

# 3D Tiles Specification

## Table of Contents

1. 3D Tiles Format Specification .....	2
1.1. Introduction .....	2
1.2. File Extensions and Media Types .....	5
1.3. JSON encoding .....	5
1.4. URIs .....	5
1.5. Units .....	6
1.6. Coordinate reference system (CRS) .....	6
1.7. Concepts .....	6
1.8. Tile format specifications .....	46
1.9. Declarative styling specification .....	47
2. Tile Formats .....	47
2.1. Legacy Tile Formats .....	47
2.2. glTF .....	48
2.3. Batch Table .....	53
2.4. Feature Table .....	57
2.5. Batched 3D Model .....	59
2.6. Instanced 3D Model .....	64
2.7. Point Cloud .....	73
2.8. Composite .....	84
3. Implicit Tiling .....	85
3.1. Overview .....	86
3.2. Implicit Root Tile .....	86
3.3. Subdivision Scheme .....	87
3.4. Tile Coordinates .....	89
3.5. Template URIs .....	91
3.6. Subtrees .....	91
3.7. Subtree JSON Format .....	98
3.8. Subtree Binary Format .....	106
4. 3D Metadata Specification .....	106
4.1. Overview .....	106
4.2. Concepts .....	107
4.3. Schemas .....	108
4.4. Storage Formats .....	116
5. Styling .....	123
5.1. Overview .....	123
5.2. Concepts .....	124

5.3. Expressions .....	127
5.4. Point Cloud .....	149
5.5. File extension and MIME type .....	150
Appendix A: Migration From Legacy Tile Formats .....	151
Appendix B: Availability Indexing .....	152
Appendix C: 3D Metadata Reference Implementation .....	157
Appendix D: 3D Metadata Semantic Reference .....	164
Appendix E: Acknowledgements .....	171
Appendix F: License .....	172



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

# 1. 3D Tiles Format Specification

## 1.1. Introduction

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.

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

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

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

Tiles may also reference the legacy 3D Tiles 1.0 formats listed below. These formats were

deprecated in 3D Tiles 1.1 and may be removed in a future version of 3D Tiles.

Legacy Format	Uses
<a href="#">Batched 3D Model (b3dm)</a>	Heterogeneous 3D models
<a href="#">Instanced 3D Model (i3dm)</a>	3D model instances
<a href="#">Point Cloud (pnts)</a>	Massive number of points
<a href="#">Composite (cmt)</a>	Concatenate tiles of different formats into one tile

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

The content references a set of *features*, such as 3D models representing buildings or trees, or points in a point cloud. Each feature has position and appearance properties and additional application-specific properties. A client may choose to select features at runtime and retrieve their properties for visualization or analysis.

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

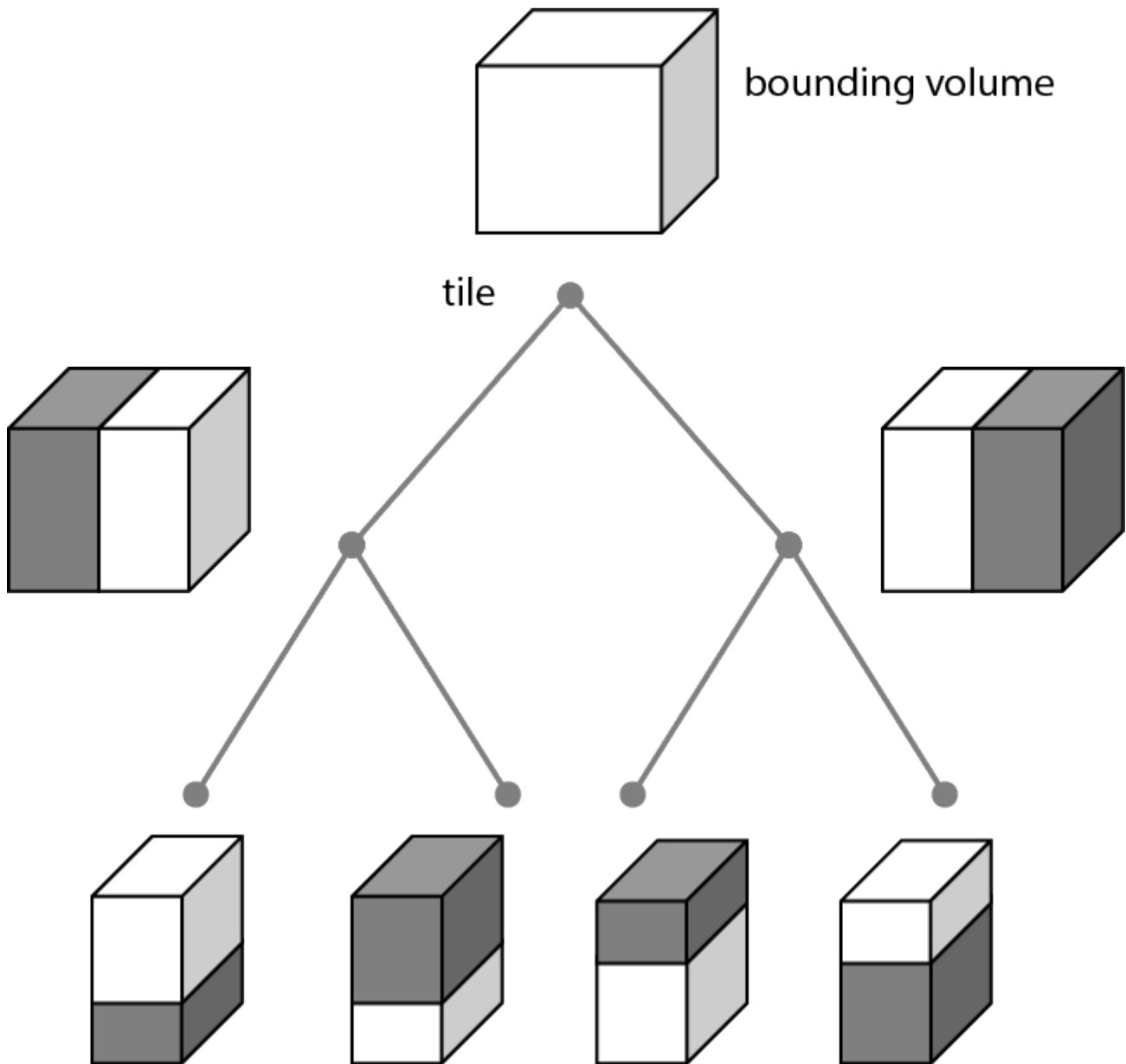


Figure 1. A tree of tiles

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

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

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

## 1.2. File Extensions and Media Types

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

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

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

## 1.3. 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, without JSON escaping.
3. Non-ASCII characters that appear as property *values* in JSON may be escaped.
4. Names (keys) within JSON objects must be unique, i.e., duplicate keys aren't allowed.
5. Some properties are defined as integers in the schema. Such values may be stored as decimals with a zero fractional part or by using exponent notation, as defined in [RFC 8259, Section 6](#).

## 1.4. 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" URL scheme \(RFC2397\)](#).

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

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

## 1.5. Units

The unit for all linear distances is meters.

All angles are in radians.

## 1.6. Coordinate reference system (CRS)

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

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

An additional [tile transform](#) may be applied to transform a tile's local coordinate system to the parent tile's coordinate system.

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

## 1.7. Concepts

### 1.7.1. Tiles

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

#### Tile Content

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

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

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

Each content can be associated with a bounding volume. While the `tile.boundingBox` is a bounding volume encloses *all* contents of the tile, each individual `content.boundingBox` is a

tightly fit bounding volume enclosing just the respective content. More details about the role of tile- and content bounding volumes are given in the [bounding volume](#) section.

## Geometric error

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

A tile's geometric error defines the selection metric for that tile. Its value is a nonnegative number that specifies the error, in meters, of the tile's simplified representation of its source geometry. Generally, the root tile will have the largest geometric error, and each successive level of children will have a smaller geometric error than its parent, with leaf tiles having a geometric error of or close to 0.

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

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

## Refinement

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

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

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

## Replacement

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

Parent Tile	Refined
	




**Additive**

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

Parent Tile	Refined
	

**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	Bounding sphere	Bounding region
		



## 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 2. A bounding region

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

## 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.

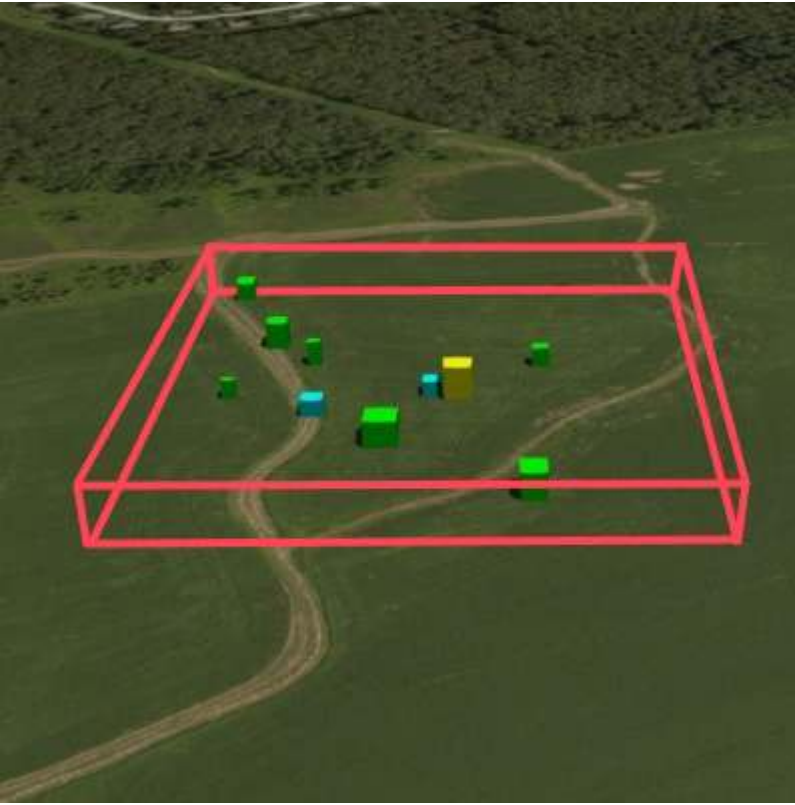


Figure 3. A bounding box

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

### Sphere

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



Figure 4. A bounding sphere

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

### Content Bounding Volume

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

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

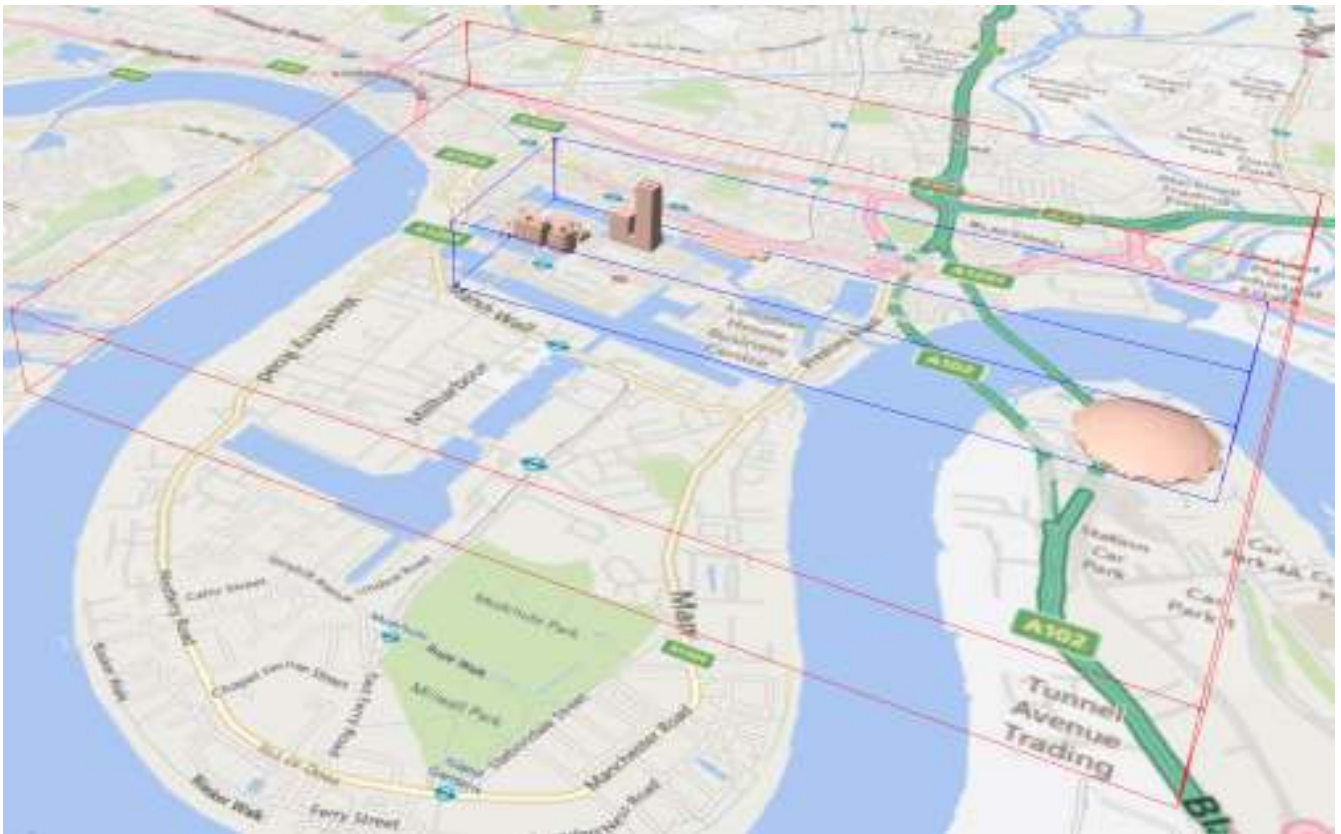


Figure 5. Bounding volumes for the root tile of a tileset. Building data from [CyberCity3D](#). Imagery data from [Bing Maps](#)

## Extensions

Other bounding volume types are supported through extensions.

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

## Viewer request volume

A tile's `viewerRequestVolume` can be used for combining heterogeneous datasets, and can be combined with [external tilesets](#).

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

```

{
  "children": [{
    "transform": [
      4.843178171884396, 1.2424271388626869, 0, 0,
      -0.7993325488216595, 3.1159251367235608, 3.8278032889280675, 0,
      0.9511533376784163, -3.7077466670407433, 3.2168186118075526, 0,
      1215001.7612985559, -4736269.697480114, 4081650.708604793, 1
    ],
    "boundingVolume": {
      "box": [
        0, 0, 6.701,
        3.738, 0, 0,
        0, 3.72, 0,
        0, 0, 13.402
      ]
    },
    "geometricError": 32,
    "content": {
      "uri": "building.glb"
    }
  }, {
    "transform": [
      0.968635634376879, 0.24848542777253732, 0, 0,
      -0.15986650990768783, 0.6231850279035362, 0.7655606573007809, 0,
      0.19023066741520941, -0.7415493329385225, 0.6433637229384295, 0,
      1215002.0371330238, -4736270.772726648, 4081651.6414821907, 1
    ],
    "viewerRequestVolume": {
      "sphere": [0, 0, 0, 15]
    },
    "boundingVolume": {
      "sphere": [0, 0, 0, 1.25]
    },
    "geometricError": 0,
    "content": {
      "uri": "points.glb"
    }
  }
]}
}

```

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

## Transforms

### 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 4x4 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 3x3 matrix of the inverse-transpose of `transform` to account for [correct vector transforms when scale is used](#).
  - `content.boundingBox`, except when `content.boundingBox.region` is defined, which is explicitly in EPSG:4979 coordinates.
- `tile.boundingBox`, except when `tile.boundingBox.region` is defined, which is explicitly in EPSG:4979 coordinates.
- `tile.viewerRequestVolume`, except when `tile.viewerRequestVolume.region` is defined, which is explicitly in EPSG:4979 coordinates.

