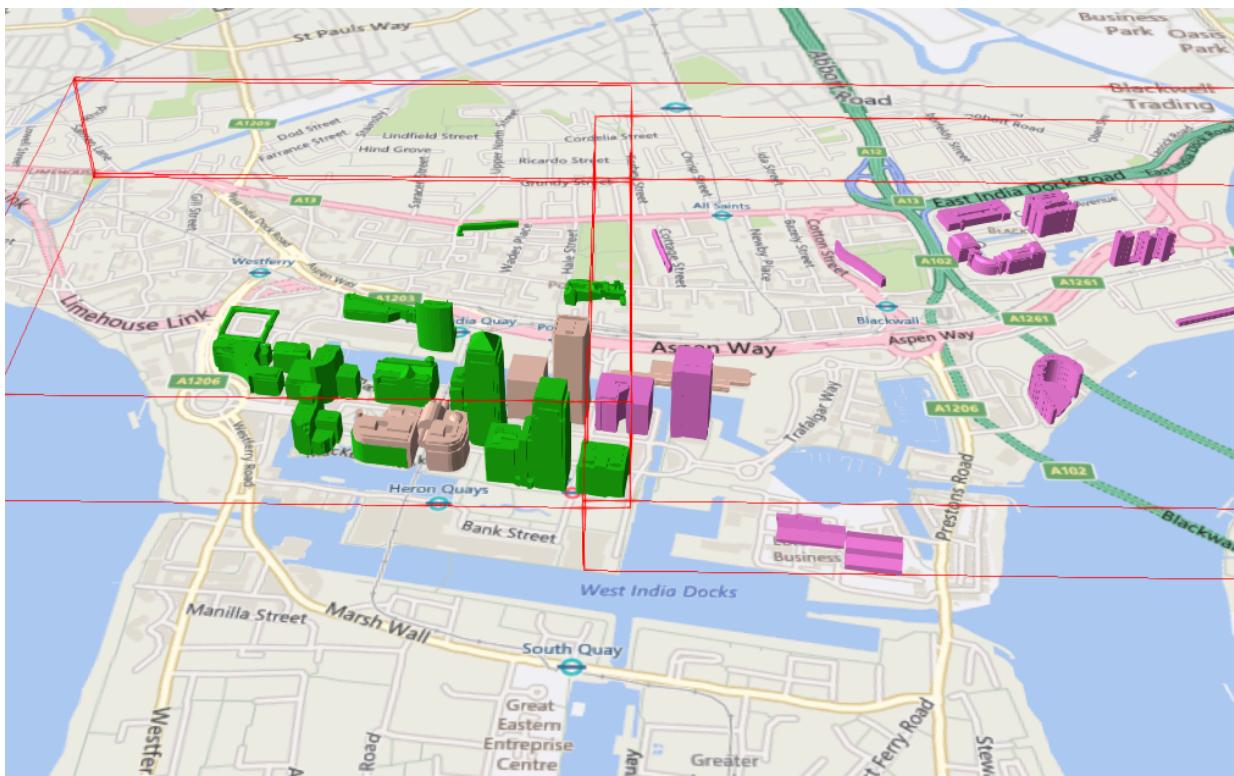


Figure 18 – Two sibling tiles with overlapping bounding volumes

6.8.3.2. 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.

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.

6.8.3.3. 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.

6.8.3.4. 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:

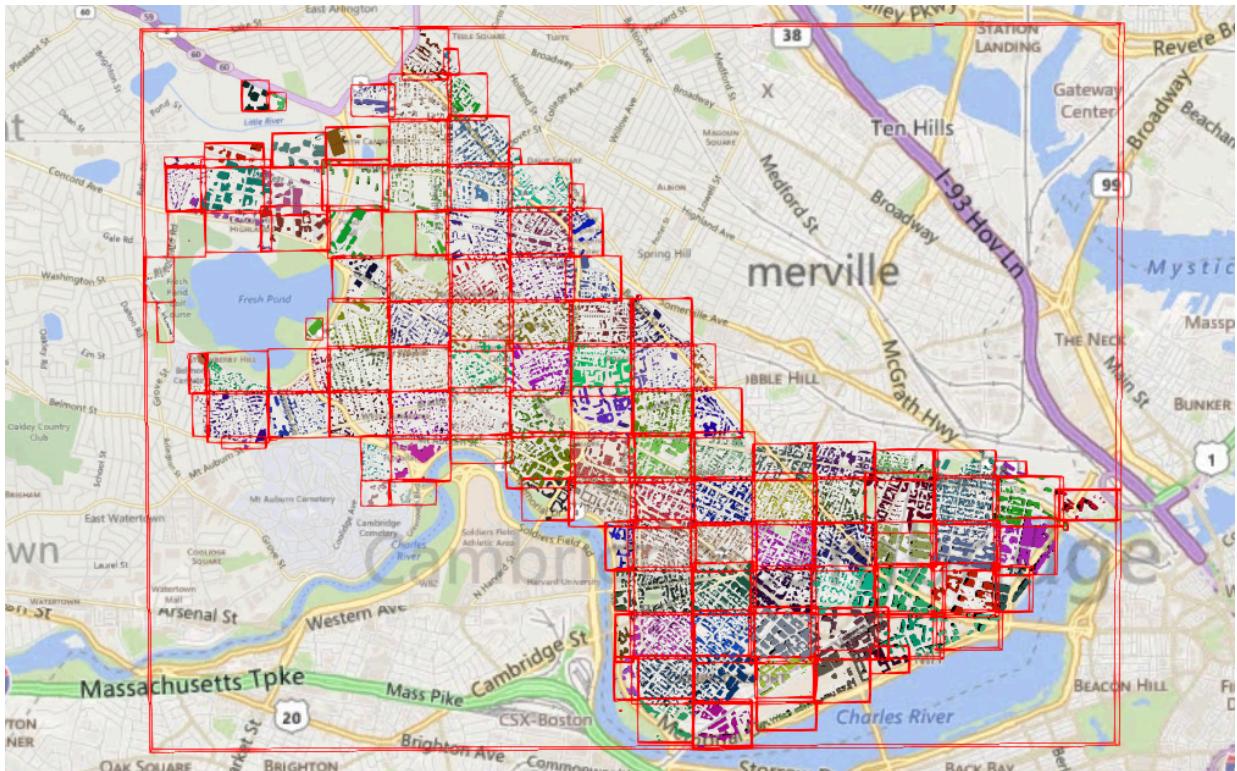


Figure 19 – A tileset with an overlapping grid spatial data structure

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).

6.9. Specifying extensions and application specific extras

3D Tiles defines extensions to allow the base specification to have extensibility for new features, as well as extras to allow for application specific metadata.

6.9.1. 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.b3dm"  
  },  
  "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 must 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 must 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 must also exist in extensionsUsed.