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

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

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

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

### glTF transforms

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

### glTF node hierarchy

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

### 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. [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.

#### **NOTE**

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"
  }
]
```

#### **Example**

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



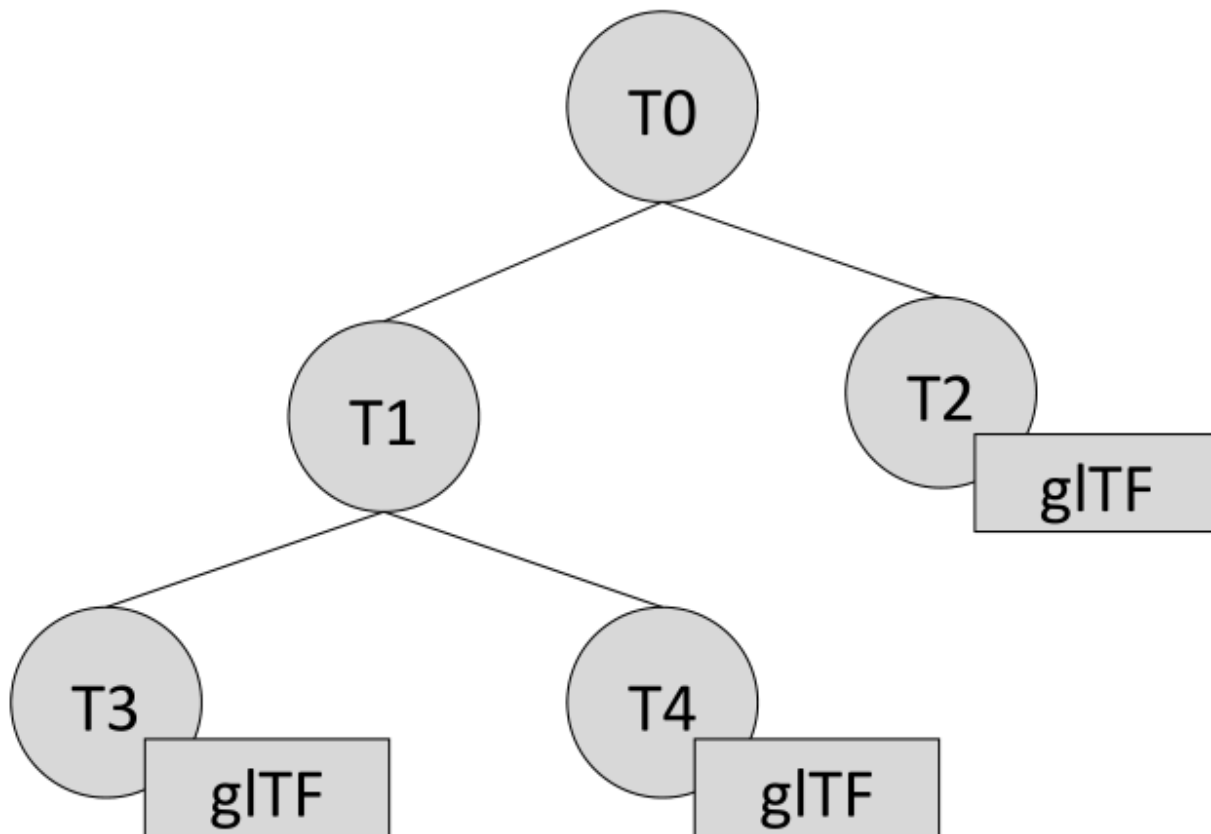


Figure 6. Structure of an example tileset with tiles that contain glTF content

The computed transform for each tile is:

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

The full computed transforms, taking into account the [glTF y-up to z-up transform](#) and [glTF Transforms](#) are

- T0: [T0]
- T1: [T0][T1]
- T2: [T0][T2][glTF y-up to z-up][glTF transform]
- T3: [T0][T1][T3][glTF y-up to z-up][glTF transform]
- T4: [T0][T1][T4][glTF y-up to z-up][glTF transform]

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

  computeTransform(root, transformToRoot);
}

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

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

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

## Tile JSON

A tile JSON object consists of the following properties.

# tile

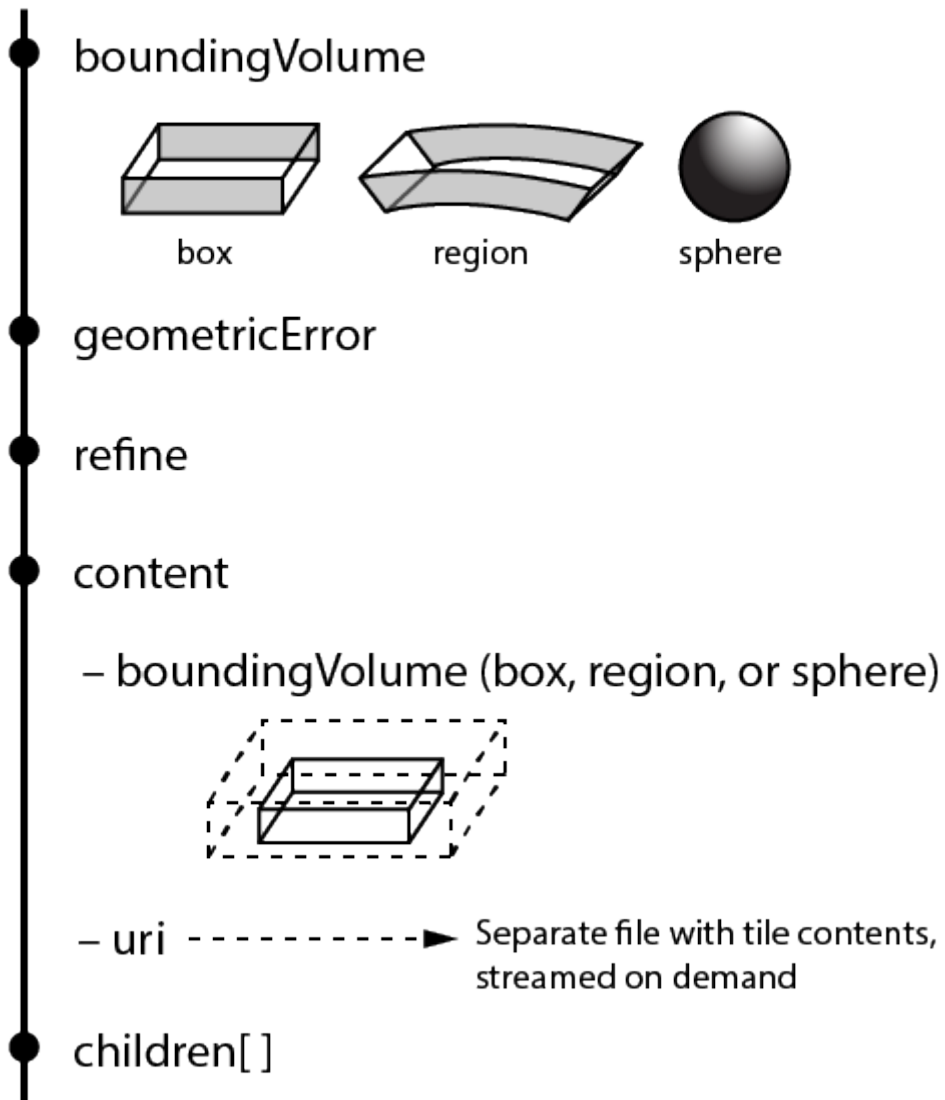


Figure 7. Elements of a tile JSON object

The following example shows one non-leaf tile.

```

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

```

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

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

The optional `viewerRequestVolume` property (not shown above) defines a volume, using the same schema as `boundingVolume`, that the viewer 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 [Refinement](#). It is required for the root tile of a tileset; it is optional for all other tiles. A tileset can use any combination of additive and replacement refinement. When the `refine` property is omitted, it is inherited from the parent tile.

The `content` property is an object that describes the [tile content](#). A file extension is not required for

`content.uri`. A content's `tile format` can be identified by the `magic` field in its header, or else as an external tileset if the content is JSON.

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

It is also possible to define multiple contents for a tile: The `contents` property (not shown above) is an array containing one or more contents. `contents` and `content` are mutually exclusive. When a tile has a single content it should use `content` for backwards compatibility with engines that only support 3D Tiles 1.0. Multiple contents allow for different representations of the tile content — for example, one as a triangle mesh and one as a point cloud:

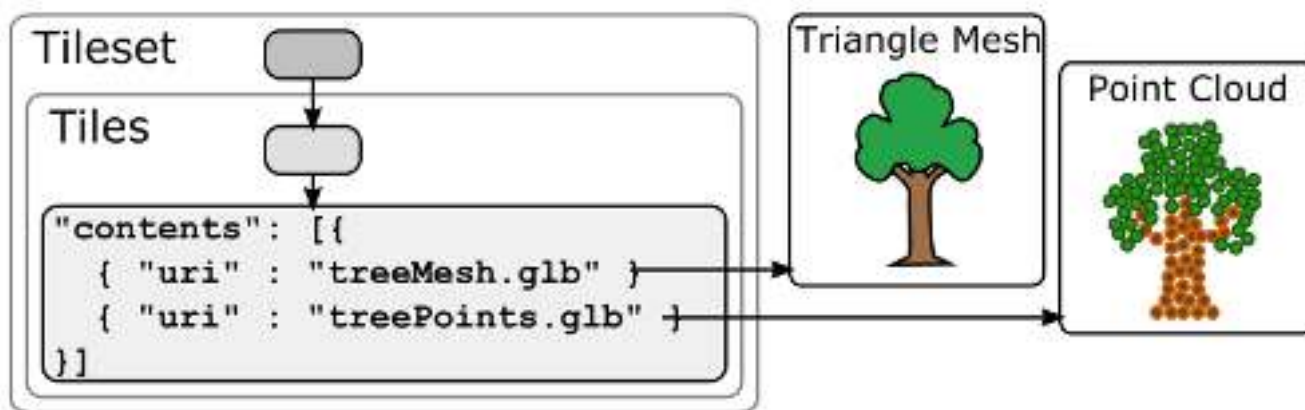


Figure 8. An example of a tile that defines multiple contents

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

```

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

```

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

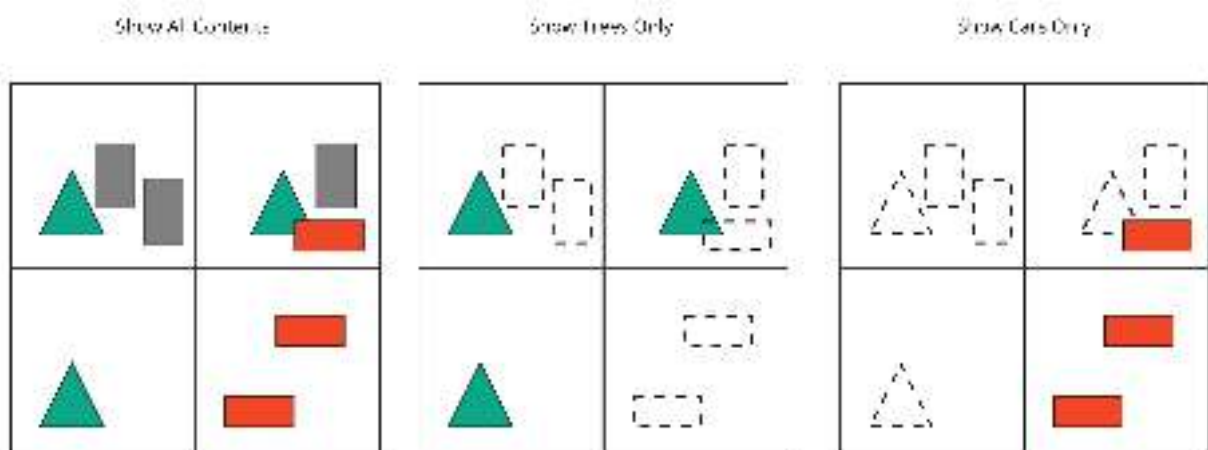


Figure 9. Illustration of rendering options based on content groups

The optional `transform` property (not shown above) defines a 4x4 affine transformation matrix that transforms the tile's `content`, `boundingVolume`, and `viewerRequestVolume` as described in the `Tile transform` section.

The optional `implicitTiling` property (not shown above) defines how the tile is subdivided and

where to locate content resources. See [Implicit Tiling](#).

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

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

### 1.7.2. Tileset JSON

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

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