6.9.2. 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.b3dm"  
  },  
  "extras": {  
    "name": "Empire State Building"  
  }  
}
```

6.10. Tile format specifications

Each tile's content.uri property may be the uri of binary blob that contains information for rendering the tile's 3D content. The content is an instance of one of the formats listed in the table below.

| FORMAT | USES |
|--------------------|---|
| Batched 3D Model | Heterogeneous 3D models. E.g. textured terrain and surfaces, 3D building exteriors and interiors, massive models. |
| Instanced 3D Model | 3D model instances. E.g. trees, windmills, bolts. |
| Point Cloud | Massive number of points. |
| Composite | Concatenate tiles of different formats into one tile. |

A tileset can contain any combination of tile formats. 3D Tiles may also support different formats in the same tile using a Composite tile.

7

PROPERTY REFERENCE

PROPERTY REFERENCE

7.1. Tileset

A 3D Tiles tileset.

Properties

| | TYPE | DESCRIPTION | REQUIRE |
|---------------------|-------------|--|---------|
| asset | object | Metadata about the entire tileset. | Yes |
| properties | any | A dictionary object of metadata about per-feature properties. | 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 | object | A tile in a 3D Tiles tileset. | Yes |
| 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 | object | Dictionary object with extension-specific objects. | No |
| extras | any | Application-specific data. | No |

Additional properties are not allowed.

7.1.1. Tileset.asset

Metadata about the entire tileset.

- **Type:** object
- **Required:** Yes

7.1.2. Tileset.properties

A dictionary object of metadata about per-feature properties.

- **Type:** any
- **Required:** No
- **Type of each property:** object

7.1.3. 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

7.1.4. Tileset.root

A tile in a 3D Tiles tileset.

- **Type:** object
- **Required:** Yes

7.1.5. Tileset.extensionsUsed

Names of 3D Tiles extensions used somewhere in this tileset.

- **Type:** string[1-*]
- Each element in the array must be unique.
- **Required:** No

7.1.6. Tileset.extensionsRequired

Names of 3D Tiles extensions required to properly load this tileset.

- **Type:** string[1-*]
- Each element in the array must be unique.
- **Required:** No

7.1.7. Tileset.extensions

Dictionary object with extension-specific objects.

- **Type:** object
- **Required:** No
- **Type of each property:** Extension

7.1.8. Tileset.extras

Application-specific data.

- **Type:** any
- **Required:** No

7.2. Asset

Metadata about the entire tileset.

Properties

| | TYPE | DESCRIPTION | REQUIRE |
|-----------------|--------|--|---------|
| 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 | object | Dictionary object with extension-specific objects. | No |
| extras | any | Application-specific data. | No |

Additional properties are not allowed.

7.2.1. 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

7.2.2. Asset.tilesetVersion

Application-specific version of this tileset, e.g., for when an existing tileset is updated.

- **Type:** string
- **Required:** No

7.2.3. Asset.extensions

Dictionary object with extension-specific objects.

- **Type:** object
- **Required:** No
- **Type of each property:** Extension

7.2.4. Asset.extras

Application-specific data.

- **Type:** any
- **Required:** No

7.3. Bounding Volume

A bounding volume that encloses a tile or its content. Exactly one box, region, or sphere property is required.

Properties

| | TYPE | DESCRIPTION | REQUIRE |
|------------|------------|--|---------|
| 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 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. | No |
| 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 | object | Dictionary object with extension-specific objects. | No |
| extras | any | Application-specific data. | No |

Additional properties are not allowed.

7.3.1. 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

7.3.2. 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

7.3.3. 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

7.3.4. BoundingVolume.extensions

Dictionary object with extension-specific objects.

- **Type:** object
- **Required:** No
- **Type of each property:** Extension

7.3.5. BoundingVolume.extras

Application-specific data.

- **Type:** any
- **Required:** No

7.4. Extension

Dictionary object with extension-specific objects.

Additional properties are allowed.

- **Type of each property:** object

7.5. Extras

Application-specific data.

7.6. Properties

A dictionary object of metadata about per-feature properties.

Properties

| TYPE | DESCRIPTION | REQUIRED | |
|------------|-------------|--|-----|
| maximum | number | The maximum value of this property of all the features in the tileset. | Yes |
| minimum | number | The minimum value of this property of all the features in the tileset. | Yes |
| extensions | object | Dictionary object with extension-specific objects. | No |
| extras | any | Application-specific data. | No |

Additional properties are not allowed.

7.6.1. Properties.maximum

The maximum value of this property of all the features in the tileset.

- **Type:** number
- **Required:** Yes

7.6.2. Properties.minimum

The minimum value of this property of all the features in the tileset.

- **Type:** number
- **Required:** Yes

7.6.3. Properties.extensions

Dictionary object with extension-specific objects.

- **Type:** object
- **Required:** No
- **Type of each property:** Extension

7.6.4. Properties.extras

Application-specific data.

- **Type:** any
- **Required:** No

7.7. Tile

A tile in a 3D Tiles tileset.

Properties

| | TYPE | DESCRIPTION | REQUIRED |
|---------------------|--------|---|----------|
| boundingVolume | object | A bounding volume that encloses a tile or its content. Exactly one box, region, or sphere property is required. | Yes |
| viewerRequestVolume | object | A bounding volume that encloses a tile or its content. Exactly one box, region, or sphere property is required. | No |
| geometricError | number | The error, in meters, introduced if this tile is rendered and its children | Yes |

| Type | Description | Required |
|-----------|---|--|
| | are not. At runtime, the geometric error is used to compute screen space error (SSE), i.e., the error measured in pixels. | |
| refine | <p>Specifies if additive or replacement refinement is used when traversing the tileset for rendering.</p> <p>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.</p> | No |
| transform | <p>A floating-point 4×4 affine transformation matrix, stored in column-major order, that transforms the tile's content—i.e., its features as well as content.</p> <p>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 geometric Error, nor does it apply any volume property when the volume is a region, defined in EPSG:4979 coordinates.</p> | No, default: [1,0,0,0,0,1,0,0,0,0,0,1,0,0,0,1] |
| content | Metadata about the tile's content and a link to the content. | No |
| children | An array of objects that define child tiles. Each child tile content is fully enclosed by its parent | No |

| Type | Description | Required |
|------------|--|----------|
| | 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. | |
| extensions | object Dictionary object with extension-specific objects. | No |
| extras | any Application-specific data. | No |

Additional properties are not allowed.

7.7.1. Tile.boundingVolume

A bounding volume that encloses a tile or its content. Exactly one box, region, or sphere property is required.

- **Type:** object
- **Required:** Yes

7.7.2. Tile.viewerRequestVolume

A bounding volume that encloses a tile or its content. Exactly one box, region, or sphere property is required.

- **Type:** object
- **Required:** No

7.7.3. 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

7.7.4. 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"

7.7.5. Tile.transform

A floating-point 4×4 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 geometricError, nor does it apply any volume property when the volume is a region, defined in EPSG:4979 coordinates.

- **Type:** number[16]
- **Required:** No, default: [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1]

7.7.6. Tile.content

Metadata about the tile's content and a link to the content.

- **Type:** object
- **Required:** No

7.7.7. 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:** array[]
 - Each element in the array must be unique.
- **Required:** No

7.7.8. Tile.extensions

Dictionary object with extension-specific objects.

- **Type:** object
- **Required:** No
- **Type of each property:** Extension

7.7.9. Tile.extras

Application-specific data.

- **Type:** any
- **Required:** No

7.8. Tile Content

Metadata about the tile's content and a link to the content.

Properties

| | TYPE | DESCRIPTION | REQUIRE |
|--------------------|--------|---|---------|
| bounding Volume | object | A bounding volume that encloses a tile or its content. Exactly one box, region, or sphere property is required. | No |
| uri | string | A uri that points to the tile's content. When the uri is relative, it is relative to the referring tileset JSON file. | Yes |
| extensions | object | Dictionary object with extension-specific objects. | No |
| extras | any | Application-specific data. | No |

Additional properties are not allowed.

7.8.1. TileContent.boundingVolume

A bounding volume that encloses a tile or its content. Exactly one box, region, or sphere property is required.

- **Type:** object
- **Required:** No

7.8.2. TileContent.uri

A uri that points to the tile's content. When the uri is relative, it is relative to the referring tileset JSON file.

- **Type:** string
- **Required:** Yes

7.8.3. TileContent.extensions

Dictionary object with extension-specific objects.

- **Type:** object
- **Required:** No
- **Type of each property:** Extension

7.8.4. TileContent.extras

Application-specific data.

- **Type:** any
- **Required:** No

8

FEATURE TABLE

8.1. 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.

8.2. 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 20 – Feature Table layout

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.

8.2.1. Padding

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

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

Binary properties must 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 must start at an offset that is a multiple of 4. Preceding binary properties must be padded with additional bytes, of any value, to satisfy this requirement.

8.2.2. JSON header

Feature Table values can be represented in the JSON header in three 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. An array of values, e.g., "POSITION" : [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0].
 - This is used for per-feature semantics like "POSITION" in Instanced 3D Model. Above, each POSITION refers to a float32[3] data type so there are three features: Feature 0's position=(1.0, 0.0, 0.0), Feature 1's position=(0.0, 1.0, 0.0), Feature 2's position=(0.0, 0.0, 1.0).
3. 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 must 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 must 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.

8.2.3. 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:

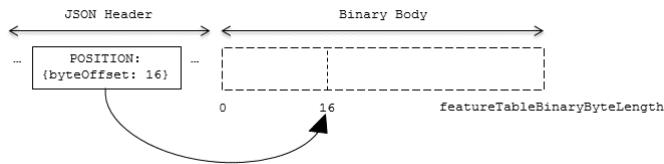


Figure 21 – Feature Table binary body layout

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.

8.3. Implementation example

This section is non-normative

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 Cesium implementation of 3D Tiles.

8.4. Property reference

8.4.1. Feature Table

A set of semantics containing per-tile and per-feature values defining the position and appearance properties for features in a tile.

Properties

| Type | Description | Required | |
|------------|-------------|--|----|
| extensions | object | Dictionary object with extension-specific objects. | No |
| extras | any | Application-specific data. | No |

Additional properties are allowed.

- **Type of each property:** Property

8.4.1.1. FeatureTable.extensions

Dictionary object with extension-specific objects.

- **Type:** object
- **Required:** No
- **Type of each property:** Extension

8.4.1.2. FeatureTable.extras

Application-specific data.

- **Type:** any
- **Required:** No

8.4.2. 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.

Properties

| Type | Description | Required | |
|------------|-------------|--------------------------------------|-----|
| byteOffset | number | The offset into the buffer in bytes. | Yes |

Additional properties are allowed.

8.4.2.1. BinaryBodyReference.byteOffset

The offset into the buffer in bytes.

- **Type:** number
- **Required:** Yes
- **Minimum:** ≥ 0

8.4.3. 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.

9

BATCH TABLE

BATCH TABLE

9.1. 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)

9.2. 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:

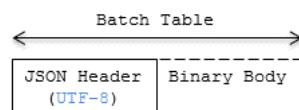


Figure 22 – Batch Table layout

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` `int32` fields, which can be used to extract each respective part of the Batch Table.

9.2.1. Padding

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

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

Binary properties must 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 must start at an offset that is a multiple of 4. Preceding binary properties must be padded with additional bytes, of any value, to satisfy this requirement.

9.2.2. 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` must be a multiple of the size in bytes of the property's `componentType`, e.g., a property with the component type `FLOAT` must 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.

Implementation 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'};
```