```

{
  "asset" : {
    "version": "1.1",
    "tilesetVersion": "e575c6f1-a45b-420a-b172-6449fa6e0a59",
  },
  "properties": {
    "Height": {
      "minimum": 1,
      "maximum": 241.6
    }
  },
  "geometricError": 494.50961650991815,
  "root": {
    "boundingVolume": {
      "region": [
        -0.0005682966577418737,
        0.8987233516605286,
        0.00011646582098558159,
        0.8990603398325034,
        0,
        241.6
      ]
    },
    "geometricError": 268.37878244706053,
    "refine": "ADD",
    "content": {
      "uri": "0/0/0.glb",
      "boundingVolume": {
        "region": [
          -0.0004001690908972599,
          0.8988700116775743,
          0.00010096729722787196,
          0.8989625664878067,
          0,
          241.6
        ]
      }
    }
  },
  "children": [...]
}

```

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

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

### Implementation Note

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

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

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

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

### External tilesets

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



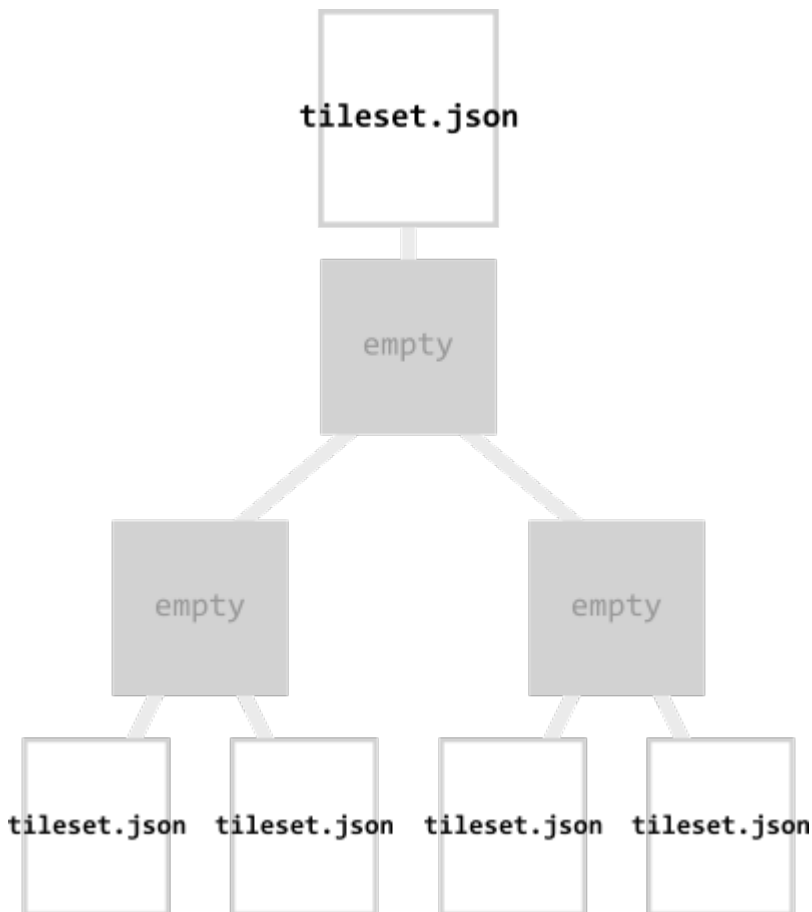


Figure 10. A tileset that refers to other tilesets

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

- Cannot have any children; `tile.children` 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]`.

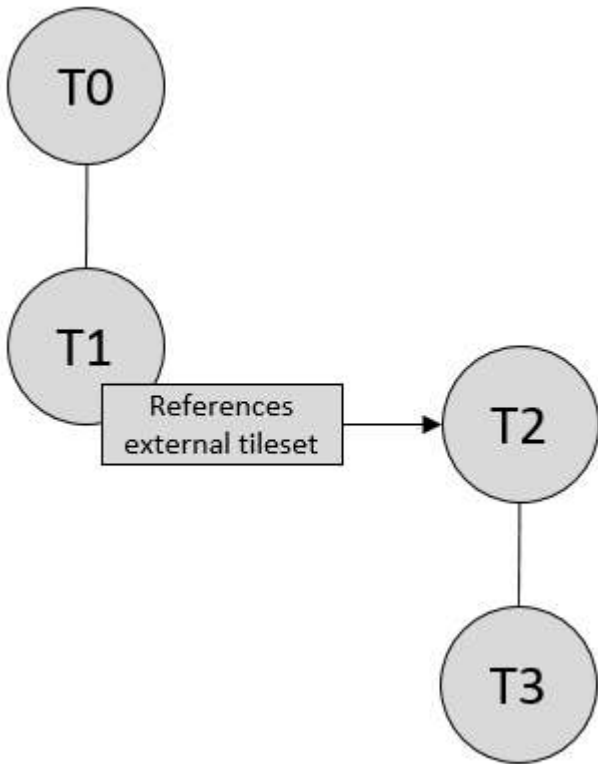
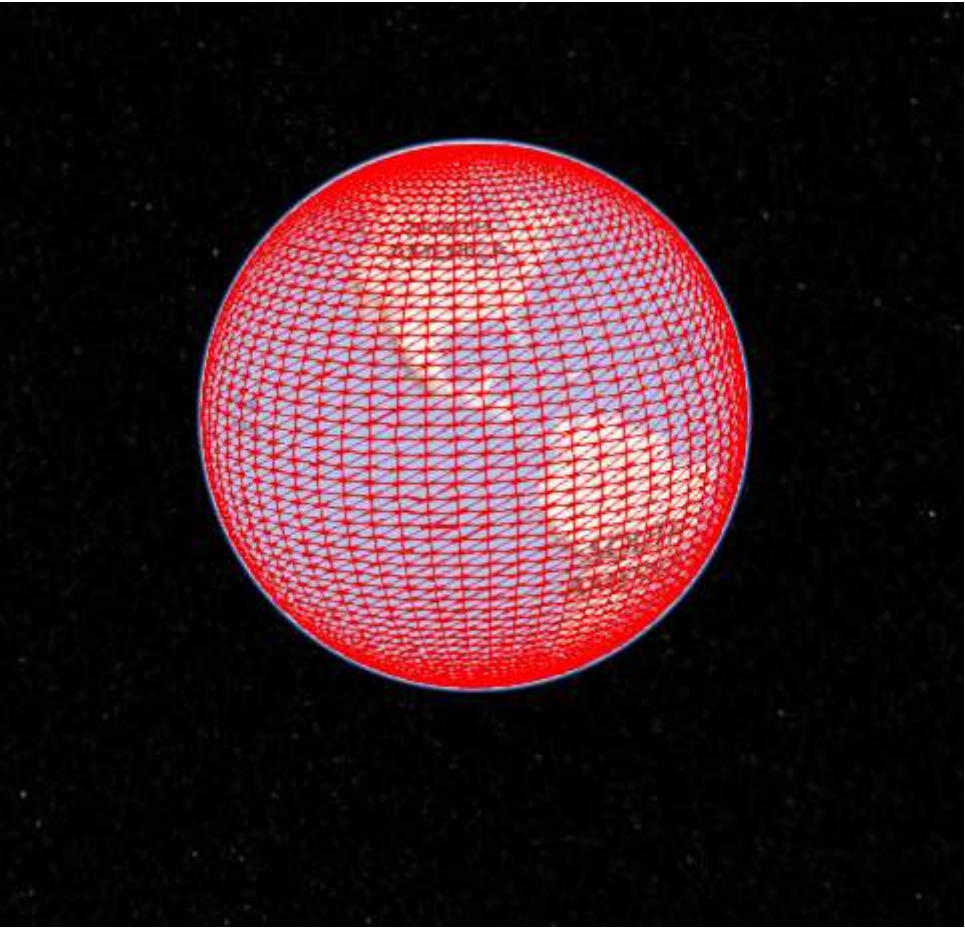


Figure 11. The chain of transforms for a tileset that refers to another tileset

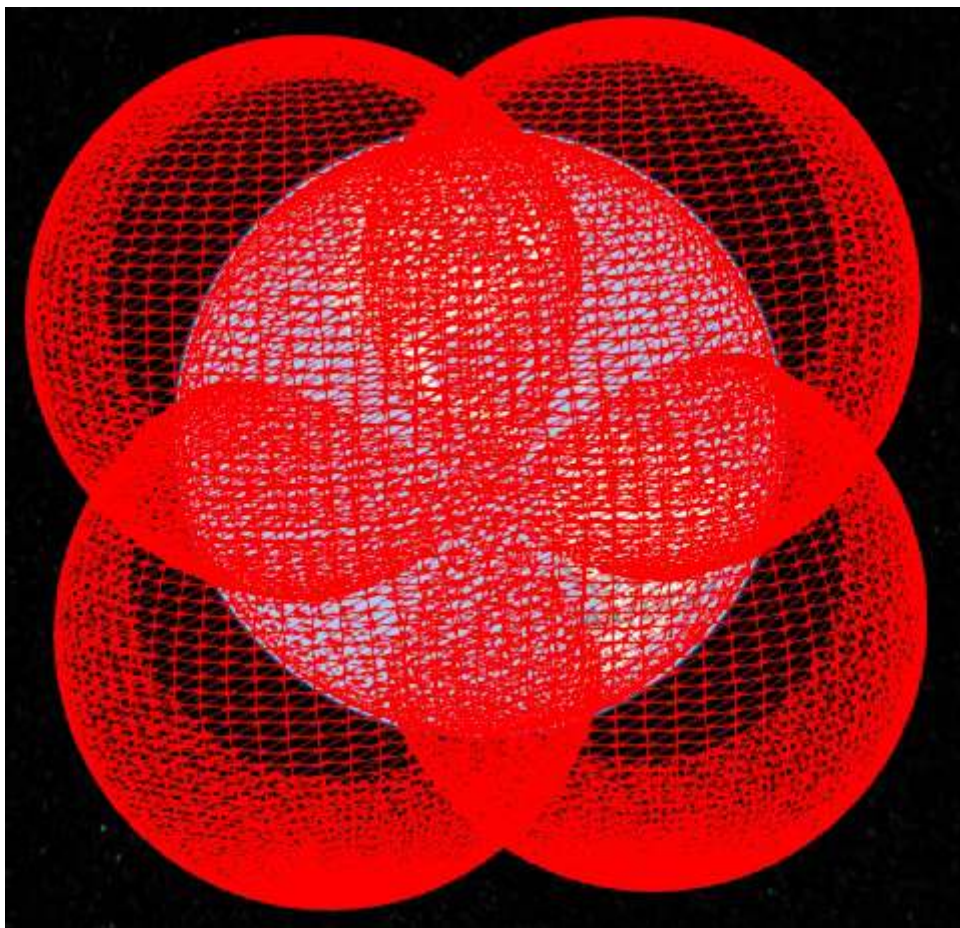
If an external tileset defines `asset.tilesetVersion`, this overrides the value from the parent tileset. If the external tileset does not define `asset.tilesetVersion`, the value is inherited from the parent tileset (if defined).

### Bounding volume spatial coherence

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



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



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

## **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.

### **Quadtrees**

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

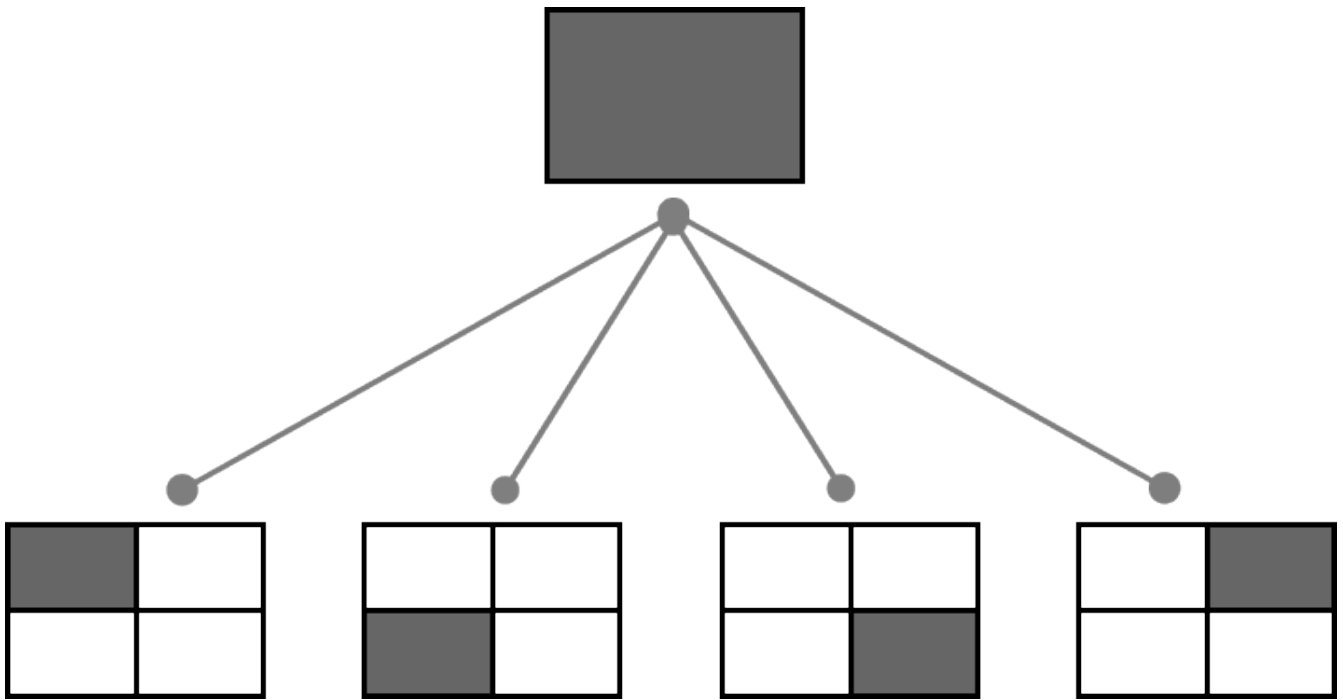


Figure 14. Classic quadtree subdivision

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

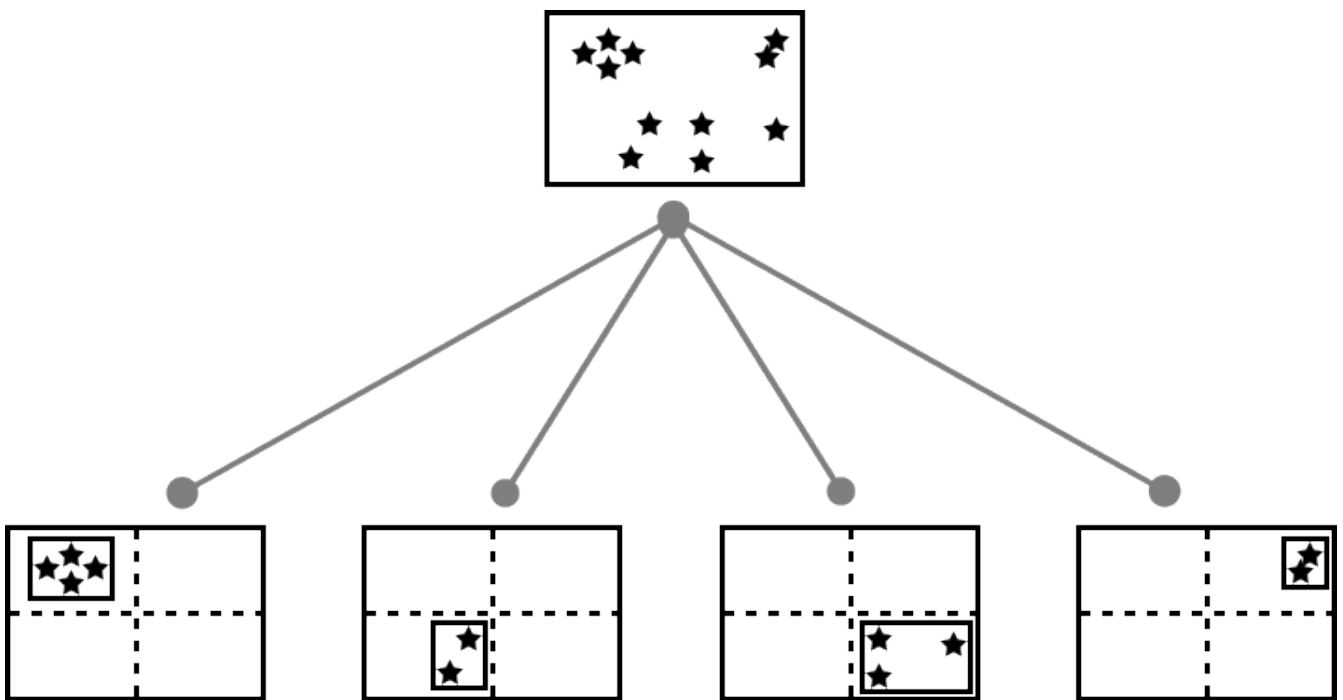


Figure 15. Quadtree with tight bounding volumes around each child

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

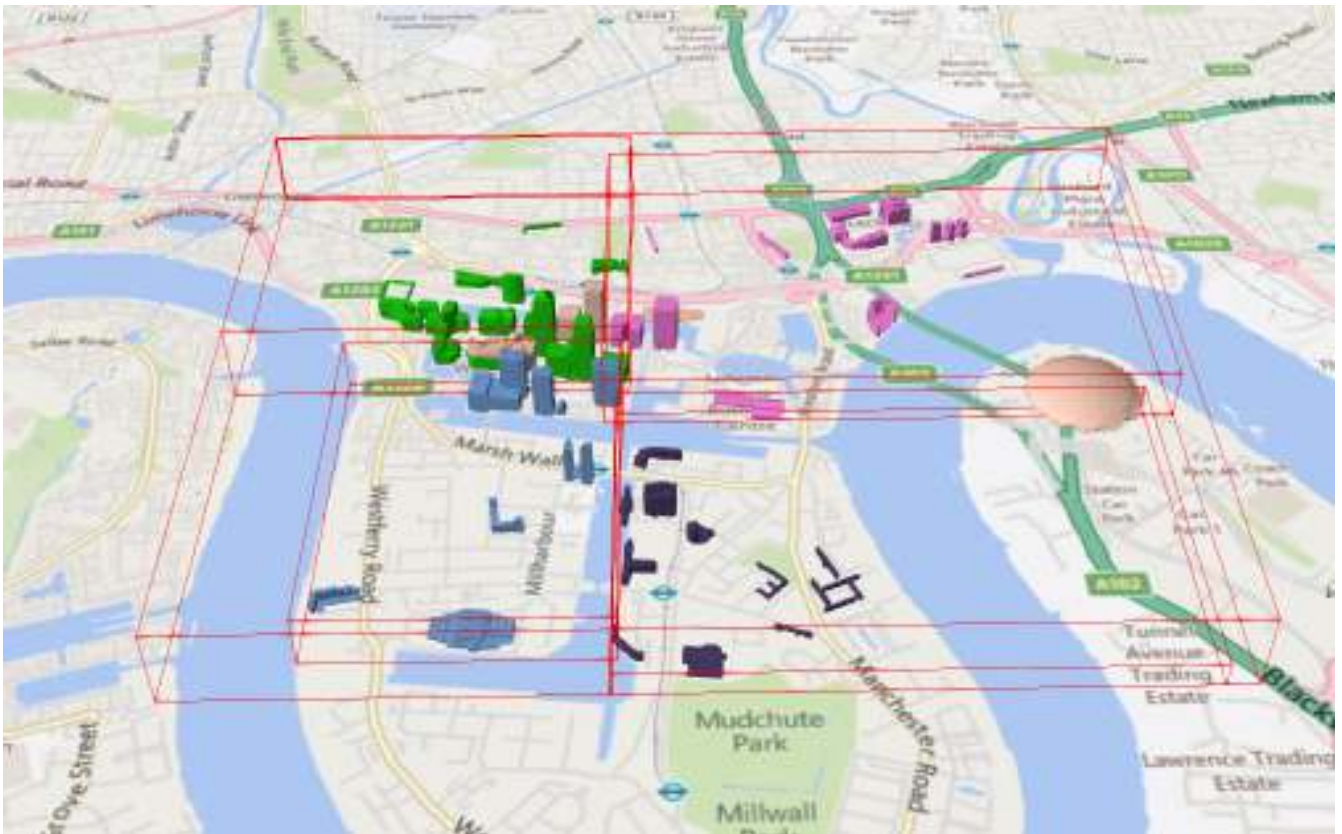


Figure 16. Building data from *CyberCity3D*. Imagery data from *Bing Maps*

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

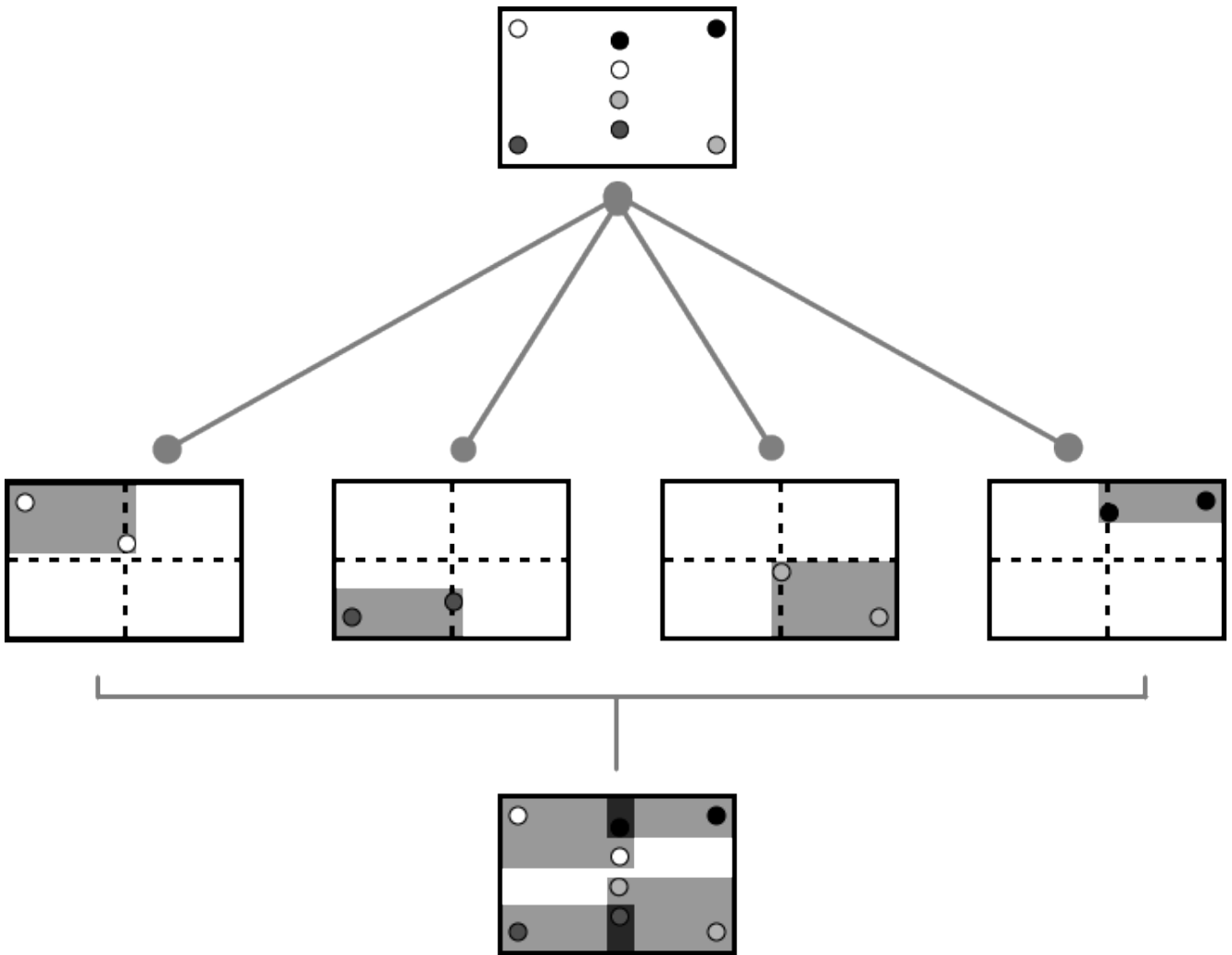


Figure 17. Quadtree with non-uniform and overlapping tiles

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



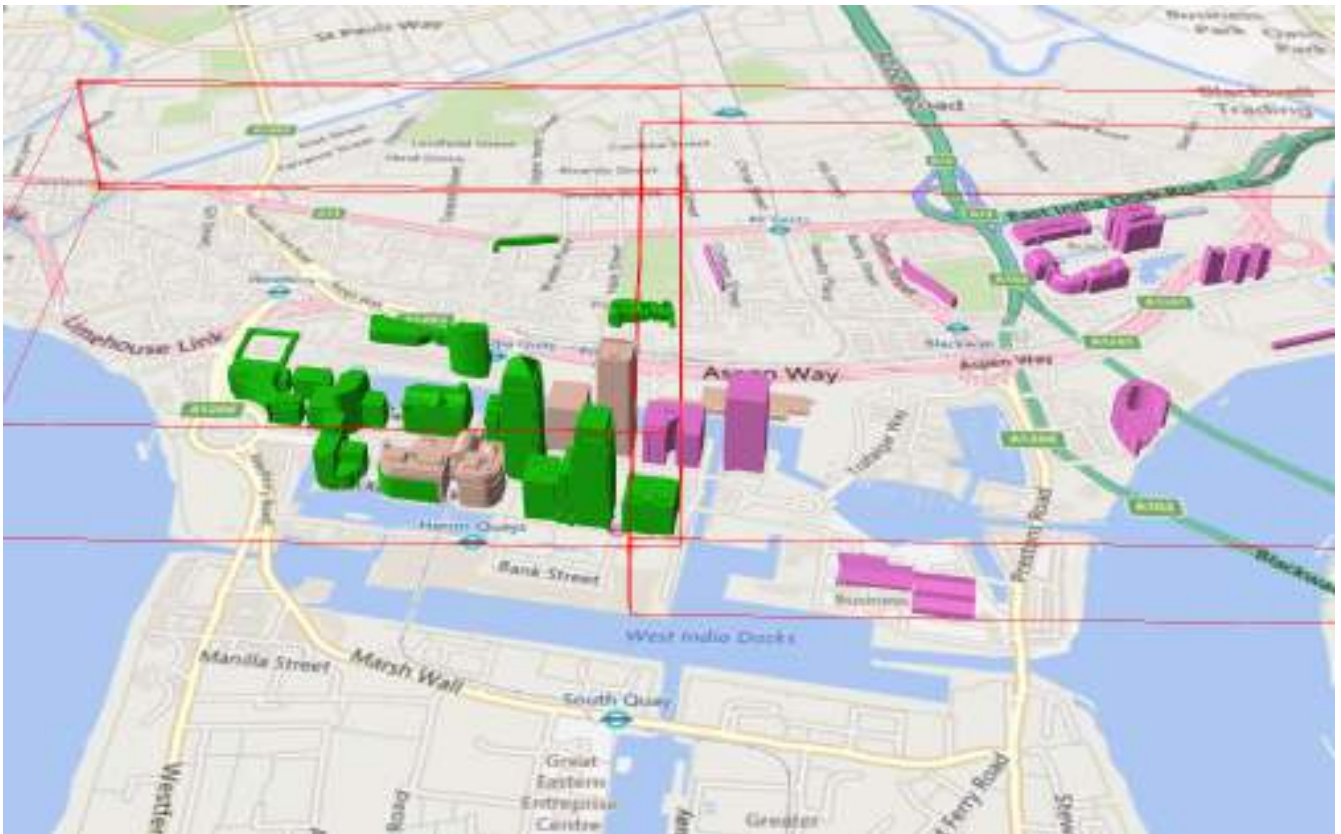


Figure 18. Building data from *CyberCity3D*. Imagery data from *Bing Maps*

### K-d trees

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



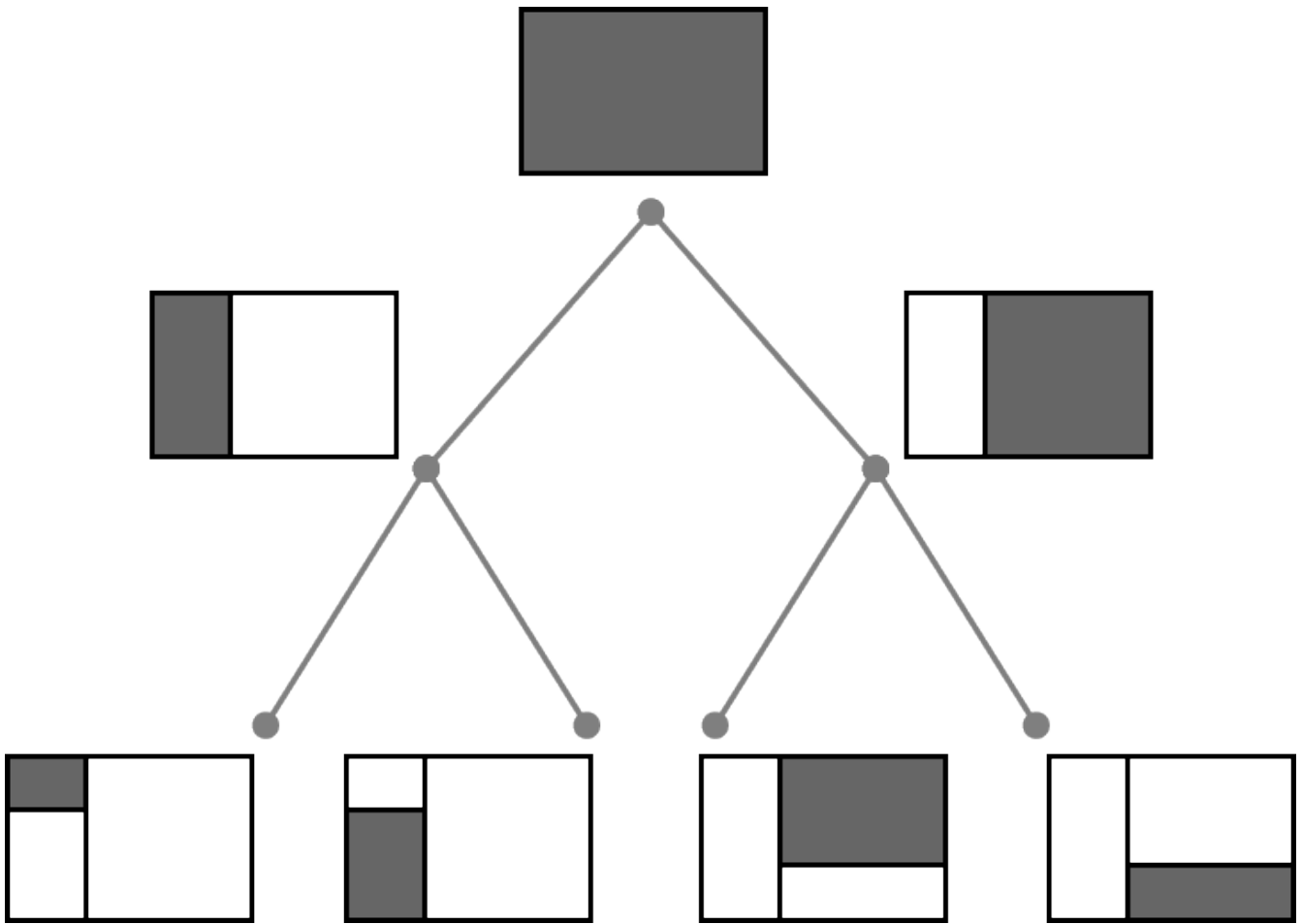


Figure 19. Example k-d tree. Note the non-uniform subdivision

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

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

### Octrees

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

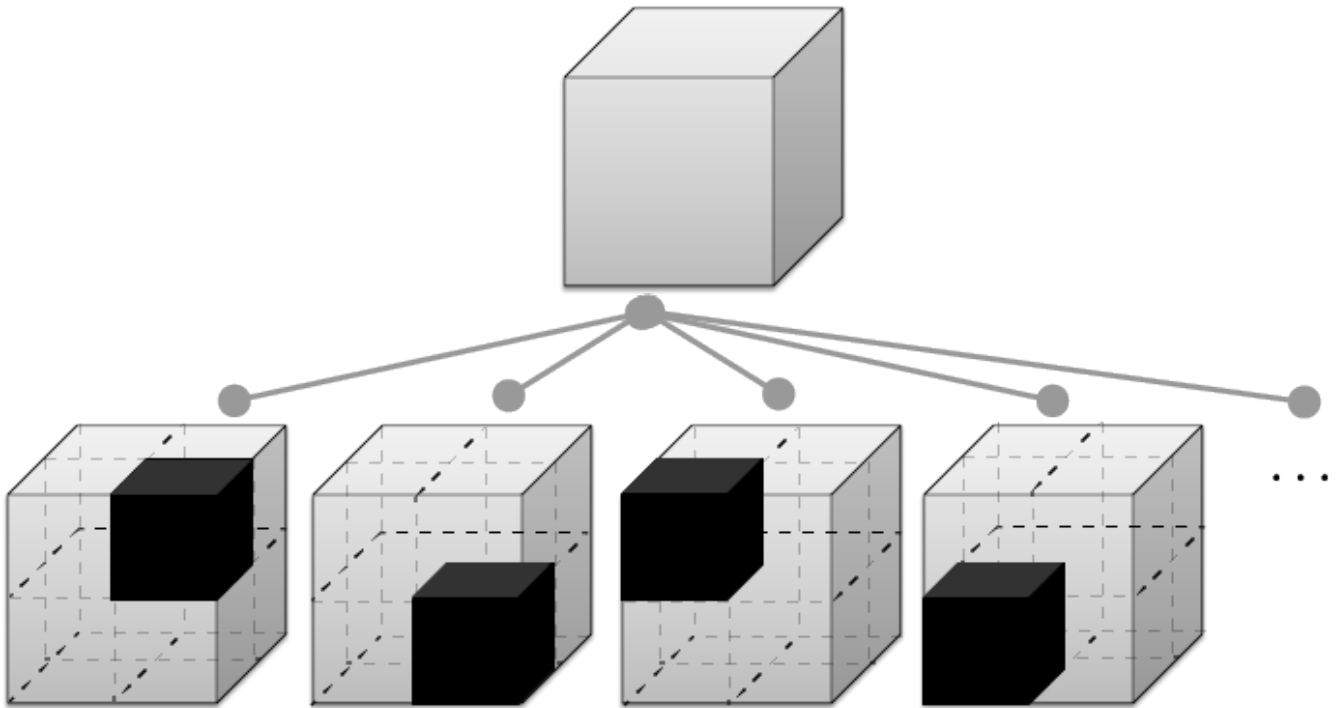


Figure 20. Traditional octree subdivision

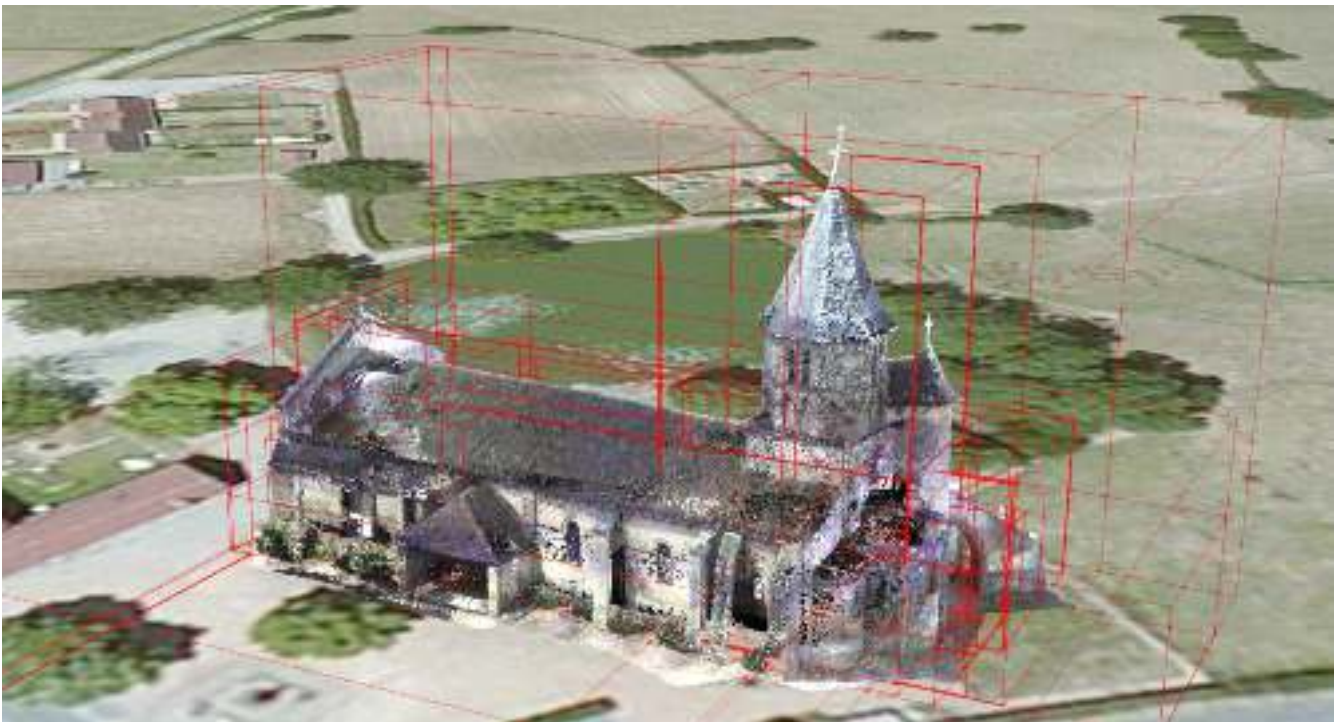


Figure 21. Non-uniform octree subdivision for a point cloud using additive refinement. Point Cloud of [the Church of St Marie at Chappes, France](#) by Prof. Peter Allen, Columbia University Robotics Lab. Scanning by Alejandro Troccoli and Matei Ciocarlie.

### Grids

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



Figure 22. Building data from *CyberCity3D*. Imagery data from *Bing Maps*

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

### Implicit Tiling

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

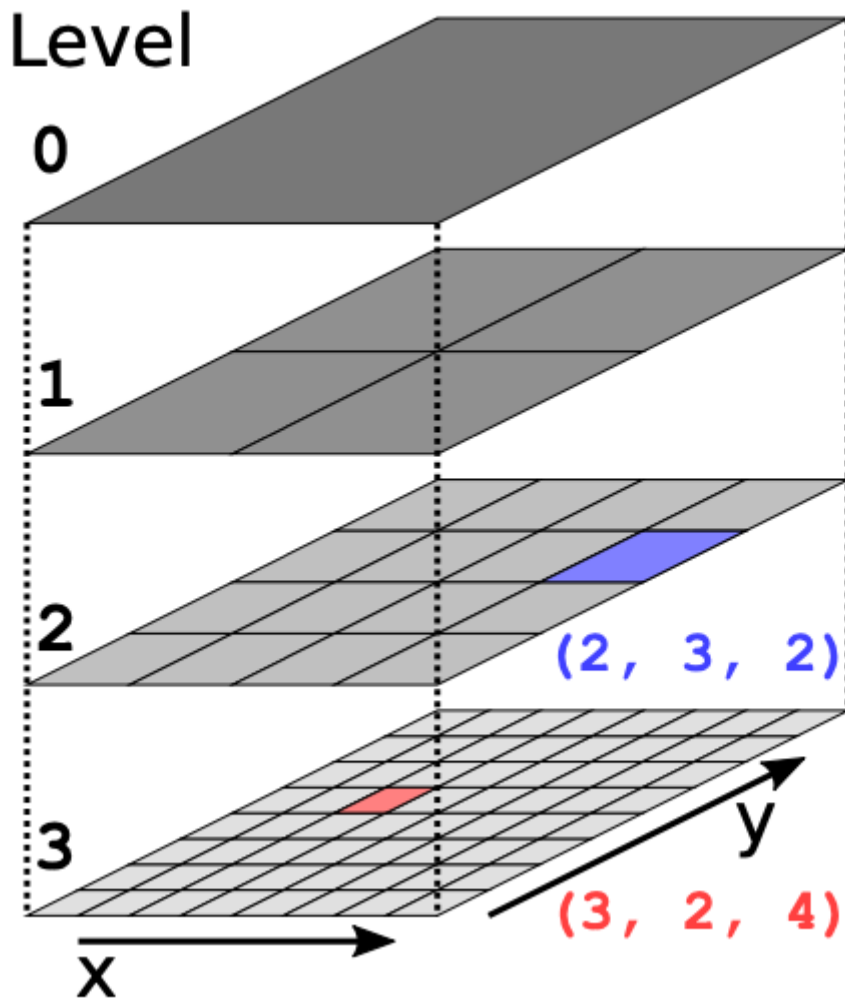


Figure 23. Quadtree with tile coordinates

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

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

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