9.2.3. 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:

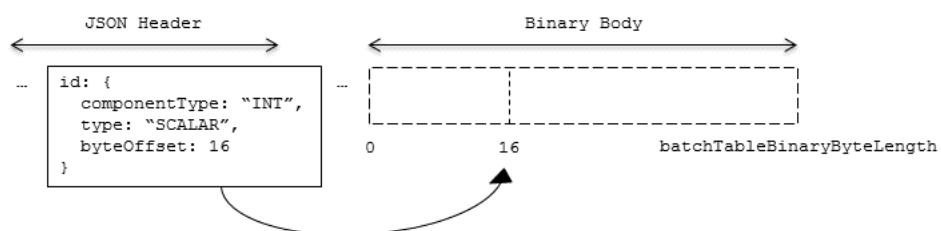


Figure 23 – Batch Table binary body layout

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

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

| COMPONENTTYPE | SIZE IN BYTES |
|---------------|---------------|
| "BYTE" | 1 |

| COMPONENTTYPE | SIZE IN BYTES |
|------------------|----------------------|
| "UNSIGNED_BYTE" | 1 |
| ``SHORT`` | 2 |
| "UNSIGNED_SHORT" | 2 |
| "INT" | 4 |
| "UNSIGNED_INT" | 4 |
| "FLOAT" | 4 |
| "DOUBLE" | 8 |
| type | Number of components |
| "SCALAR" | 1 |
| "VEC2" | 2 |
| "VEC3" | 3 |
| "VEC4" | 4 |

9.3. Implementation example

This section is non-normative

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 *
    numberComponents // 10 * 8 * 3
var geographicArray = new Float64Array(batchTableBinary.buffer, byteOffset,
    geographicArrayByteLength);

// Using subarray creates a view into the array, and not a new array.
var geographicOfFeature = positionArray.subarray(batchId * numberComponents,
    batchId * numberComponents + numberComponents);
```

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

9.4. Property reference

9.4.1. Batch Table

A set of properties defining application-specific metadata for features in a tile.

Properties

| TYPE | DESCRIPTION | REQUIRED |
|------------|-------------|--|
| extensions | object | Dictionary object with extension-specific objects. |
| extras | any | Application-specific data. |

Additional properties are allowed.

- Type of each property: Property

9.4.1.1. BatchTable.extensions

Dictionary object with extension-specific objects.

- Type: object
- Required: No
- Type of each property: Extension

9.4.1.2. BatchTable.extras

Application-specific data.

- Type: any
- Required: No

9.4.2. 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.

Properties

| TYPE | DESCRIPTION | REQUIRED |
|---------------|-------------|--|
| byteOffset | number | The offset into the buffer in bytes. |
| componentType | string | The datatype of components in the property. |
| type | string | Specifies if the property is a scalar or vector. |

Additional properties are allowed.

9.4.2.1. BinaryBodyReference.byteOffset

The offset into the buffer in bytes.

- Type: number

- **Required:** Yes
- **Minimum:** >= 0

9.4.2.2. 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"

9.4.2.3. BinaryBodyReference.type

Specifies if the property is a scalar or vector.

- **Type:** string
- **Required:** Yes
- **Allowed values:**
 - "SCALAR"
 - "VEC2"
 - "VEC3"
 - "VEC4"

9.4.3. 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.

10

TILE FORMAT SPECIFICATIONS

TILE FORMAT SPECIFICATIONS

Each tile's content.uri property points to a tile that is one of the formats listed in the table below.

| FORMAT | USES |
|------------------------------|---|
| Batched 3D Model (b3dm) | Heterogeneous 3D models. E.g. textured terrain and surfaces, 3D building exteriors and interiors, massive models. |
| Instanced 3D Model (i3dm) | 3D model instances. E.g. trees, windmills, bolts. |
| Point Cloud (pnts) | Massive number of points. |
| Composite (cmpt) | Concatenate tiles of different formats into one tile. |

A tileset can contain any combination of tile formats. 3D Tiles may also support different formats in the same tile using a Composite tile.

10.1. Batched 3D Model

10.1.1. 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.

10.1.2. 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):

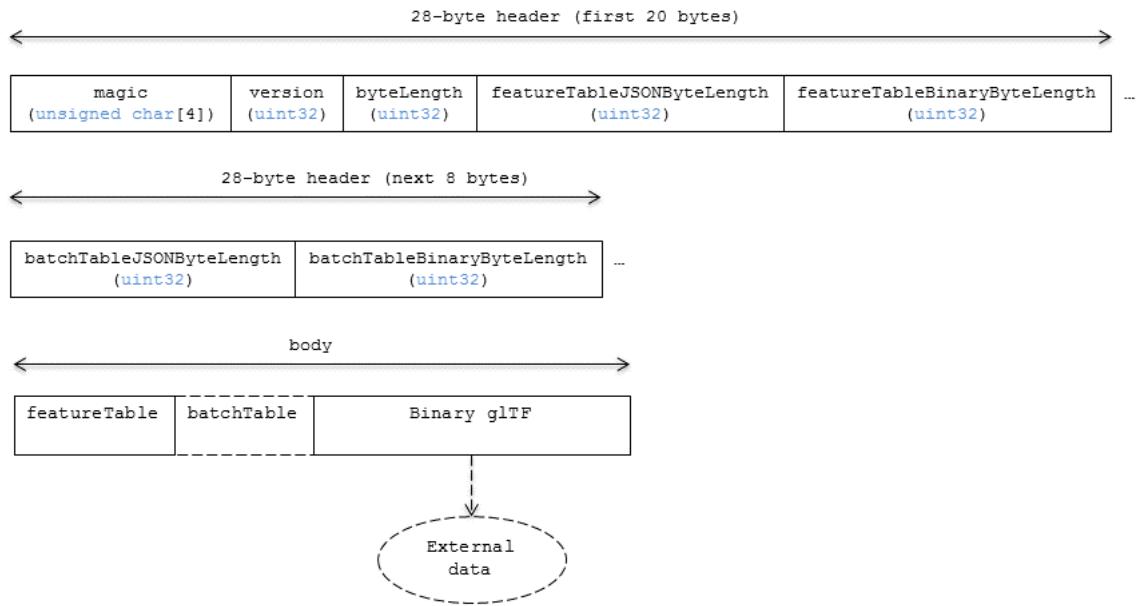


Figure 24 – Batched 3D Model layout

10.1.2.1. Padding

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

The binary glTF must 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.

10.1.3. Header

The 28-byte header contains the following fields:

| 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. |

| FIELD NAME | DATA TYPE | DESCRIPTION |
|------------------------------|-----------|---|
| 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.

10.1.4. Feature Table

Contains values for b3dm semantics.

More information is available in the Feature Table specification.

10.1.4.1. Semantics

10.1.4.1.1. Feature semantics

There are currently no per-feature semantics.

10.1.4.1.2. Global semantics

These semantics define global properties for all features.

| SEMANTIC | DATA TYPE | DESCRIPTION | REQUIRE |
|--------------|------------|---|---------|
| BATCH_LENGTH | uint32 | The number of distinguishable models, also called features, in the batch. If the Binary glTF does not have a batchId attribute, this field must 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. |

10.1.5. Batch Table

The *Batch Table* contains per-model 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. In the binary glTF section, each vertex has an numeric `batchId` attribute in the integer range $[0, \text{ number of models in the batch } - 1]$. 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.

10.1.6. 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":5125,
```

```

        "count": 4860,
        "max": [2],
        "min": [0],
        "type": "SCALAR"
    }
]
}

```

The accessor.type must be a value of “SCALAR”. All other properties must 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.