```

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

```

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

### 1.7.3. Metadata

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

Metadata enables additional use cases and functionality for the format:

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

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

- **Tileset** - The tileset as a whole may be associated with global metadata. Common examples



might include year of collection, author details, or other general context for the tileset contents.

- **Tile** - Tiles may be individually associated with more specific metadata. This may be the timestamp when a tile was last updated or the maximum height of the tile, or spatial hints to optimize traversal algorithms.
- **Groups** - Tile contents may be organized into groups. Each group definition represents a metadata entity that can be assigned to the tile contents by specifying the index within this list as the `group` property of the content. This is useful for working with collections of contents as layers, e.g. to manage visibility or visual styling.
- **Content** - Tile contents may be individually associated with more specific metadata, such as a list of attribution strings.
- **Features** glTF 2.0 assets with feature metadata can be included as tile contents. The `EXT_structural_metadata` extension allows associating metadata with vertices or texels.

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

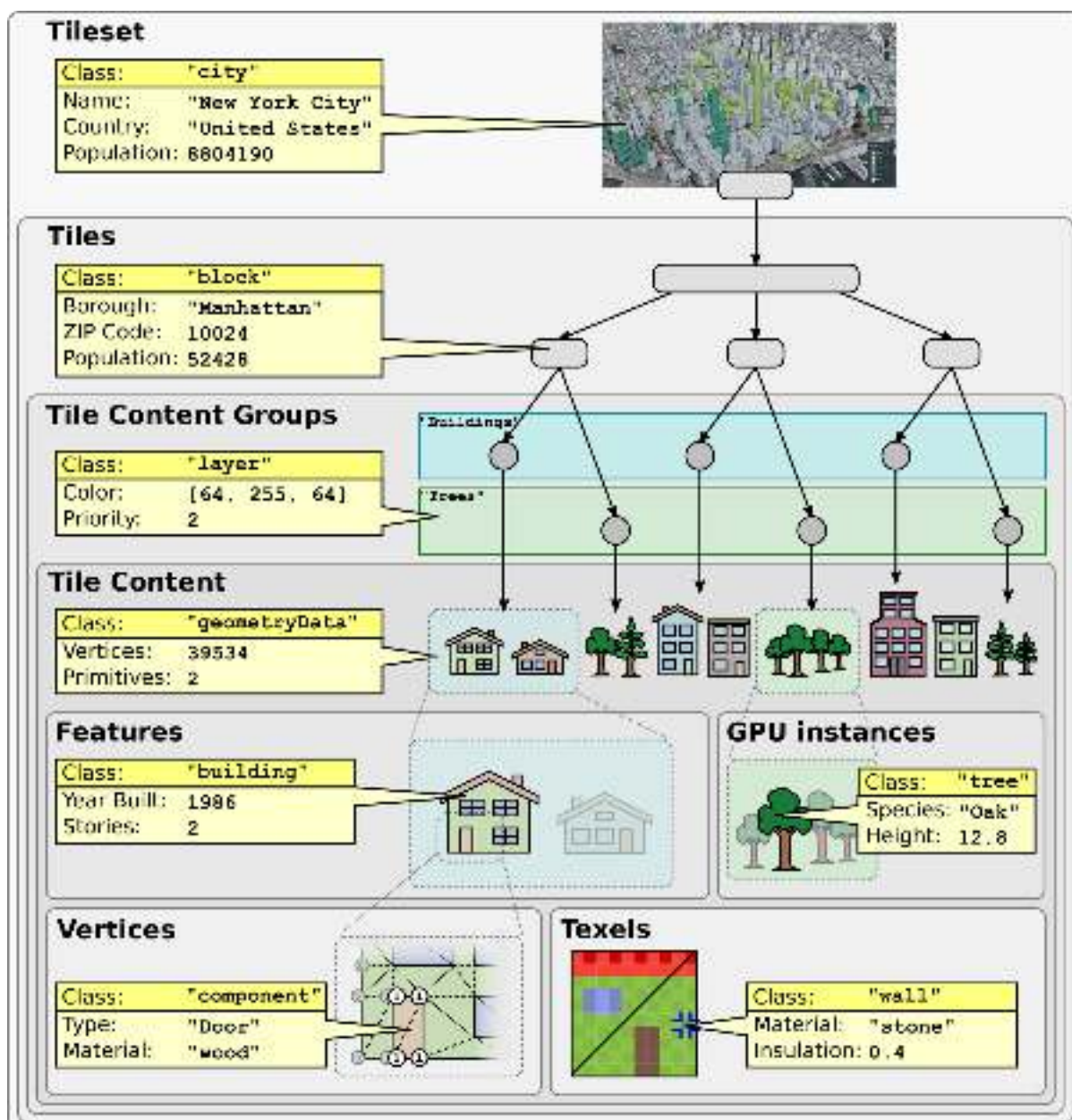


Figure 24. Illustration of the different granularity levels for applying metadata

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

## Metadata Schema

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

Schemas may be embedded in tilesets with the `schema` property, or referenced externally by the `schemaUri` property. Multiple tilesets and glTF contents may refer to the same schema to avoid duplication.

### Example

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

NOTE

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

### Example

External schema referenced by a URI.

NOTE

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



## Assigning Metadata

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

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

### Example

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

### NOTE

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

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

## Metadata Statistics

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

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

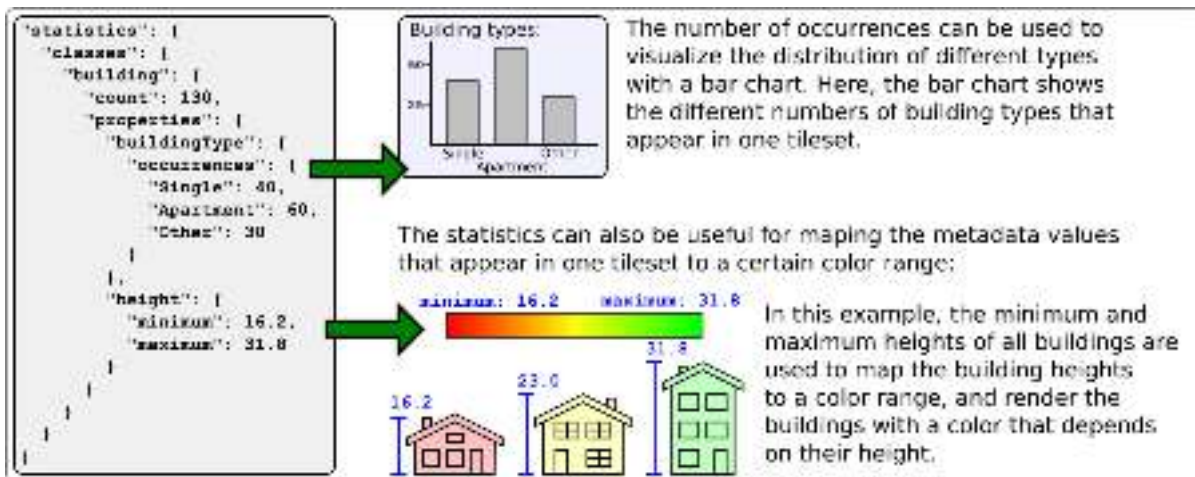


Figure 25. Illustration of how to use metadata statistics for rendering an analytics

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

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

Properties may include the following built-in statistics:

Name	Description	Type
<code>minimum</code>	The minimum property value	Scalars, vector, matrices
<code>maximum</code>	The maximum property value	...
<code>mean</code>	The arithmetic mean of the property values	...
<code>median</code>	The median of the property values	...
<code>standardDeviation</code>	The standard deviation of the property values	...
<code>variance</code>	The variance of the property values	...
<code>sum</code>	The sum of the property values	...
<code>occurrences</code>	Frequencies of value occurrences	Object in which keys are property values (for enums, the enum name), and values are the number of occurrences of that property value

Tileset authors may define their own additional statistics, like `mode` in the example below. Application-specific statistics should use an underscore prefix (\*) and lowerCamelCase for

consistency and to avoid conflicting with future built-in statistics.

### Example

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

NOTE

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