10.1.6.1. 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 must 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.

10.1.7. File extension and MIME type

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

10.1.8. Implementation example

This section is non-normative

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

10.1.8.1. Property reference

10.1.8.1.1. Batched 3D Model Feature Table

A set of Batched 3D Model semantics that contain additional information about features in a tile.

Properties

| Type | Description | Require | |
|--------------|---------------------------------|---|-----|
| extensions | object | Dictionary object with extension-specific objects. | No |
| extras | any | Application-specific data. | No |
| BATCH_LENGTH | object, number[1], number | A GlobalPropertyScalar object defining a numeric property for all features. See the corresponding property semantic in Semantics. | Yes |
| RTC_CENTER | object, number[3] | A GlobalPropertyCartesian3 object defining a 3-component numeric property for all features. See the corresponding property semantic in Semantics. | No |

Additional properties are allowed.

- **Type of each property:** Property

10.1.8.1.1.1. Batched3DModelFeatureTable.extensions

Dictionary object with extension-specific objects.

- **Type:** object
- **Required:** No
- **Type of each property:** Extension

10.1.8.1.1.2. Batched3DModelFeatureTable.extras

Application-specific data.

- **Type:** any
- **Required:** No

10.1.8.1.1.3. Batched3DModelFeatureTable.BATCH_LENGTH

A GlobalPropertyCartesian3 object defining a numeric property for all features. See the corresponding property semantic in Semantics.

- **Type:** object, number[1], number
- **Required:** Yes

10.1.8.1.1.4. Batched3DModelFeatureTable.RTC_CENTER

A GlobalPropertyCartesian3 object defining a 3-component numeric property for all features. See the corresponding property semantic in Semantics.

- **Type:** object, number[3]
- **Required:** No

10.1.8.1.2. 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.

Properties

| Type | Description | Required |
|------------|--|----------|
| byteOffset | number The offset into the buffer in bytes. | Yes |

Additional properties are allowed.

10.1.8.1.2.1. BinaryBodyReference.byteOffset

The offset into the buffer in bytes.

- **Type:** number
- **Required:** Yes
- **Minimum:** ≥ 0

10.1.8.1.3. GlobalPropertyCartesian3

An object defining a global 3-component numeric property values for all features.

10.1.8.1.4. GlobalPropertyScalar

An object defining a global numeric property values for all features.

10.1.8.1.5. 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.

10.2. Instanced 3D Model

10.2.1. 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.

Implementation Note: A 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.

Implementation Note: Instanced 3D Model maps well to the [ANGLE_instanced_arrays](#) extension for efficient rendering with WebGL.

10.2.2. 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):

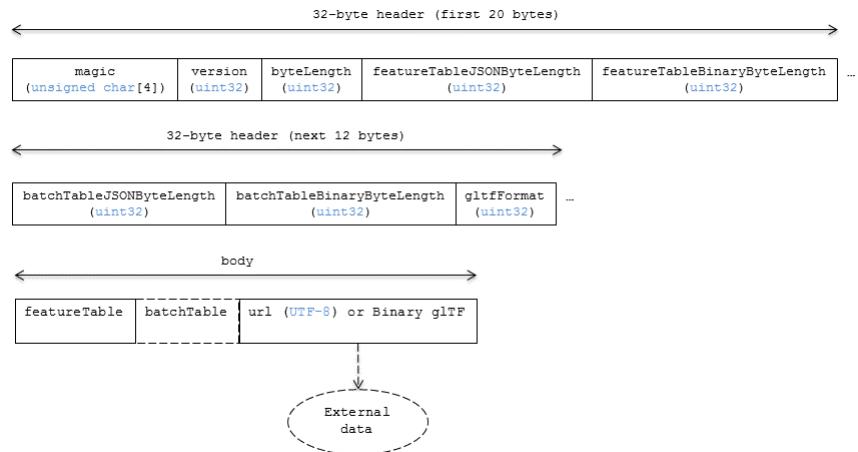


Figure 25 – Instanced 3D Model layout

10.2.2.1. Padding

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

The binary glTF (if present) must 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 must be padded with trailing Space characters (0x20) to satisfy alignment requirements of the tile, which must be removed at runtime before requesting the glTF asset.

10.2.3. Header

The 32-byte header contains the following fields:

| 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. |

| FIELD NAME | DATA TYPE | DESCRIPTION |
|------------------------------|-----------|--|
| 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.

10.2.4. Feature Table

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

10.2.4.1. Semantics

10.2.4.1.1. Instance semantics

These semantics map to an array of feature values that are used to create instances. The length of these arrays must be the same for all semantics and is equal to the number of instances.

The value for each instance semantic must 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 must 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.

| 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_ |

| SEMANTIC | DATA TYPE | DESCRIPTION | REQUIRED |
|---------------------|------------------------------------|--|--|
| 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. | QUANTIZED 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. Must 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 for the orientation of the instance. Must be orthogonal to up. | No, unless NORMAL_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. |

10.2.4.1.2. Global semantics

These semantics define global properties for all instances.

| 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. |

| SEMANTIC | DATA TYPE | DESCRIPTION | REQUIRED |
|------------------------|------------|--|---|
| 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.

10.2.4.2. 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.

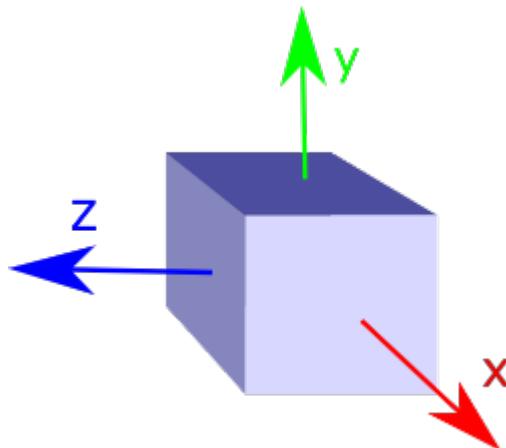


Figure 26 – A box in the standard basis

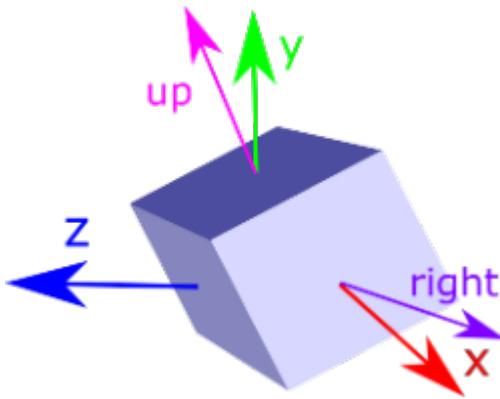


Figure 27 – A box transformed into a rotated basis

10.2.4.2.1. 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.

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

10.2.4.2.2. 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.

10.2.4.3. Instance position

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

10.2.4.3.1. 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.