```
    "buildingType": {
      "occurrences": {
        "Residential": 50000,
        "Commercial": 40950,
        "Hospital": 50
      }
    }
  }
}
}
```

### 1.7.4. Specifying extensions and application specific extras

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

#### Extensions

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

```

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

```

All extensions used in a tileset or any descendant external tilesets 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`.

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

The full JSON schema can be found in [tileset.schema.json](#).

## 1.8. Tile format specifications

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

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

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

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

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

Legacy Format	Uses
<a href="#">Batched 3D Model (b3dm)</a>	Heterogeneous 3D models
<a href="#">Instanced 3D Model (i3dm)</a>	3D model instances
<a href="#">Point Cloud (pnts)</a>	Massive number of points
<a href="#">Composite (cmpt)</a>	Concatenate tiles of different formats into one tile

## 1.9. Declarative styling specification

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

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

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

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

For complete details, see the [Declarative Styling](#) specification.

## 2. Tile Formats

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

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

### 2.1. Legacy Tile Formats

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

Legacy Format	Uses
<a href="#">Batched 3D Model (b3dm)</a>	Heterogeneous 3D models
<a href="#">Instanced 3D Model (i3dm)</a>	3D model instances
<a href="#">Point Cloud (pnts)</a>	Massive number of points
<a href="#">Composite (cmpt)</a>	Concatenate tiles of different formats into one tile

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

## 2.2. glTF

### 2.2.1. Overview



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

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

### 2.2.2. Use Cases

#### 3D Models

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

Photogrammetry	3D Buildings
 <p data-bbox="177 1375 750 1449"><i>San Francisco photogrammetry model from Aerometrex in O3DE</i></p>	 <p data-bbox="863 1393 1401 1429"><i>3D buildings from swisstopo in CesiumJS</i></p>

#### Point Clouds

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





Figure 26. 40 billion point cloud from the City of Montreal with ASPRS classification codes (CC-BY 4.0)

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

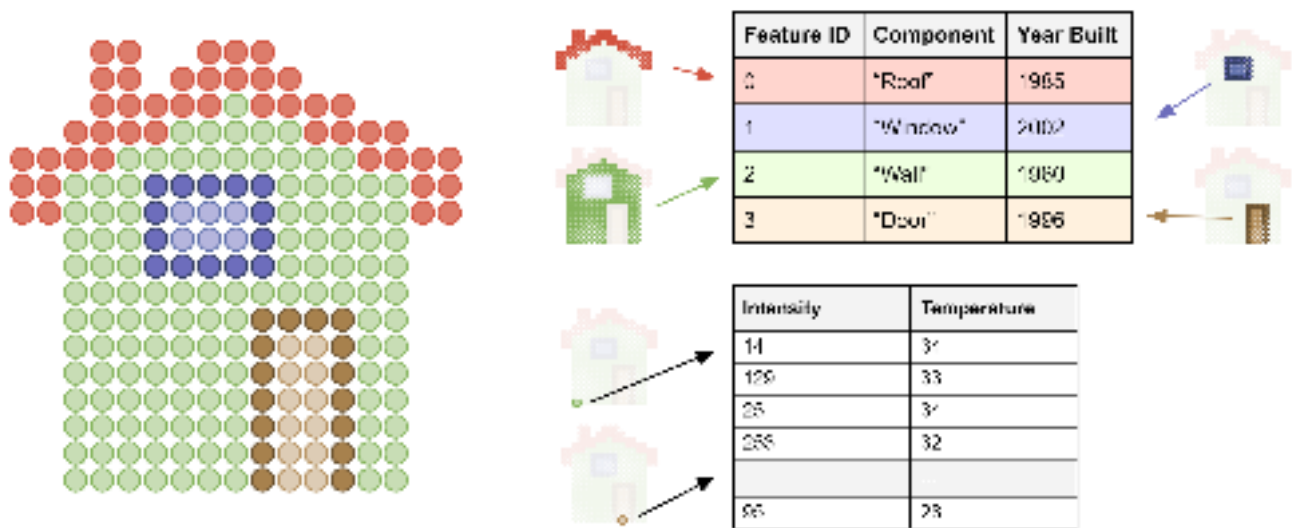


Figure 27. A point cloud with two property tables, one storing metadata for groups of points and the other storing metadata for individual points

### Instancing

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





Figure 28. Instanced tree models in Philadelphia from OpenTreeMap

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

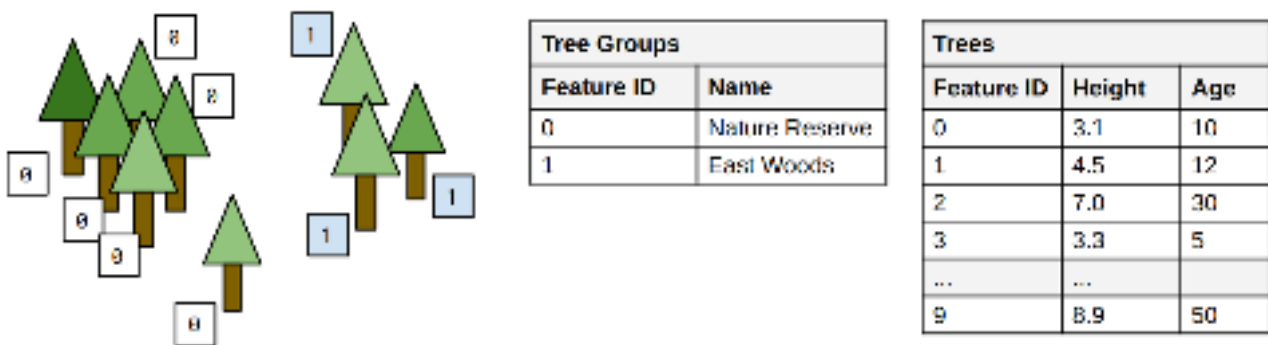
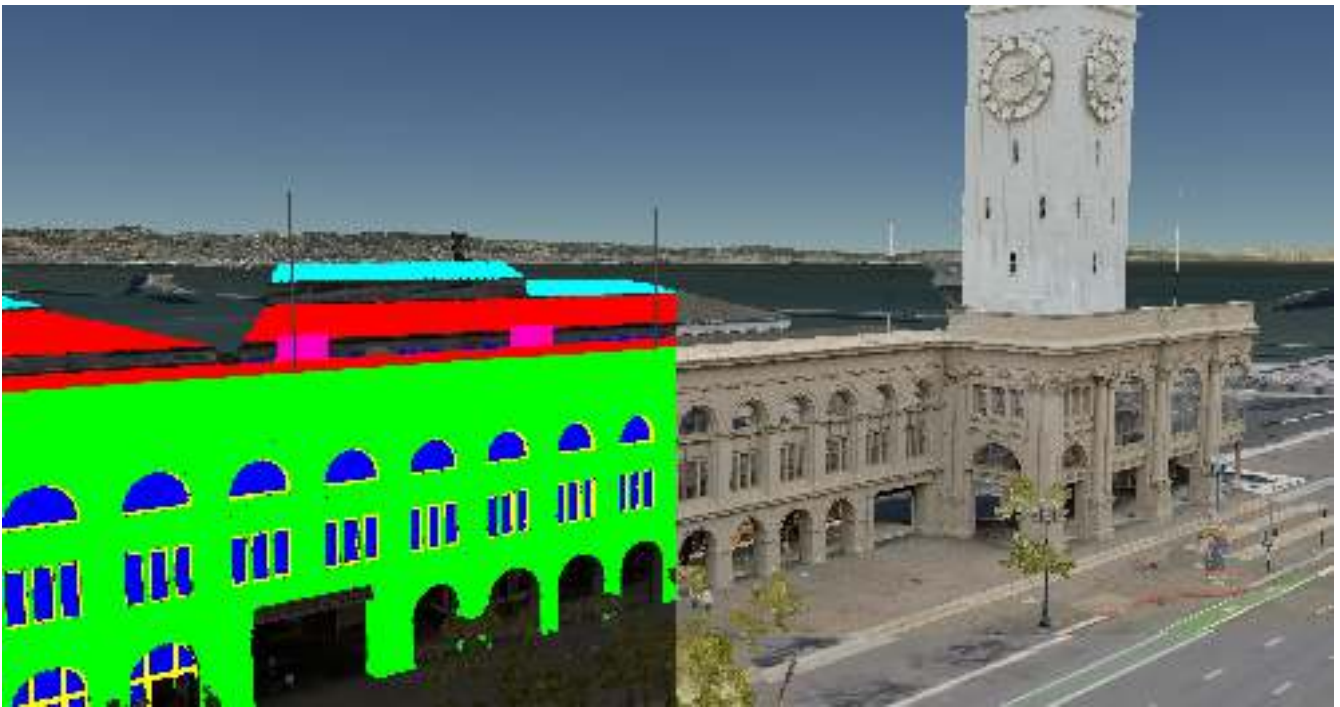


Figure 29. Instanced tree models where trees are assigned to groups with a per-instance feature ID attribute. One feature table stores per-group metadata and the other stores per-tree metadata

### 2.2.3. Feature Identification

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



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

#### **2.2.4. Metadata**

`EXT_structural_metadata` stores metadata at per-vertex, per-texel, and per-feature granularity and uses the type system defined in the [3D Metadata Specification](#). This metadata can be used for visualization and analysis.



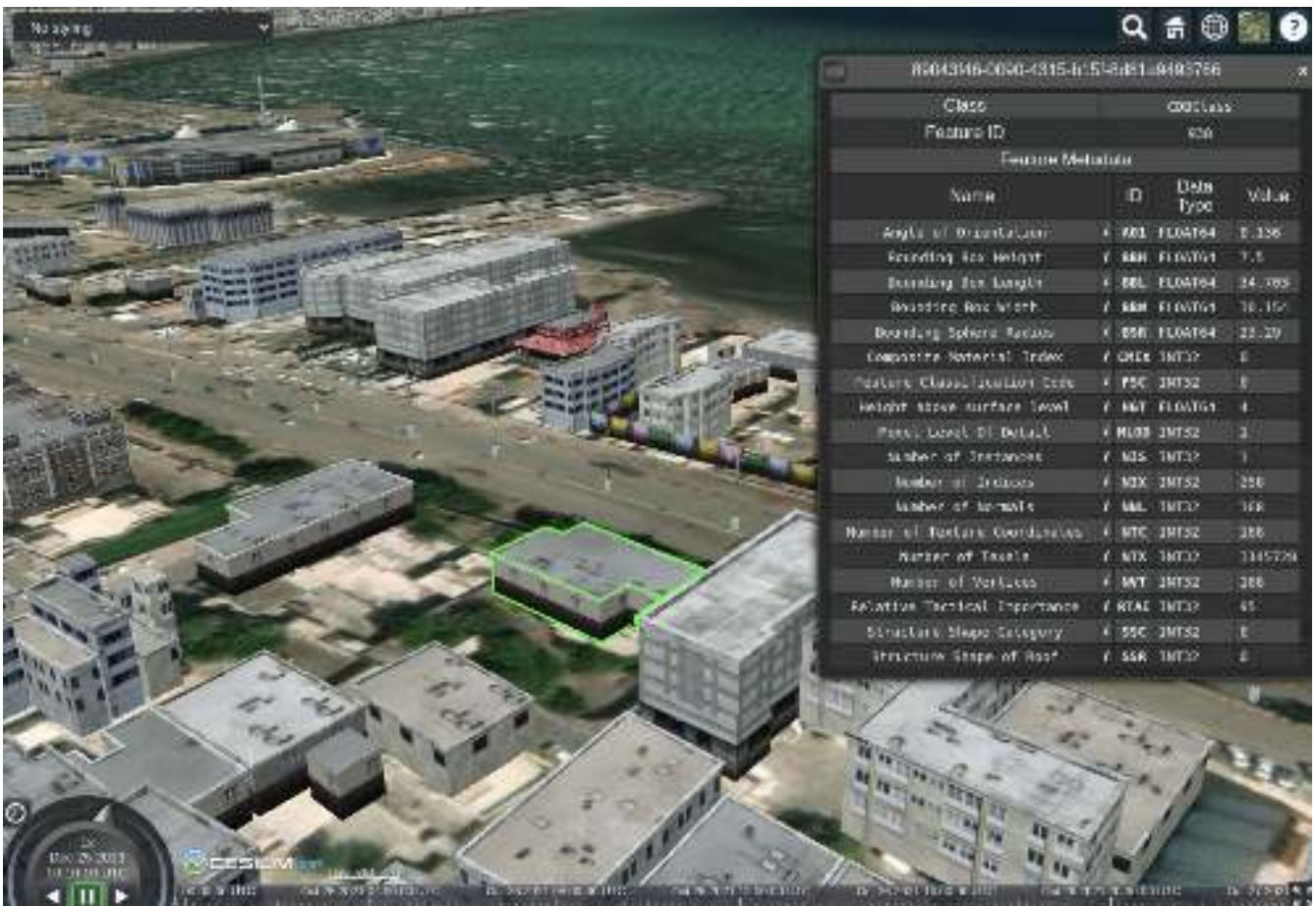


Figure 31. An example of metadata associated with glTF models

## 2.2.5. Compression

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

### Geometry Compression

- [KHR\\_mesh\\_quantization](#)
- [EXT\\_meshopt\\_compression](#)
- [KHR\\_draco\\_mesh\\_compression](#)

### Texture Compression

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

## 2.2.6. File Extensions and Media Types

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

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

## 2.3. Batch Table

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

### 2.3.1. Overview

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

A Batch Table is used by the following tile formats:

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

### 2.3.2. Layout

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

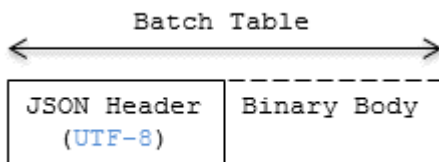


Figure 32. Data layout of a Batch Table

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

#### 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.

## 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*

#### **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'};
```

## Binary body

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

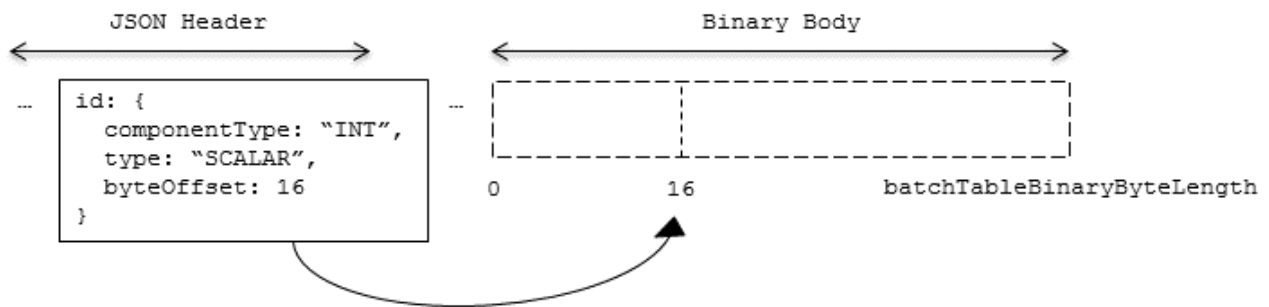


Figure 33. An example showing how to access the binary body, based on the information from the JSON header

Values can be retrieved using the number of features, `batchLength`; the desired batch id, `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.

<code>componentType</code>	Size in bytes
"BYTE"	1
"UNSIGNED_BYTE"	1
"SHORT"	2
"UNSIGNED_SHORT"	2
"INT"	4
"UNSIGNED_INT"	4
"FLOAT"	4
"DOUBLE"	8

<code>type</code>	Number of components
"SCALAR"	1
"VEC2"	2
"VEC3"	3

type	Number of components
"VEC4"	4

### 2.3.3. Extensions

The following extensions can be applied to a Batch Table.