10.2.4.3.2. 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:

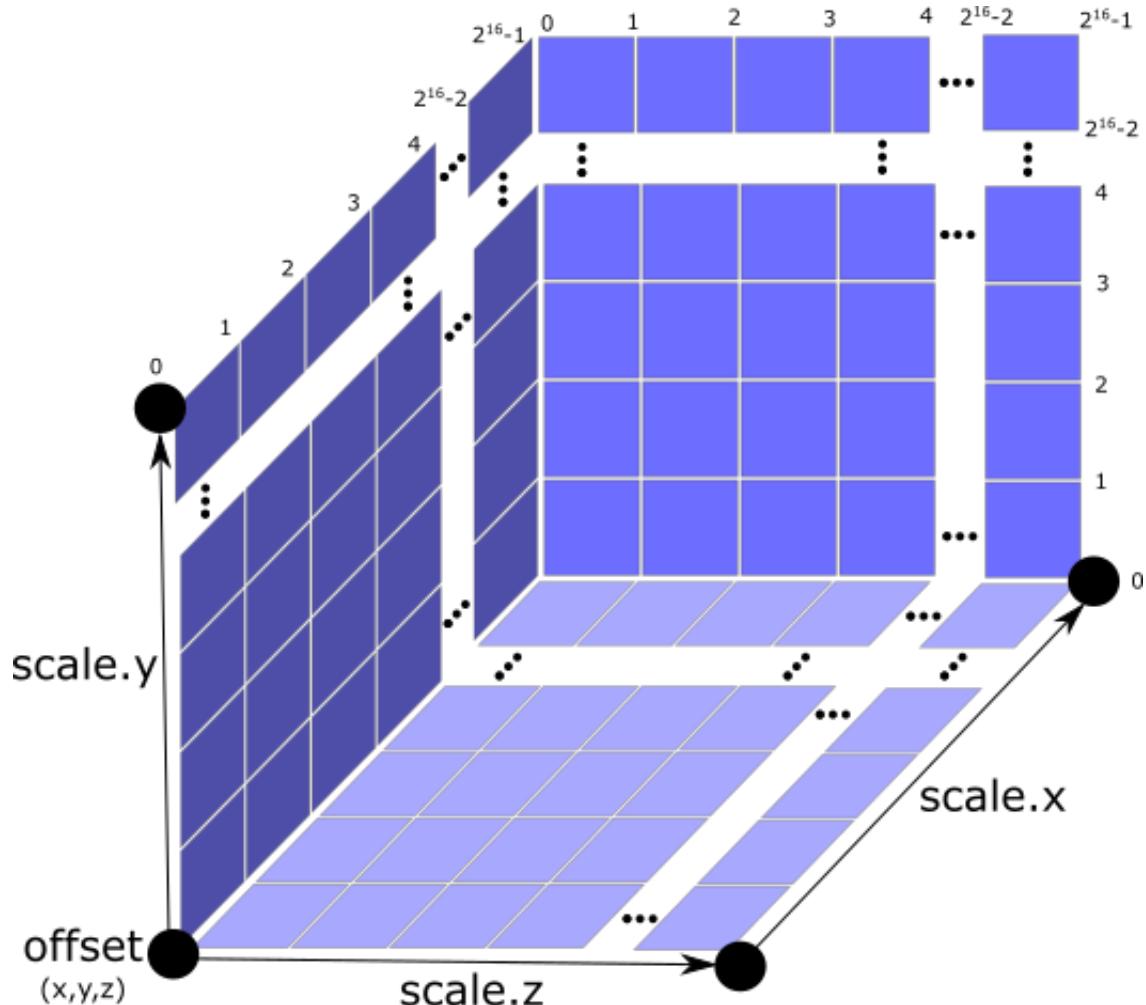


Figure 28 – A quantized volume

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
```

10.2.4.4. 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.

10.2.4.5. Examples

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

10.2.4.5.1. 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);
```

10.2.4.5.2. 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);

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

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

```

10.2.5. Batch Table

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

10.2.6. 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.

In either case, header.gltfByteLength contains the length of the glTF field in bytes.

10.2.6.1. 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 must be transformed at runtime. See glTF transforms for more details.

10.2.7. File extension and MIME type

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

10.2.8. Property reference

10.2.8.1. 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.

Properties

| | TYPE | DESCRIPTION | REQUIRE |
|---------------------------------|--------|--|---------|
| <code>extensions</code> | object | Dictionary object with extension-specific objects. | No |
| <code>extras</code> | any | Application-specific data. | No |
| <code>POSITION</code> | object | A <code>BinaryBodyReference</code> object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics. | No |
| <code>POSITION_QUANTIZED</code> | object | A <code>BinaryBodyReference</code> object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics. | No |
| <code>NORMAL_UP</code> | object | A <code>BinaryBodyReference</code> object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics. | No |
| <code>NORMAL_RIGHT</code> | object | A <code>BinaryBodyReference</code> object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics. | No |

| | TYPE | DESCRIPTION | REQUIRE |
|-------------------------|---------------------------------|---|---------|
| NORMAL_UP_OCT32P | object | A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics. | No |
| NORMAL_RIGHT_OCT32P | object | A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics. | No |
| SCALE | object | A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics. | No |
| SCALE_NON_UNIFORM | object | A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics. | No |
| BATCH_ID | object | A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics. | No |
| INSTANCES_LENGTH | object, number[1], number | A GlobalPropertyScalar object defining a numeric property for all features. See the corresponding property semantic in Semantics. | Yes |
| QUANTIZED_VOLUME_OFFSET | object, number[3] | A GlobalPropertyCartesian3 object defining a 3-component numeric property for all features. See the corresponding property semantic in Semantics. | No |
| QUANTIZED_VOLUME_SCALE | object, number[3] | A GlobalPropertyCartesian3 object defining a 3-component numeric property for all features. See the corresponding property semantic in Semantics. | No |

Additional properties are allowed.

- **Type of each property:** Property

10.2.8.1.1. Instanced3DModelFeatureTable.extensions

Dictionary object with extension-specific objects.

- **Type:** object
- **Required:** No
- **Type of each property:** Extension

10.2.8.1.2. Instanced3DModelFeatureTable.extras

Application-specific data.

- **Type:** any
- **Required:** No

10.2.8.1.3. Instanced3DModelFeatureTable.POSITION

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics.

- **Type:** object
- **Required:** No

10.2.8.1.4. Instanced3DModelFeatureTable.POSITION_QUANTIZED

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics.

- **Type:** object
- **Required:** No

10.2.8.1.5. Instanced3DModelFeatureTable.NORMAL_UP

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics.

- **Type:** object
- **Required:** No

10.2.8.1.6. Instanced3DModelFeatureTable.NORMAL_RIGHT

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics.

- **Type:** object

- **Required:** No

10.2.8.1.7. Instanced3DModelFeatureTable.NORMAL_UP_OCT32P

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics.

- **Type:** object
- **Required:** No

10.2.8.1.8. Instanced3DModelFeatureTable.NORMAL_RIGHT_OCT32P

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics.

- **Type:** object
- **Required:** No

10.2.8.1.9. Instanced3DModelFeatureTable.SCALE

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics.

- **Type:** object
- **Required:** No

10.2.8.1.10. Instanced3DModelFeatureTable.SCALE_NON_UNIFORM

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics.

- **Type:** object
- **Required:** No

10.2.8.1.11. Instanced3DModelFeatureTable.BATCH_ID

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics.

- **Type:** object
- **Required:** No

10.2.8.1.12. Instanced3DModelFeatureTable.INSTANCES_LENGTH

A GlobalPropertyScalar object defining a numeric property for all features. See the corresponding property semantic in Semantics.

- **Type:** object, number[1], number
- **Required:** Yes

10.2.8.1.13. Instanced3DModelFeatureTable.QUANTIZED_VOLUME_OFFSET

A GlobalPropertyCartesian3 object defining a 3-component numeric property for all features. See the corresponding property semantic in Semantics.

- **Type:** object, number[3]
- **Required:** No

10.2.8.1.14. Instanced3DModelFeatureTable.QUANTIZED_VOLUME_SCALE

A GlobalPropertyCartesian3 object defining a 3-component numeric property for all features. See the corresponding property semantic in Semantics.

- **Type:** object, number[3]
- **Required:** No

10.2.8.2. 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.

Properties

| Type | Description | Required |
|------------|-------------|--------------------------------------|
| byteOffset | number | The offset into the buffer in bytes. |

Additional properties are allowed.

10.2.8.2.1. BinaryBodyReference.byteOffset

The offset into the buffer in bytes.

- **Type:** number
- **Required:** Yes
- **Minimum:** >= 0

10.2.8.3. GlobalPropertyCartesian3

An object defining a global 3-component numeric property values for all features.

10.2.8.4. GlobalPropertyScalar

An object defining a global numeric property values for all features.

10.2.8.5. 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.

10.3. Point Cloud

10.3.1. 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.

10.3.2. 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):

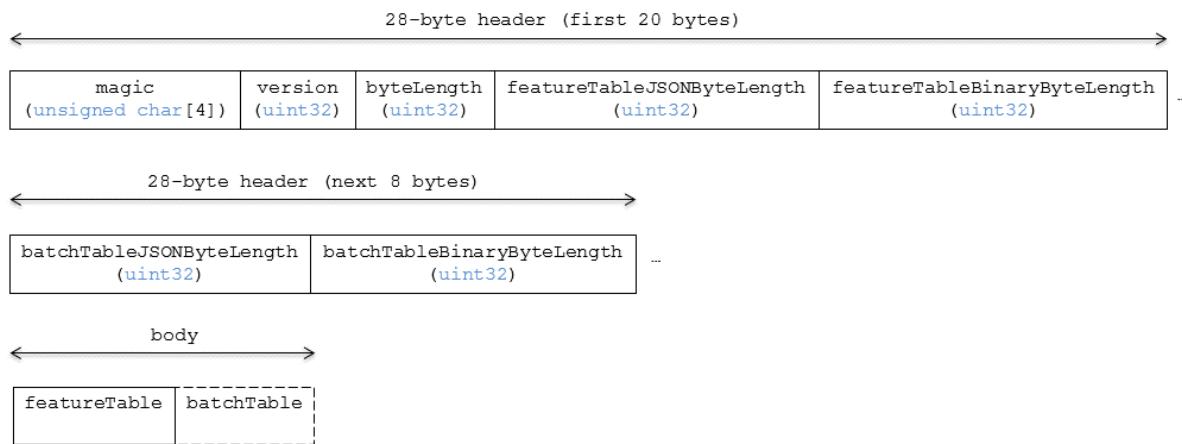


Figure 29 – Point Cloud layout

10.3.2.1. Padding

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

10.3.3. Header

The 28-byte header contains the following fields:

| 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. |

| FIELD NAME | DATA TYPE | DESCRIPTION |
|----------------------------|-----------|---|
| 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. |

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

10.3.4. 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.

10.3.4.1. Semantics

10.3.4.1.1. Point semantics

These semantics map to an array of feature values that define each point. The length of these arrays must be the same for all semantics and is equal to the number of points.

The value for each point semantic must 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 must 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.

| 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. |

10.3.4.1.2. Global semantics

These semantics define global properties for all points.

| 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. |
| 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.

10.3.4.2. Point positions

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

10.3.4.2.1. 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).

10.3.4.2.2. 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.

10.3.4.2.3. 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:

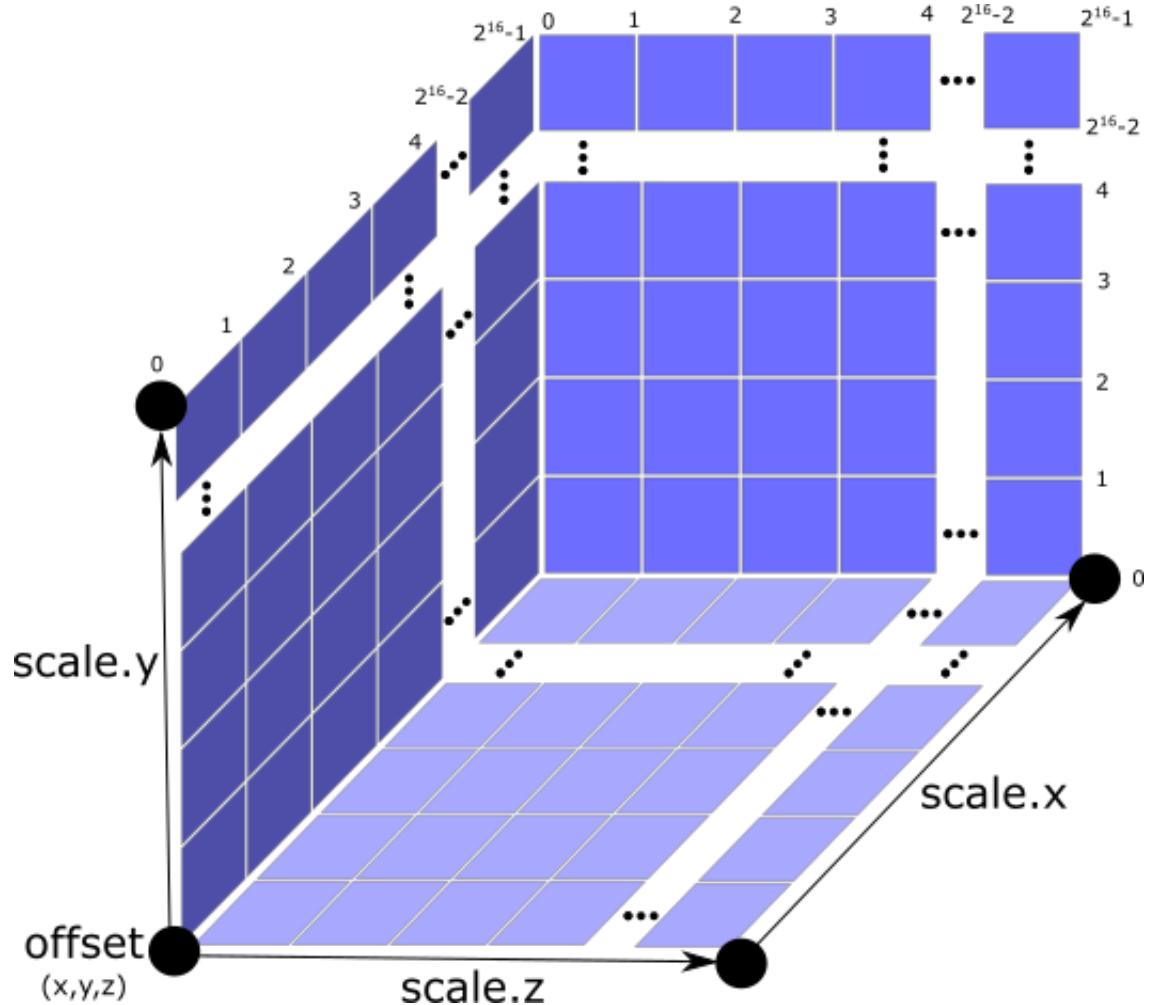


Figure 30 – A quantized volume

`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
```

10.3.4.3. 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, a 3D Tiles Style may be used to change the final rendered color and other visual properties at runtime.

10.3.4.4. 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.

10.3.4.4.1. 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.

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

10.3.4.5. 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.

10.3.4.6. Examples

This section is non-normative

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

10.3.4.6.1. 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);

```

10.3.4.6.2. 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
    },
    RGB : {
        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]);

```

10.3.4.6.3. 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]);

```

10.3.4.6.4. 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,
    1.0,0.0,1.0
]).buffer);

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

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

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