- `3DTILES_batch_table_hierarchy`

### 2.3.4. 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) *
numberOfComponents(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 numberOfComponents = numberOfComponents(type);

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

```

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

## 2.4. Feature Table

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

### 2.4.1. Overview

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

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

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

### 2.4.2. Layout

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

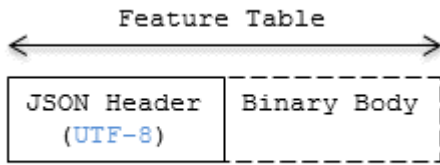


Figure 34. Data layout of a Feature Table

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

## 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.

## JSON header

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

1. A single value or object, e.g., `"INSTANCES_LENGTH" : 4`.
  - This is used for global semantics like `"INSTANCES_LENGTH"`, which defines the number of model instances in an Instanced 3D Model tile.
2. A reference to data in the binary body, denoted by an object with a `byteOffset` property, e.g., `"SCALE" : { "byteOffset" : 24}`.
  - `byteOffset` specifies a zero-based offset relative to the start of the binary body. The value of `byteOffset` 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.

## Binary body

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

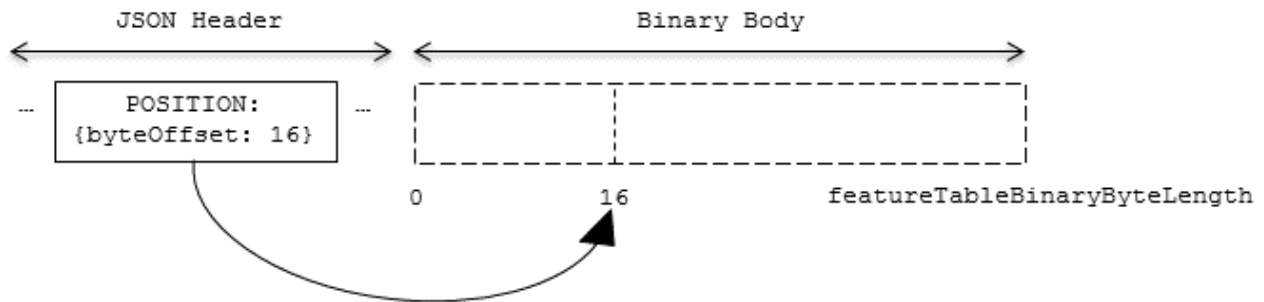


Figure 35. An example showing how to access the binary body, based on the information from the JSON header

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

### 2.4.3. 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 CesiumJS implementation of 3D Tiles.

## 2.5. Batched 3D Model

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

### 2.5.1. Overview

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

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

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

## 2.5.2. Layout

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

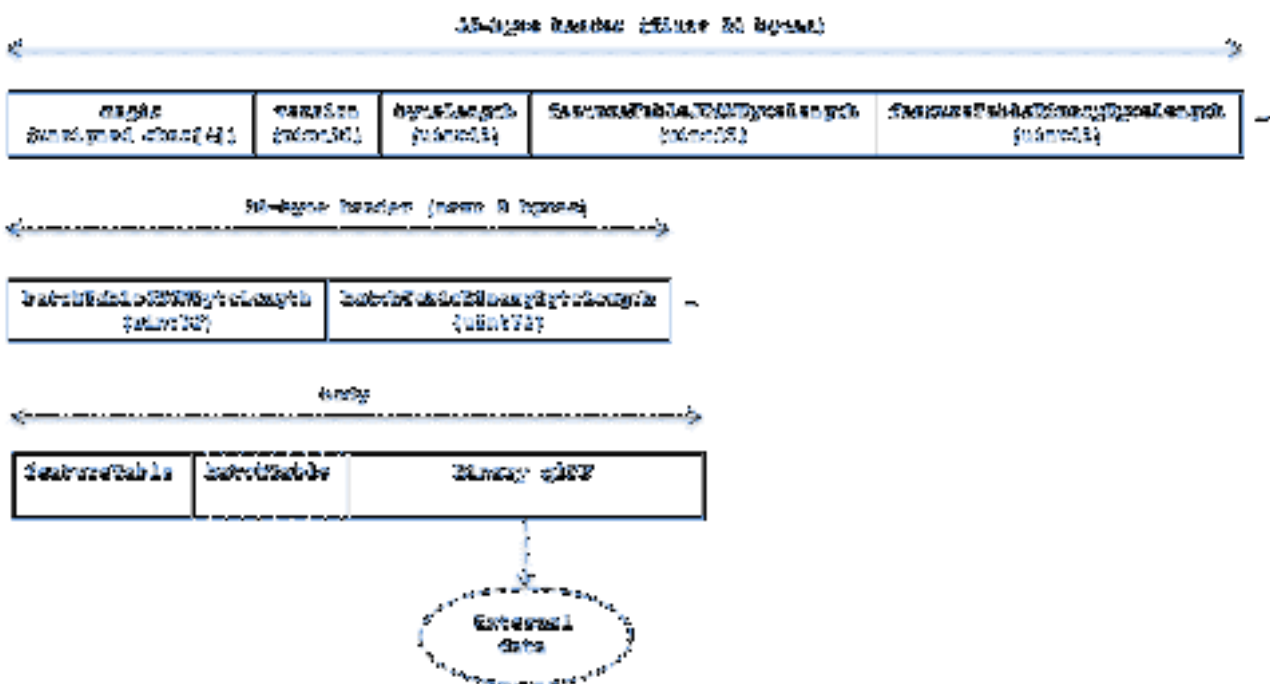


Figure 36. Data layout of a Batched 3D Model

### 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.

## 2.5.3. Header

The 28-byte header contains the following fields:

Field name	Data type	Description
<code>magic</code>	4-byte ANSI string	"b3dm". This can be used to identify the content as a Batched 3D Model tile.

Field name	Data type	Description
<code>version</code>	<code>uint32</code>	The version of the Batched 3D Model format. It is currently <code>1</code> .
<code>byteLength</code>	<code>uint32</code>	The length of the entire tile, including the header, in bytes.
<code>featureTableJSONByteLength</code>	<code>uint32</code>	The length of the Feature Table JSON section in bytes.
<code>featureTableBinaryByteLength</code>	<code>uint32</code>	The length of the Feature Table binary section in bytes.
<code>batchTableJSONByteLength</code>	<code>uint32</code>	The length of the Batch Table JSON section in bytes. Zero indicates there is no Batch Table.
<code>batchTableBinaryByteLength</code>	<code>uint32</code>	The length of the Batch Table binary section in bytes. If <code>batchTableJSONByteLength</code> 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`.

## 2.5.4. Feature Table

Contains values for `b3dm` semantics.

More information is available in the [Feature Table specification](#).

### Semantics

#### Feature semantics

There are currently no per-feature semantics.

#### Global semantics

These semantics define global properties for all features.

Semantic	Data Type	Description	Required
<code>BATCH_LENGTH</code>	<code>uint32</code>	The number of distinguishable models, also called features, in the batch. If the Binary glTF does not have a <code>batchId</code> attribute, this field <i>must</i> be <code>0</code> .	Yes.
<code>RTC_CENTER</code>	<code>float32[3]</code>	A 3-component array of numbers defining the center position when positions are defined relative-to-center, (see <a href="#">Coordinate system</a> ).	No.

## 2.5.5. Batch Table

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

See the [Batch Table](#) reference for more information.

## 2.5.6. Binary glTF

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

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

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

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

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

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

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

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

```
"primitives": [
  {
    "attributes": {
      "_BATCHID": 0
    }
  }
]
```

```

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

```

The `accessor.type` 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.

### 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. Specifically, when the `RTC_CENTER` is defined in the feature table of a Batched 3D Model, the computation of the [tile transform](#) is done as follows:

1. [glTF node hierarchy transformations](#)
2. [glTF y-up to z-up transform](#)
3. The transform for the `RTC_CENTER`, which is used to translate model vertices
4. [Tile transform](#)

### 2.5.7. 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.

### 2.5.8. Implementation example

*This section is non-normative*

Code for reading the header can be found in `Batched3DModelTileContent.js` in the CesiumJS implementation of 3D Tiles.

## 2.6. Instanced 3D Model

### WARNING

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

### 2.6.1. Overview

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

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

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

### NOTE

#### *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.

### NOTE

#### *Implementation Note*

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

### 2.6.2. Layout

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





Field name	Data type	Description
<code>featureTableBinaryByteLength</code>	<code>uint32</code>	The length of the Feature Table binary section in bytes.
<code>batchTableJSONByteLength</code>	<code>uint32</code>	The length of the Batch Table JSON section in bytes. Zero indicates that there is no Batch Table.
<code>batchTableBinaryByteLength</code>	<code>uint32</code>	The length of the Batch Table binary section in bytes. If <code>batchTableJSONByteLength</code> is zero, this will also be zero.
<code>glTFFormat</code>	<code>uint32</code>	Indicates the format of the glTF field of the body. <code>0</code> indicates it is a URI, <code>1</code> indicates it is embedded binary glTF. See the <a href="#">glTF</a> section below.

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

## 2.6.4. Feature Table

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

### Semantics

#### 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
<a href="#">POSITION</a>	<code>float32[3]</code>	A 3-component array of numbers containing <i>x</i> , <i>y</i> , and <i>z</i> Cartesian coordinates for the position of the instance.	Yes, unless <a href="#">POSITION_QUANTIZED</a> is defined.
<a href="#">POSITION_QUANTIZED</a>	<code>uint16[3]</code>	A 3-component array of numbers containing <i>x</i> , <i>y</i> , and <i>z</i> in quantized Cartesian coordinates for the position of the instance.	Yes, unless <a href="#">POSITION</a> is defined.

Semantic	Data Type	Description	Required
NORMAL_UP	float32[3]	A unit vector defining the <b>up</b> direction for the orientation of the instance.	No, unless <b>NORMAL_RIGHT</b> is defined.
NORMAL_RIGHT	float32[3]	A unit vector defining the <b>right</b> direction for the orientation of the instance. Must be orthogonal to <b>up</b> .	No, unless <b>NORMAL_UP</b> is defined.
NORMAL_UP_OCT32P	uint16[2]	An oct-encoded unit vector with 32-bits of precision defining the <b>up</b> direction for the orientation of the instance.	No, unless <b>NORMAL_RIGHT_OCT32P</b> is defined.
NORMAL_RIGHT_OCT32P	uint16[2]	An oct-encoded unit vector with 32-bits of precision defining the <b>right</b> direction for the orientation of the instance. Must be orthogonal to <b>up</b> .	No, unless <b>NORMAL_UP_OCT32P</b> 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 <b>x</b> , <b>y</b> , and <b>z</b> axes of the instance.	No.
BATCH_ID	uint8, uint16 (default), or uint32	The <b>batchId</b> of the instance that can be used to retrieve metadata from the <b>Batch Table</b> .	No.

### 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 <b>POSITION_QUANTIZED</b> 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](#).

### Instance orientation

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

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

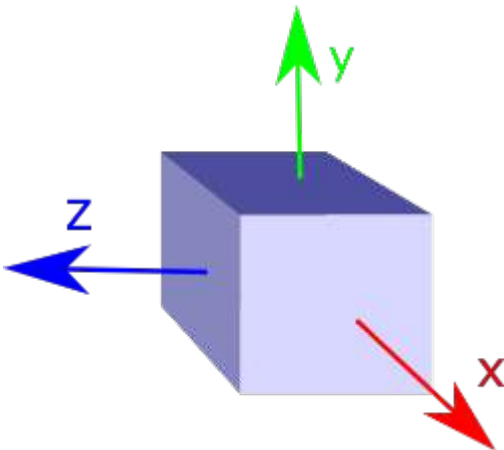


Figure 38. A box in the standard basis

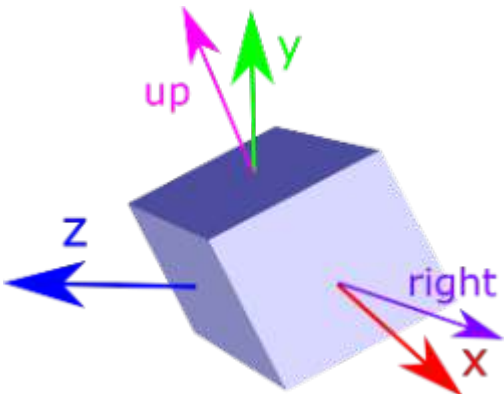


Figure 39. A box transformed into a rotated basis

## 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.

### *Implementation Note*

#### NOTE

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

## 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.

## Instance position

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

### `RTC_CENTER`

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

## Quantized positions

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

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

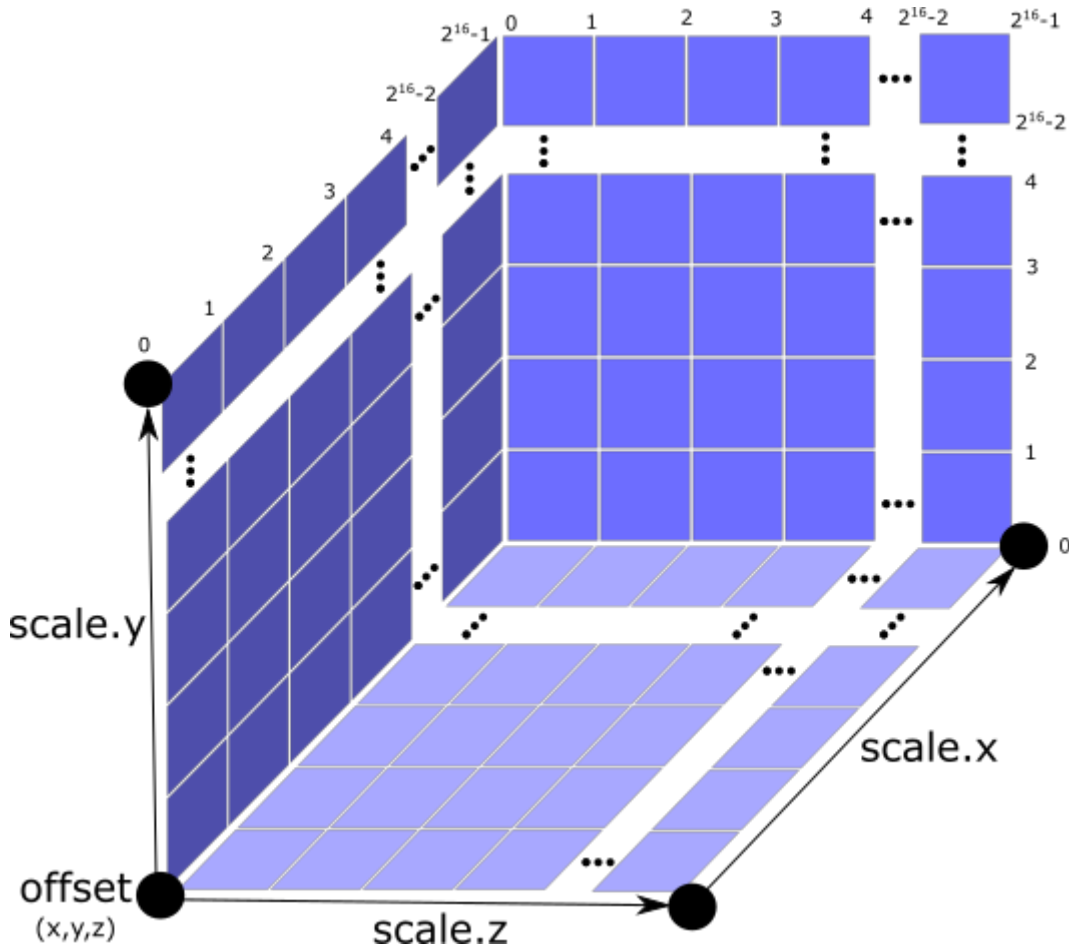


Figure 40. Illustration of the quantization that is used for the `POSITION_QUANTIZED` semantic

`offset` is stored in the global semantic `QUANTIZED_VOLUME_OFFSET`, and `scale` is stored in the global semantic `QUANTIZED_VOLUME_SCALE`. If those global semantics are not defined, `POSITION_QUANTIZED` cannot be used.

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

$$\text{POSITION} = \text{POSITION\_QUANTIZED} * \text{QUANTIZED\_VOLUME\_SCALE} / 65535.0 + \text{QUANTIZED\_VOLUME\_OFFSET}$$

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

### 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.

### Examples

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

#### 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);
```