```

10.3.4.6.5. 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']
};
```

10.3.5. 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 link: [http://.../ggetz/Documents/_Batch_Table\[Batch Table\]](http://.../ggetz/Documents/_Batch_Table[Batch Table]) reference for more information.

10.3.6. File extension and MIME type

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

10.3.7. Implementation example

This section is non-normative

Code for reading the header can be found in [PointCloud3DModelTileContent.js](#) in the Cesium implementation of 3D Tiles.

10.3.8. Property reference

10.3.8.1. Point Cloud Feature Table

A set of Point Cloud semantics that contains values defining the position and appearance properties for points in a tile.

Properties

| | TYPE | DESCRIPTION | REQUIRE |
|--------------------|--------|---|---------|
| extensions | object | Dictionary object with extension-specific objects. | No |
| extras | any | Application-specific data. | No |
| POSITION | object | A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics. | No |
| POSITION_QUANTIZED | object | A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics. | No |
| RGBA | object | A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics. | No |
| RGB | object | A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics. | No |
| RGB565 | object | A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics. | No |
| NORMAL | object | A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics. | No |

| | TYPE | DESCRIPTION | REQUIRE |
|-------------------------|---------------------------|---|---------|
| NORMAL_OCT16P | object | A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics. | No |
| BATCH_ID | object | A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics. | No |
| POINTS_LENGTH | object, number[1], number | A GlobalPropertyScalar object defining a numeric property for all points. See the corresponding property semantic in Semantics. | Yes |
| RTC_CENTER | object, number[3] | A GlobalPropertyCartesian3 object defining a 3-component numeric property for all points. See the corresponding property semantic in Semantics. | No |
| QUANTIZED_VOLUME_OFFSET | object, number[3] | A GlobalPropertyCartesian3 object defining a 3-component numeric property for all points. See the corresponding property semantic in Semantics. | No |
| QUANTIZED_VOLUME_SCALE | object, number[3] | A GlobalPropertyCartesian3 object defining a 3-component numeric property for all points. See the corresponding property semantic in Semantics. | No |
| CONSTANT_RGBA | object, number[4] | A GlobalPropertyCartesian4 object defining a 4-component numeric property for all points. See the corresponding property semantic in Semantics. | No |
| BATCH_LENGTH | object, number[1], number | A GlobalPropertyScalar object defining a numeric property for all points. See the corresponding property semantic in Semantics. | No |

Additional properties are allowed.

- **Type of each property:** Property

10.3.8.1.1. PointCloudFeatureTable.extensions

Dictionary object with extension-specific objects.

- **Type:** object
- **Required:** No
- **Type of each property:** Extension

10.3.8.1.2. PointCloudFeatureTable.extras

Application-specific data.

- **Type:** any
- **Required:** No

10.3.8.1.3. PointCloudFeatureTable.POSITION

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics.

- **Type:** object
- **Required:** No

10.3.8.1.4. PointCloudFeatureTable.POSITION_QUANTIZED

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics.

- **Type:** object
- **Required:** No

10.3.8.1.5. PointCloudFeatureTable.RGBA

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics.

- **Type:** object
- **Required:** No

10.3.8.1.6. PointCloudFeatureTable.RGB

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics.

- **Type:** object

- **Required:** No

10.3.8.1.7. PointCloudFeatureTable.RGB565

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics.

- **Type:** object
- **Required:** No

10.3.8.1.8. PointCloudFeatureTable.NORMAL

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics.

- **Type:** object
- **Required:** No

10.3.8.1.9. PointCloudFeatureTable.NORMAL_OCT16P

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics.

- **Type:** object
- **Required:** No

10.3.8.1.10. PointCloudFeatureTable.BATCH_ID

A BinaryBodyReference object defining the reference to a section of the binary body where the property values are stored. See the corresponding property semantic in Semantics.

- **Type:** object
- **Required:** No

10.3.8.1.11. PointCloudFeatureTable.POINTS_LENGTH

A GlobalPropertyScalar object defining a numeric property for all points. See the corresponding property semantic in Semantics.

- **Type:** object, number[1], number
- **Required:** Yes

10.3.8.1.12. PointCloudFeatureTable.RTC_CENTER

A GlobalPropertyCartesian3 object defining a 3-component numeric property for all points. See the corresponding property semantic in Semantics.

- **Type:** object, number[3]
- **Required:** No

10.3.8.1.13. PointCloudFeatureTable.QUANTIZED_VOLUME_OFFSET

A GlobalPropertyCartesian3 object defining a 3-component numeric property for all points. See the corresponding property semantic in Semantics.

- **Type:** object, number[3]
- **Required:** No

10.3.8.1.14. PointCloudFeatureTable.QUANTIZED_VOLUME_SCALE

A GlobalPropertyCartesian3 object defining a 3-component numeric property for all points. See the corresponding property semantic in Semantics.

- **Type:** object, number[3]
- **Required:** No

10.3.8.1.15. PointCloudFeatureTable.CONSTANT_RGBA

A GlobalPropertyCartesian4 object defining a 4-component numeric property for all points. See the corresponding property semantic in Semantics.

- **Type:** object, number[4]
- **Required:** No

10.3.8.1.16. PointCloudFeatureTable.BATCH_LENGTH

A GlobalPropertyScalar object defining a numeric property for all points. See the corresponding property semantic in Semantics.

- **Type:** object, number[1], number
- **Required:** No

10.3.8.2. 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.

Properties

| Type | Description | Required |
|------------|--|----------|
| byteOffset | number The offset into the buffer in bytes. | Yes |

Additional properties are allowed.

10.3.8.2.1. BinaryBodyReference.byteOffset

The offset into the buffer in bytes.

- **Type:** number
- **Required:** Yes
- **Minimum:** ≥ 0

10.3.8.3. GlobalPropertyCartesian3

An object defining a global 3-component numeric property values for all features.

10.3.8.4. GlobalPropertyCartesian4

An object defining a global 4-component numeric property values for all features.

10.3.8.5. GlobalPropertyScalar

An object defining a global numeric property values for all features.

10.3.8.6. 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.

10.4. Composite

10.4.1. 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.

10.4.2. Layout

Composite layout (dashes indicate optional fields):

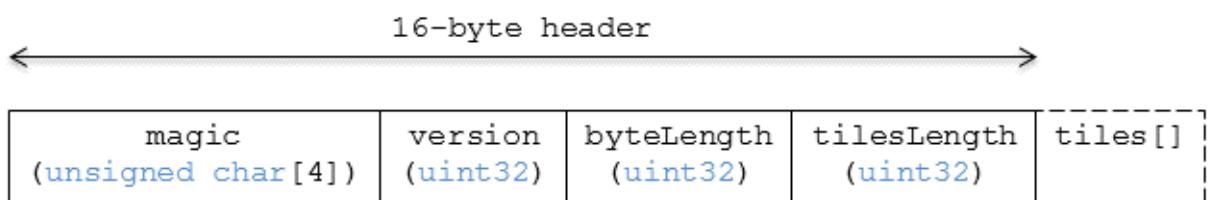


Figure 31 – Composite layout

10.4.2.1. Padding

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

10.4.3. Header

The 16-byte header section contains the following fields:

| 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. |

10.4.4. 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:

| FIELD NAME | DATA TYPE | DESCRIPTION |
|------------|--------------------|---------------------------|
| magic | 4-byte ANSI string | Indicates the tile format |
| version | uint32 | 1 |

| FIELD NAME | DATA TYPE | DESCRIPTION |
|------------|-----------|---------------------------------------|
| byteLength | uint32 | Length, in bytes, of the entire tile. |

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

10.4.5. File extension and MIME type

Composite tiles use the .cmpt extension and application/octet-stream MIME 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.

10.4.6. Implementation examples

This section is non-normative

- [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 Cesium implementation of 3D Tiles.

11

DECLARATIVE STYLING SPECIFICATION

11.1. 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)"]  
    ]  
  }  
}
```

11.2. Concepts

11.2.1. 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))"
}

```

11.2.2. 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" : [
    ["(${Height} >= 1.0) && (${Height} < 10.0)", "color('#FF00FF')"],
    ["(${Height} >= 10.0) && (${Height} < 30.0)", "color('#FF0000')"],
    ["(${Height} >= 30.0) && (${Height} < 50.0)", "color('#FFFF00')"],
    ["(${Height} >= 50.0) && (${Height} < 70.0)", "color('#00FF00')"],
    ["(${Height} >= 70.0) && (${Height} < 100.0)", "color('#00FFFF')"],
    ["(${Height} >= 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')"]
  ]
}

```

11.2.3. 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')"]
    ]
  }
}

```

```
}
```

11.2.4. 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}"
  }
}
```

11.3. 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.

Implementation Note: Cesium uses the `jsep` JavaScript expression parser library to parse style expressions into an [abstract syntax tree \(AST\)](#).

11.3.1. 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), color('cyan', 0.5))`.

11.3.2. 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.

11.3.3. 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'))

11.3.3.1. Number

As in JavaScript, numbers can be NaN or Infinity. The following test functions are supported:

- isNaN(testValue : Number) : Boolean
- isFinite(testValue : Number) : Boolean

11.3.3.2. 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:

11.3.3.2.1. 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

11.3.3.2.2. 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

11.3.3.2.3. `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

11.3.3.2.4. 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.

11.3.3.3. 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)`

11.3.3.4. 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')
```

11.3.4. Operator rules

- Unary operators + and - operate only on number and vector expressions.
- Unary operator ! operates only on boolean expressions.
- Binary operators <, \leq , >, and \geq 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 must be a boolean expression.

11.3.5. 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.

11.3.6. 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/"

11.3.7. Constants

The following constants are supported by the styling language:

11.3.7.1. PI

The mathematical constant PI, which represents a circle's circumference divided by its diameter, approximately 3.14159.

```
{
  "show" : "cos(${Angle} + Math.PI) < 0"
}
```

11.3.7.2. E

Euler's constant and the base of the natural logarithm, approximately 2.71828.

```
{
  "color" : "color() * pow(Math.E / 2.0, ${Temperature})"
}
```

11.3.8. 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. feature is implicit and can be omitted in most cases.

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:

| 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]
```

11.3.9. Built-in functions

The following built-in functions are supported by the styling language. 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.

11.3.9.1. 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"  
}
```

11.3.9.2. 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')"]  
    ]  
  }  
}
```

11.3.9.3. 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"  
}
```

11.3.9.4. 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"
}
```

11.3.9.5. 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"
}
```

11.3.9.6. 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"
}
```

11.3.9.7. 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"
}
```

11.3.9.8. 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"  
}
```

11.3.9.9. 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"  
}
```

11.3.9.10. 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"  
}
```

11.3.9.11. 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"  
}
```

11.3.9.12. 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"
}
```

11.3.9.13. 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"
}
```

11.3.9.14. 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"
}
```

11.3.9.15. 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"  
}
```

11.3.9.16. 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"  
}
```

11.3.9.17. 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"  
}
```

11.3.9.18. 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"  
}
```

11.3.9.19. 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"  
}
```

11.3.9.20. 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})"  
}
```

11.3.9.21. 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"  
}
```

11.3.9.22. 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"
```

```
}
```

11.3.9.23. 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"
}
```

11.3.9.24. 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)"
}
```

11.3.9.25. 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"  
}
```

11.3.9.26. 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"  
}
```

11.3.9.27. 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"  
}
```

11.3.9.28. 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"  
}
```

11.3.9.29. 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"  
}
```

11.3.9.30. 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)"  
}
```

11.3.10. Notes

Comments are not supported.

11.4. 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 must 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"  
}
```

Implementation 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

11.5. File extension and MIME type

Tileset styles use the `.json` extension and the `application/json` mime type.

11.6. Property reference

11.6.1. style

A 3D Tiles style.

Properties

| TYPE | DESCRIPTION | REQUIRED |
|---|--|--|
| <code>defines</code> 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. | No |
| <code>show</code> boolean, string, object | A boolean expression or conditions property which determines if a feature should be shown. | No, default: true |
| <code>color</code> string, object | A color expression or conditions property which determines the color blended with the feature's intrinsic color. | No, default: <code>color('FFFFFF')</code> |
| <code>meta</code> object | A meta object which determines the values of non-visual properties of the feature. | No |

Additional properties are not allowed.

11.6.1.1. 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
- **Type of each property:** string

11.6.1.2. style.show

A boolean expression or conditions property which determines if a feature should be shown.

- **Type:** boolean, string, object

- **Required:** No, default: true

11.6.1.3. style.color

A color expression or conditions property which determines the color blended with the feature's intrinsic color.

- **Type:** string, object
- **Required:** No, default: color('#FFFFFF')

11.6.1.4. style.meta

A meta object which determines the values of non-visual properties of the feature.

- **Type:** object
- **Required:** No
- **Type of each property:** string

11.6.2. boolean expression

A boolean or string with a 3D Tiles style expression that evaluates to a boolean. See Expressions.

11.6.3. color expression

3D Tiles style expression that evaluates to a Color. See Expressions.

11.6.4. conditions

A series of conditions evaluated in order, like a series of if...else statements that result in an expression being evaluated.

Properties

| Type | Description | Required |
|---------------------|--|----------|
| conditions array[] | 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 must all be the same type. If no condition evaluates | No |

to true, the result is undefined. When conditions is undefined, null, or an empty object, the result is undefined.

Additional properties are not allowed.

11.6.4.1. 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 must 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:** array[]
- **Required:** No

11.6.5. 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.

11.6.6. expression

A valid 3D Tiles style expression. See Expressions

11.6.7. meta

A series of property names and the expression to evaluate for the value of that property.

Additional properties are allowed.

- **Type of each property:** expression

11.6.8. number expression

3D Tiles style expression that evaluates to a number. See Expressions.

11.6.9. Point Cloud Style

A 3D Tiles style with additional properties for Point Clouds.

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. | No |
| show boolean, string, object | A boolean expression or conditions property which determines if a feature should be shown. | No, default: true |
| color string, object | A color expression or conditions property which determines the color blended with the feature's intrinsic color. | No, default: color('FFFFFF') |
| meta object | A meta object which determines the values of non-visual properties of the feature. | No |
| pointSize number, string, object | A number expression or conditions property which determines the size of the points in pixels. | No, default: 1 |

Additional properties are not allowed.

11.6.9.1. PointCloudStyle.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
- **Type of each property:** string

11.6.9.2. PointCloudStyle.show

A boolean expression or conditions property which determines if a feature should be shown.

- **Type:** boolean, string, object
- **Required:** No, default: true

11.6.9.3. PointCloudStyle.color

A color expression or conditions property which determines the color blended with the feature's intrinsic color.

- **Type:** string, object
- **Required:** No, default: color('#FFFFFF')

11.6.9.4. PointCloudStyle.meta

A meta object which determines the values of non-visual properties of the feature.

- **Type:** object
- **Required:** No
- **Type of each property:** string

11.6.9.5. PointCloudStyle.pointSize

A number expression or conditions property which determines the size of the points in pixels.

- **Type:** number, string, object
- **Required:** No, default: 1



A

ANNEX A (NORMATIVE) CONFORMANCE CLASS ABSTRACT TEST SUITE



A

ANNEX A (NORMATIVE) CONFORMANCE CLASS ABSTRACT TEST SUITE

An Abstract Test Suite is not required for a Community Standard



B

ANNEX B (INFORMATIVE) CONTRIBUTOR ACKNOWLEDGEMENTS

ANNEX B (INFORMATIVE) CONTRIBUTOR ACKNOWLEDGEMENTS

Matt Amato
Erik Andersson
Dan Bagnell
Ray Bentley
Jannes Bolling
Dylan Brown
Sarah Chow
Paul Connelly
Volker Coors
Tom Fili
Leesa Fini
Ralf Gutbell
Frederic Houbie
Christopher Mitchell, Ph.D.
Claus Nagel
Jean-Philippe Pons
Carl Reed
Kevin Ring
Scott Simmons
Stan Tillman
Piero Toffanin
Pano Voudouris
Dave Wesloh



C

ANNEX C (INFORMATIVE) REVISION HISTORY

ANNEX C (INFORMATIVE) REVISION HISTORY

| DATE | RELEASE | AUTHOR | DESCRIPTION |
|------------|---------|------------|---|
| 2018-06-04 | 1.0 | Gabby Getz | Put 3D Tiles specification document into OGC document template |
| 2018-08-03 | 1.0r1 | Gabby Getz | Respond to public comments |
| | | | <ul style="list-style-type: none"> • Updated conformance section to include schema references • Update preface to address updates to WKT for CRS (OGC12-053r5) • Revise Introduction section to clarify conceptual concepts, including tile, tile content, tile formats, features, properties, styles, and the Batch Table and Features Table and how glTF is used in 3D Tiles • Updated “tile” and “HLOD” definitions, added “tile content”, “tile format”, “geometric error”, “screen space error”, and “spatial coherence” definitions • Add reference to EPSG in CRS section • Remove glTF transforms from CRS section. Add only to relevant sections after being introduced. • Remove references to “generation tool” • Fix phrasing describing Tileset JSON • Revise Tile Formats intro • Revise embedded glTF wording. • Fix “required” property errors in the properties reference as generated from the schema • Remove 3d_tileset_time 3D Tiles Style variable • Add clarification for styling tilesets without features • Update normative references to include: <ul style="list-style-type: none"> • EPSG 4979 • IETF RFC3986 • IETF RFC2397 • UTF-8 • W3C CSS3 Color • Change glTF specification from non-normative to normative reference |
| 2018-11-16 | 1.0r2 | Gabby Getz | |

| DATE | RELEASE | AUTHOR | DESCRIPTION |
|------------|---------|--------|--|
| 2019-01-21 | TODO | TODO | <ul style="list-style-type: none">Revise internal link formattingTODO |

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.

3.1. Normative

EPSG: 4979, 2007. <http://spatialreference.org/ref/epsg/wgs-84-3/>

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

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

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

Khronos Group: glTF 2.0 – Runtime 3D Asset Delivery, 2017. <https://github.com/KhronosGroup/glTF/blob/master/README.md>

Roger Lott: OGC 12-063r5, *Geographic information – Well known text representation of coordinate reference systems*. Open Geospatial Consortium (2015). <https://docs.ogc.org/is/12-063r5/12-063r5.html>.

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