### 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]);

```

### 2.6.5. Batch Table

Contains metadata organized by `batchId` that can be used for declarative styling. See the [Batch Table](#) reference for more information.

### 2.6.6. glTF

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

The glTF asset to be instanced is stored after the Feature Table and Batch Table. It may embed all of

its geometry, texture, and animations, or it may refer to external sources for some or all of these data.

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

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

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

## 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.

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

1. [glTF node hierarchy transformations](#)
2. [glTF y-up to z-up transform](#)
3. The per-instance positions and scales, as defined in the feature table of the Instanced 3D Model.
4. The transform for the `RTC_CENTER`, which is used to translate model vertices
5. [Tile transform](#)

## 2.6.7. 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.

## 2.7. Point Cloud

**WARNING** | Point Cloud was deprecated in 3D Tiles 1.1. See [pnts migration guide](#).

### 2.7.1. Overview

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

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

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

## 2.7.2. Layout

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

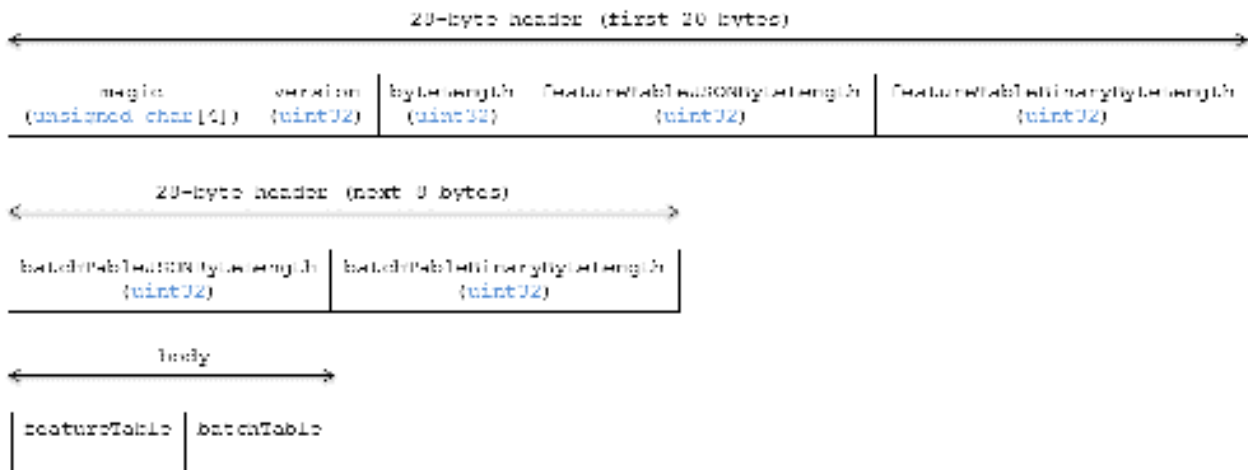


Figure 41. Data layout for a Point Cloud

### 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.

## 2.7.3. Header

The 28-byte header contains the following fields:

Field name	Data type	Description
<b>magic</b>	4-byte ANSI string	"pnts". This can be used to identify the content as a Point Cloud tile.
<b>version</b>	uint32	The version of the Point Cloud format. It is currently 1.
<b>byteLength</b>	uint32	The length of the entire tile, including the header, in bytes.
<b>featureTableJSONByteLength</b>	uint32	The length of the Feature Table JSON section in bytes.
<b>featureTableBinaryByteLength</b>	uint32	The length of the Feature Table binary section in bytes.
<b>batchTableJSONByteLength</b>	uint32	The length of the Batch Table JSON section in bytes. Zero indicates that there is no Batch Table.

Field name	Data type	Description
<code>batchTableBinaryByteLength</code>	<code>uint32</code>	The length of the Batch Table binary section in bytes. If <code>batchTableJSONByteLength</code> 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**.

## 2.7.4. Feature Table

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

The full JSON schema can be found in [pnts.featureTable.schema.json](#).

### Semantics

#### 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
<b>POSITION</b>	<code>float32[3]</code>	A 3-component array of numbers containing <i>x</i> , <i>y</i> , and <i>z</i> Cartesian coordinates for the position of the point.	Yes, unless <b>POSITION_QUANTIZED</b> is defined.
<b>POSITION_QUANTIZED</b>	<code>uint16[3]</code>	A 3-component array of numbers containing <i>x</i> , <i>y</i> , and <i>z</i> in quantized Cartesian coordinates for the position of the point.	Yes, unless <b>POSITION</b> is defined.
<b>RGBA</b>	<code>uint8[4]</code>	A 4-component array of values containing the <b>RGBA</b> color of the point.	No.
<b>RGB</b>	<code>uint8[3]</code>	A 3-component array of values containing the <b>RGB</b> color of the point.	No.
<b>RGB565</b>	<code>uint16</code>	A lossy compressed color format that packs the <b>RGB</b> color into 16 bits, providing 5 bits for red, 6 bits for green, and 5 bits for blue.	No.

Semantic	Data Type	Description	Required
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 <code>batchId</code> of the point that can be used to retrieve metadata from the <code>Batch Table</code> .	No.

### 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 <code>POSITION_QUANTIZED</code> is defined.
QUANTIZED_VOLUME_SCALE	float32[3]	A 3-component array of numbers defining the scale for the quantized volume.	No, unless <code>POSITION_QUANTIZED</code> is defined.
CONSTANT_RGBA	uint8[4]	A 4-component array of values defining a constant <code>RGBA</code> color for all points in the tile.	No.
BATCH_LENGTH	uint32	The number of unique <code>BATCH_ID</code> values.	No, unless <code>BATCH_ID</code> is defined.

Examples using these semantics can be found in the [examples section](#) below.

### Point positions

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

### 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](#)).

### 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.

### Quantized positions

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

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

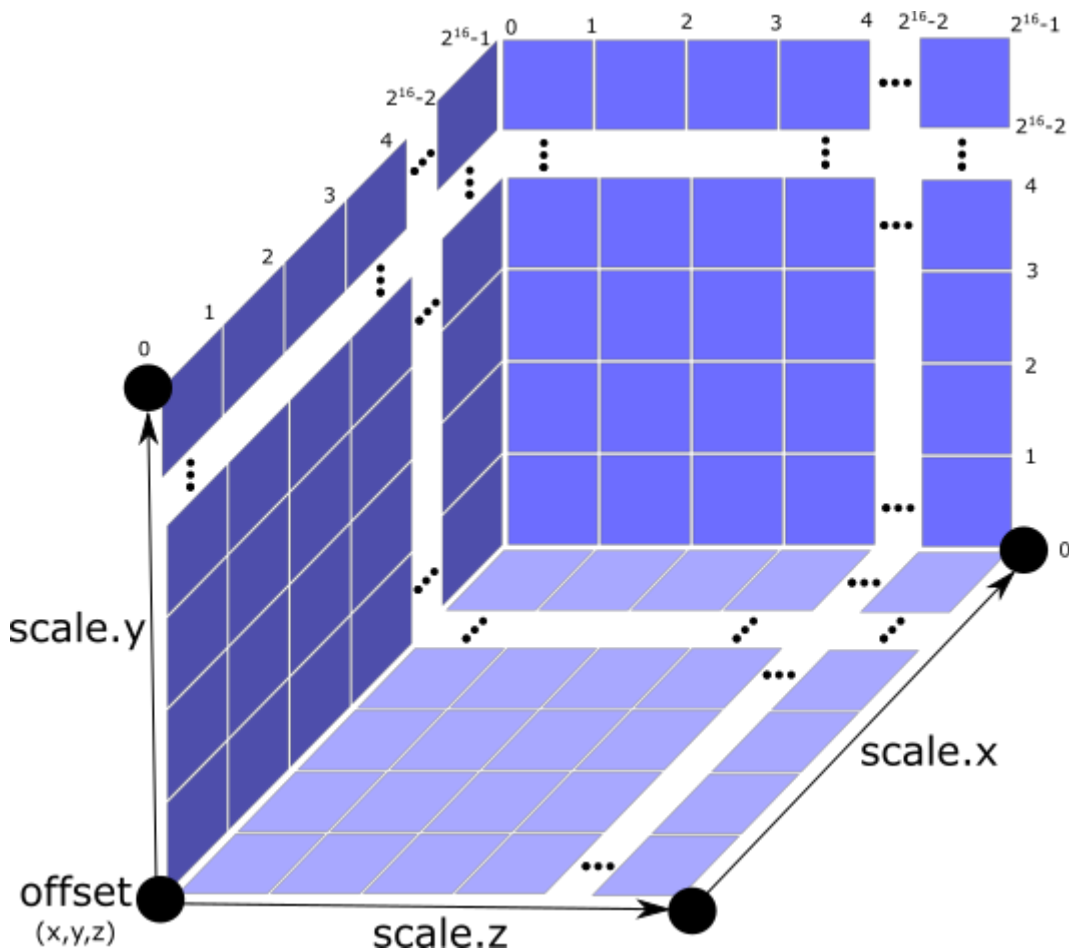


Figure 42. Illustration of the quantization that is used for the `POSITION_QUANTIZED` semantic

`offset` is stored in the global semantic `QUANTIZED_VOLUME_OFFSET`, and `scale` is stored in the global semantic `QUANTIZED_VOLUME_SCALE`. If those global semantics are not defined, `POSITION_QUANTIZED` cannot be used.

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

$POSITION = POSITION\_QUANTIZED * QUANTIZED\_VOLUME\_SCALE / 65535.0 + QUANTIZED\_VOLUME\_OFFSET$

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

### Point colors

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

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

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

### 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.

#### 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.

#### *Implementation Note*

#### NOTE

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

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

### 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.



## Examples

*This section is non-normative*

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

### 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);
```

### 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]);

```

### 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]);

```

### 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']
};

```

### 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']
};

```

### 2.7.5. Batch Table

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

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

See the [Batch Table](#) reference for more information.

### 2.7.6. Extensions

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

- `3DTILES_draco_point_compression`

### 2.7.7. 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.

### 2.7.8. Implementation example

*This section is non-normative*

Code for reading the header can be found in `PointCloud3DModelTileContent.js` in the CesiumJS

implementation of 3D Tiles.

## 2.8. Composite

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

### 2.8.1. Overview

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

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

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

A Composite tile is a binary blob in little endian.

### 2.8.2. Layout

Composite layout (dashes indicate optional fields):

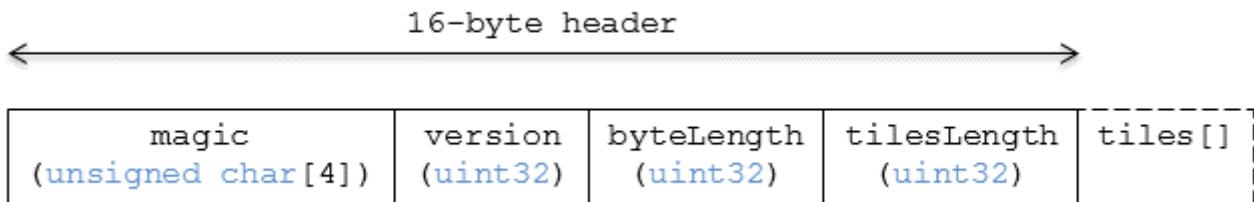


Figure 43. Data layout of a Composite tile

#### 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.

### 2.8.3. Header

The 16-byte header section contains the following fields:

Field name	Data type	Description
<code>magic</code>	4-byte ANSI string	<code>"cmt"</code> . This can be used to identify the content as a Composite tile.
<code>version</code>	<code>uint32</code>	The version of the Composite format. It is currently <code>1</code> .

Field name	Data type	Description
<code>byteLength</code>	<code>uint32</code>	The length of the entire Composite tile, including this header and each inner tile, in bytes.
<code>tilesLength</code>	<code>uint32</code>	The number of tiles in the Composite.

#### 2.8.4. Inner tiles

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

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

Field name	Data type	Description
<code>magic</code>	4-byte ANSI string	Indicates the tile format
<code>version</code>	<code>uint32</code>	1
<code>byteLength</code>	<code>uint32</code>	Length, in bytes, of the entire tile.

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

#### 2.8.5. 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.

#### 2.8.6. 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 CesiumJS implementation of 3D Tiles.

## 3. Implicit Tiling

## 3.1. Overview

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

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

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

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

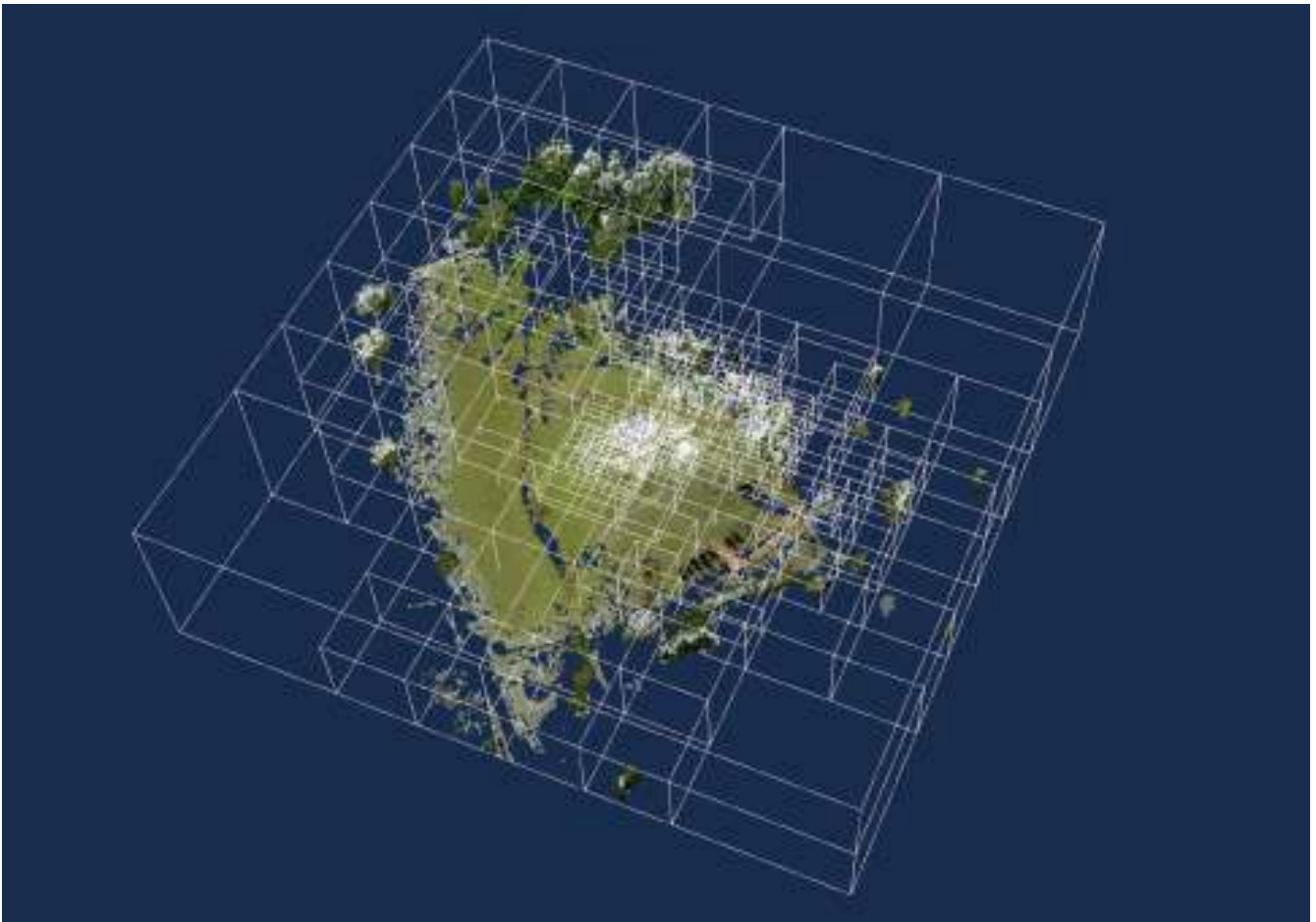


Figure 44. A point cloud organized into a sparse octree. Data source: Trimble

## 3.2. Implicit Root Tile

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



```

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

```

The `implicitTiling` object has the following properties:

Property	Description
<code>subdivisionScheme</code>	Either <code>QUADTREE</code> or <code>OCTREE</code> . See <a href="#">Subdivision scheme</a> .
<code>availableLevels</code>	How many levels there are in the tree with available tiles.
<code>subtreeLevels</code>	How many levels there are in each subtree.
<code>subtrees</code>	Template URI for subtree files. See <a href="#">Subtrees</a> .

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

The following constraints apply to implicit root tiles:

- The tile must omit the `children` property
- The tile must omit the `metadata` property
- The `content.uri` must not point to an [external tileset](#)
- The `content` must omit the `boundingVolume` property

### 3.3. Subdivision Scheme

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

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

unchanged. The resulting tree has 4 children per tile.

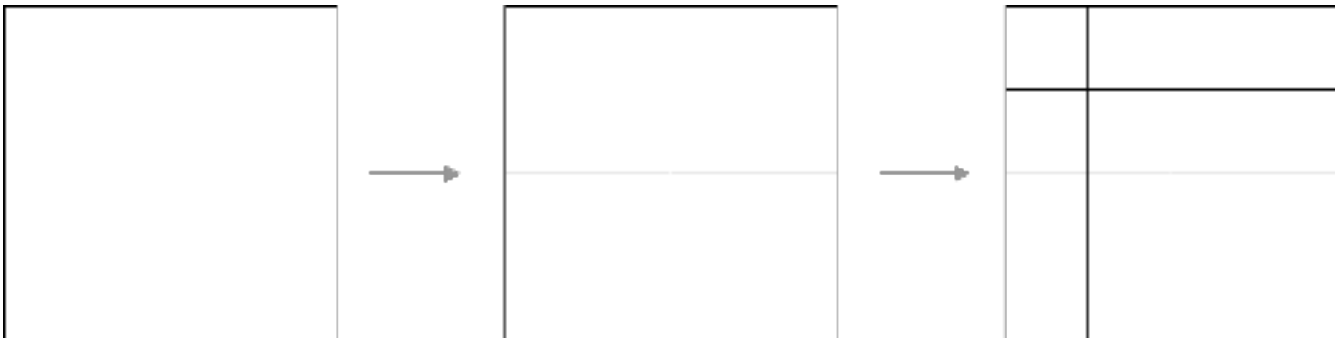


Figure 45. Subdivision in a quadtree

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

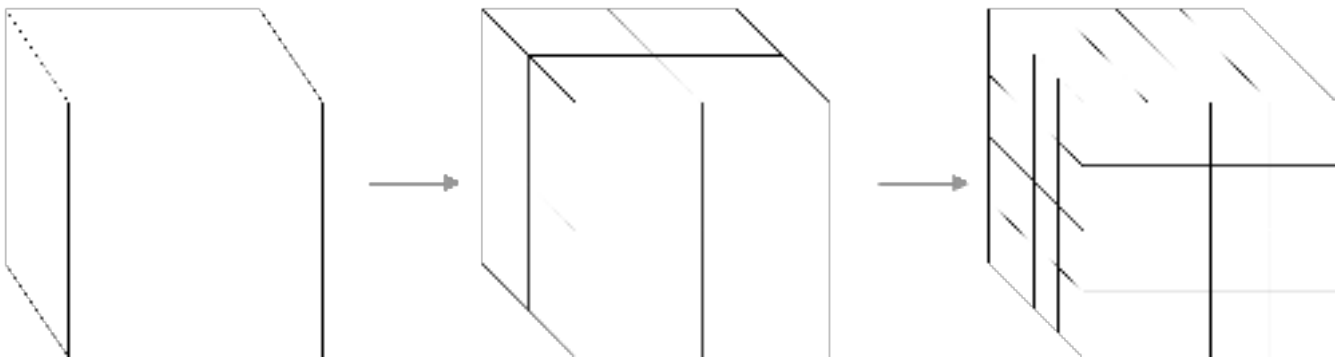


Figure 46. Subdivision in an octree

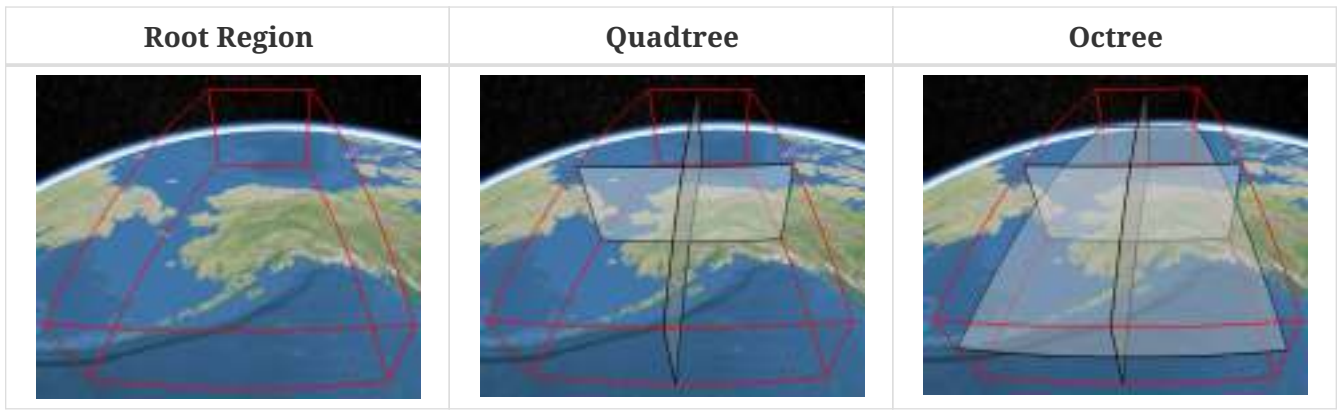
For a **region** bounding volume, **x**, **y**, and **z** refer to **longitude**, **latitude**, and **height** respectively.

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

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

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

Root Box	Quadtree	Octree



### 3.3.1. Subdivision Rules

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

Property	Subdivision Rule
<code>subdivisionScheme</code>	Constant for all descendant tiles
<code>refine</code>	Constant for all descendant tiles
<code>boundingVolume</code>	Divided into four or eight parts depending on the <code>subdivisionScheme</code>
<code>geometricError</code>	Each child's <code>geometricError</code> is half of its parent's <code>geometricError</code>

#### Implementation Note

In order to maintain numerical stability during this subdivision process, the actual bounding volumes should not be computed progressively by subdividing a non-root tile volume. Instead, the exact bounding volumes should be computed directly for a given level.

#### NOTE

Let the extent of the root bounding volume along one dimension  $d$  be  $(min_d, max_d)$ . The number of bounding volumes along that dimension for a given level is  $2^{level}$ . The size of each bounding volume at this level, along dimension  $d$ , is  $size_d = (max_d - min_d) / 2^{level}$ . The extent of the bounding volume of a child can then be computed directly as  $(min_d + size_d * i, min_d + size_d * (i + 1))$ , where  $i$  is the index of the child in dimension  $d$ .

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

## 3.4. Tile Coordinates

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

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

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

For **box** bounding volumes:

Coordinate	Positive Direction
x	Along the +x axis of the bounding box
y	Along the +y axis of the bounding box
z	Along the +z axis of the bounding box

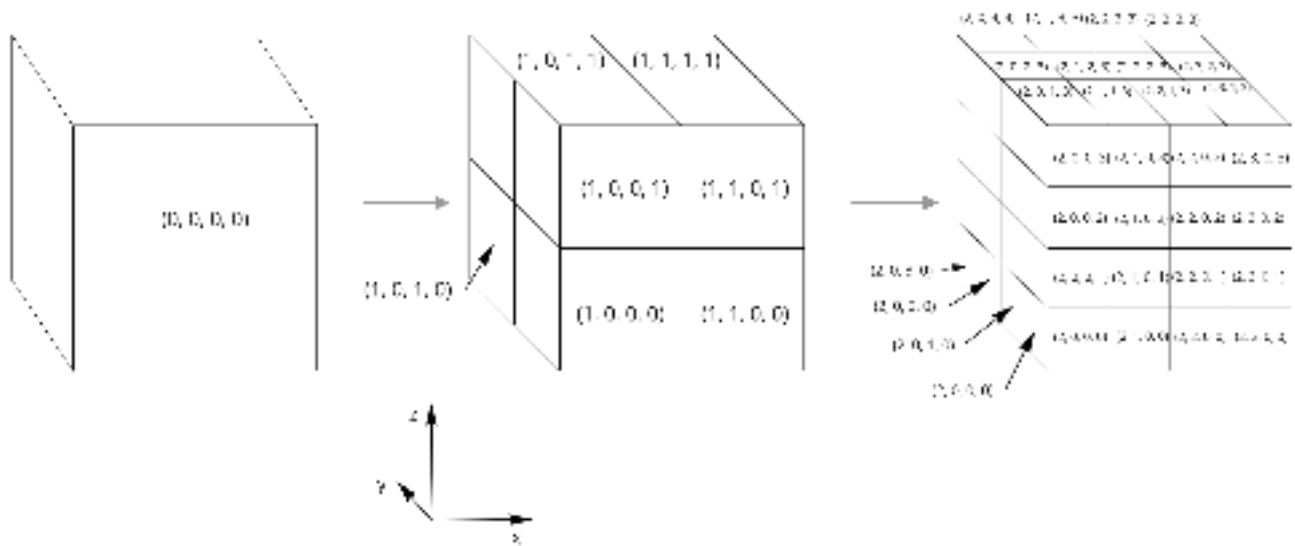


Figure 47. Coordinates of an octree node with a bounding box, and its child nodes

For **region** bounding volumes:

Coordinate	Positive Direction
x	From west to east (increasing longitude)
y	From south to north (increasing latitude)
z	From bottom to top (increasing height)

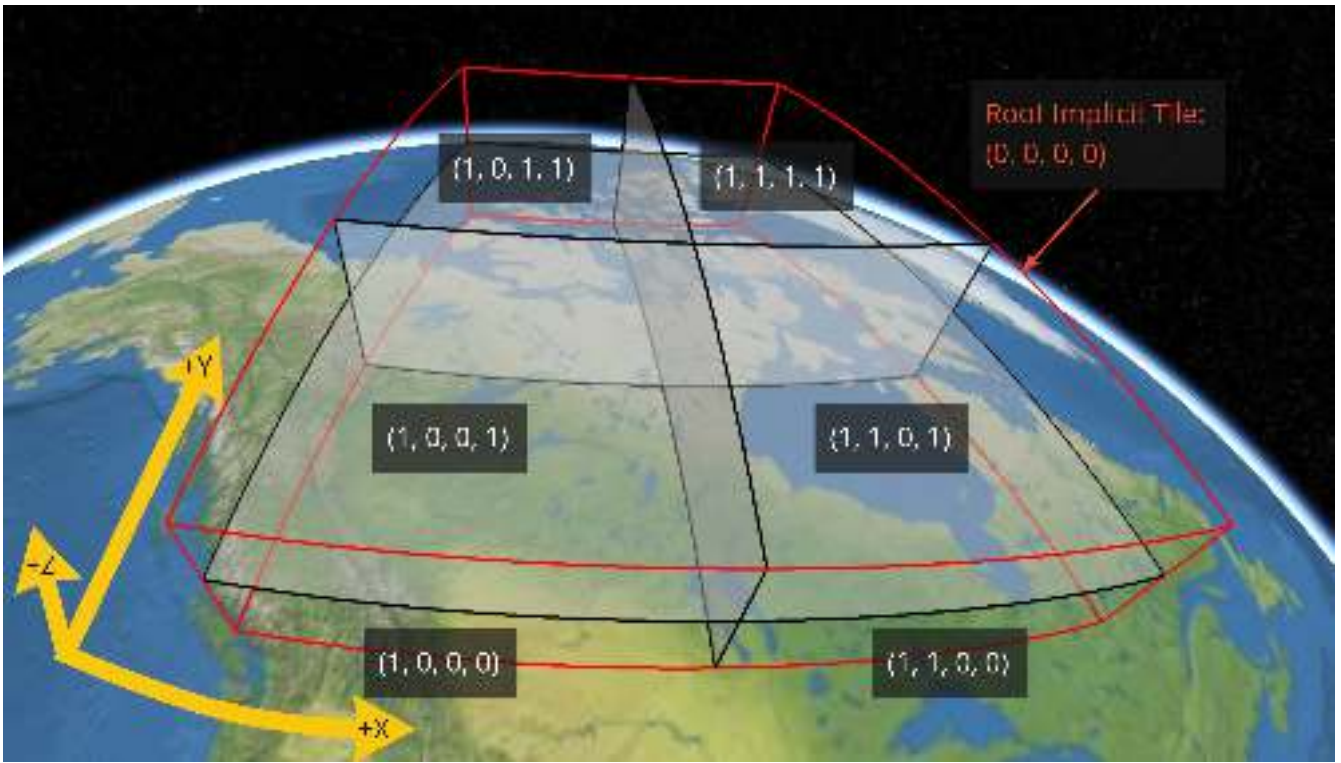


Figure 48. Coordinates of an octree node with a bounding region, and its child nodes

### 3.5. Template URIs

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

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

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

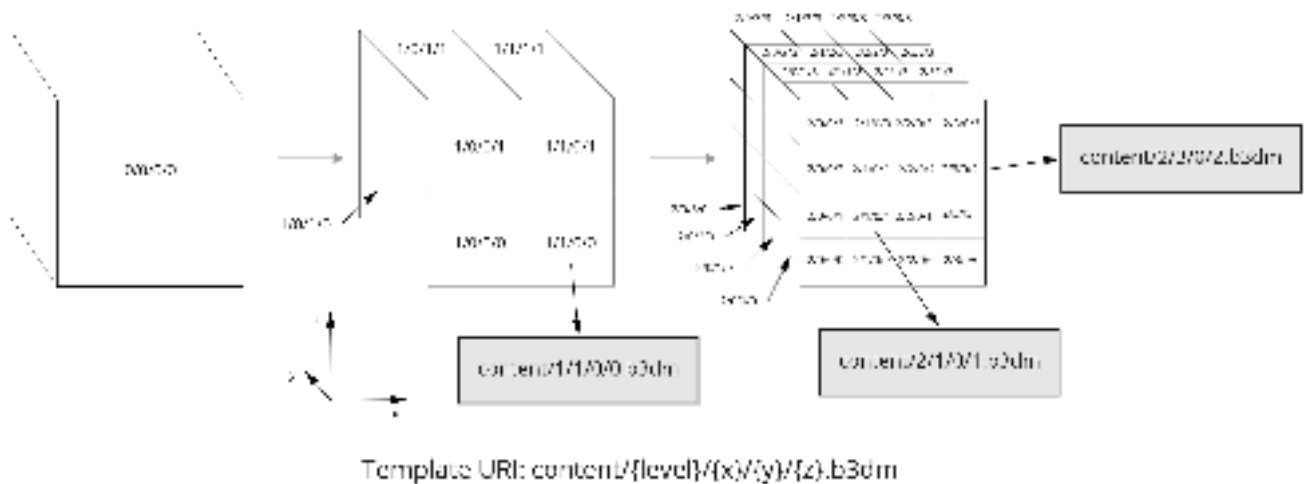


Figure 49. Examples of template URIs to identify the content for implicit tiles

### 3.6. Subtrees

In order to support sparse datasets, additional information is needed to indicate which tiles or

contents exist. This is called **availability**.

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

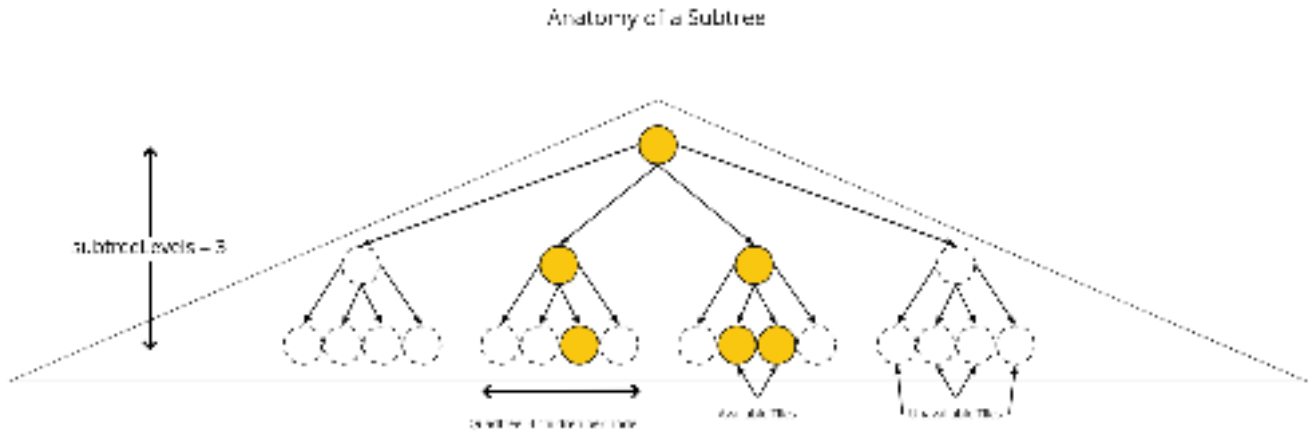


Figure 50. The structure of a subtree for implicit tiling

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

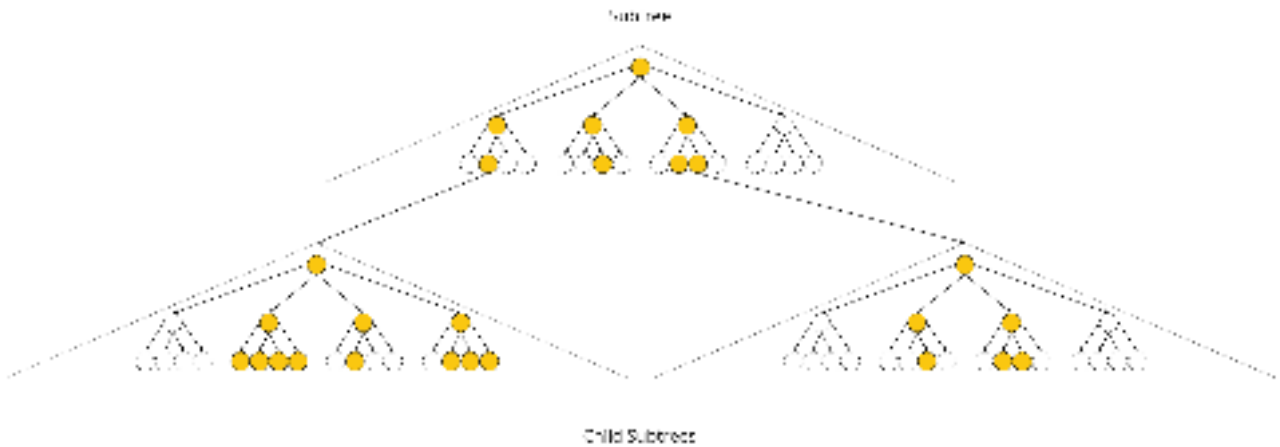


Figure 51. A tree of subtrees representing an implicit tileset

### 3.6.1. Availability

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

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

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





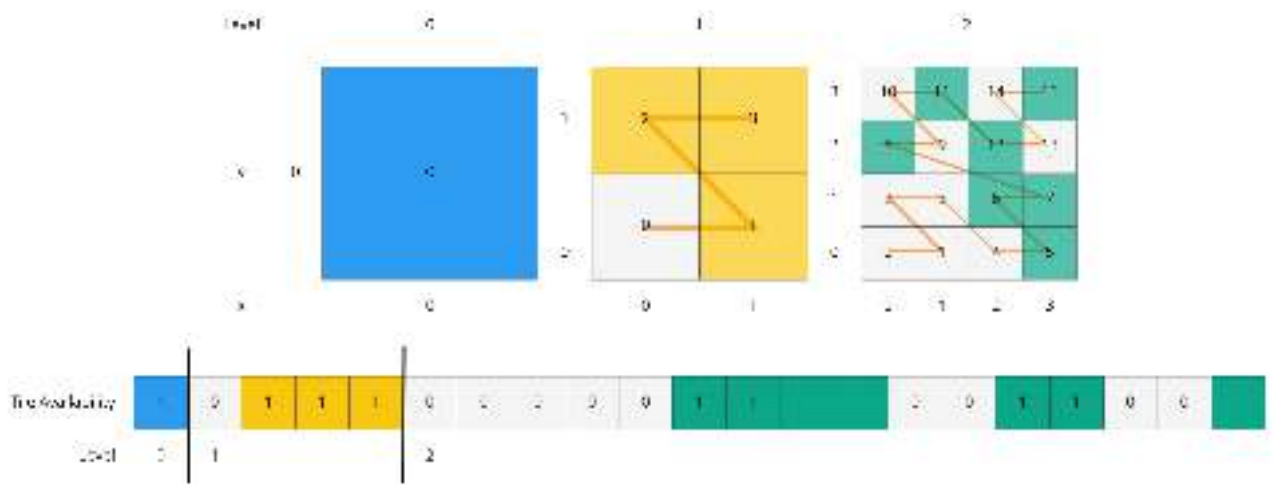


Figure 53. Illustration of a tile availability bitstream. Tiles that are available are represented with a 1 in the bitstream.

### Content Availability

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

Content availability has the following restrictions:

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

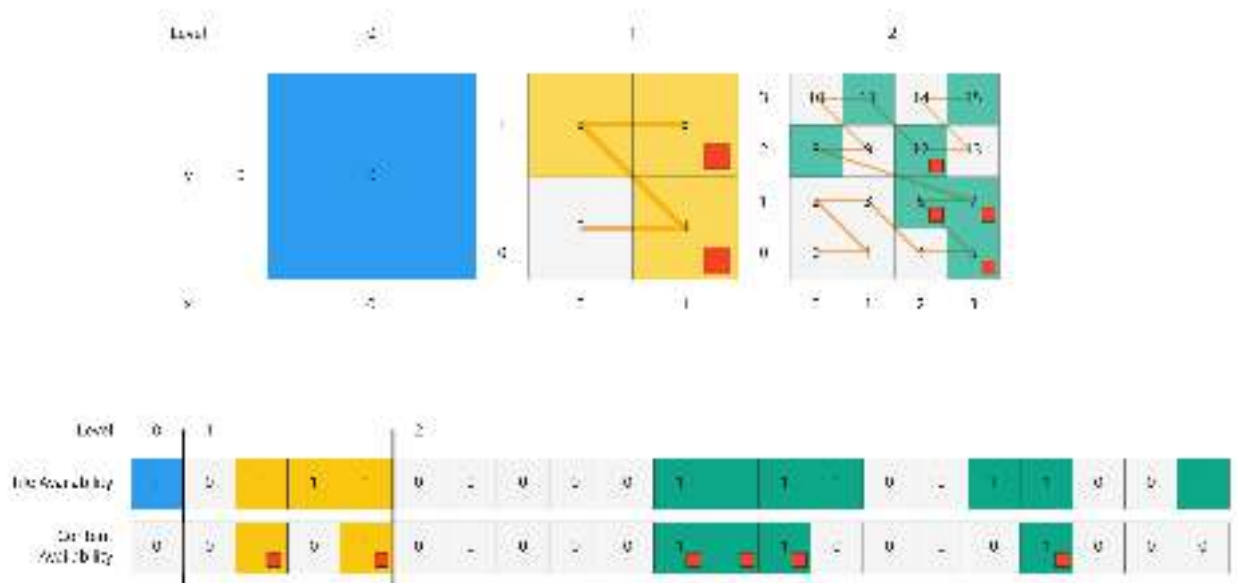


Figure 54. Illustration of a content availability bitstream. Tiles that have associated content are represented with a 1 in the bitstream.



## Child Subtree Availability

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

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

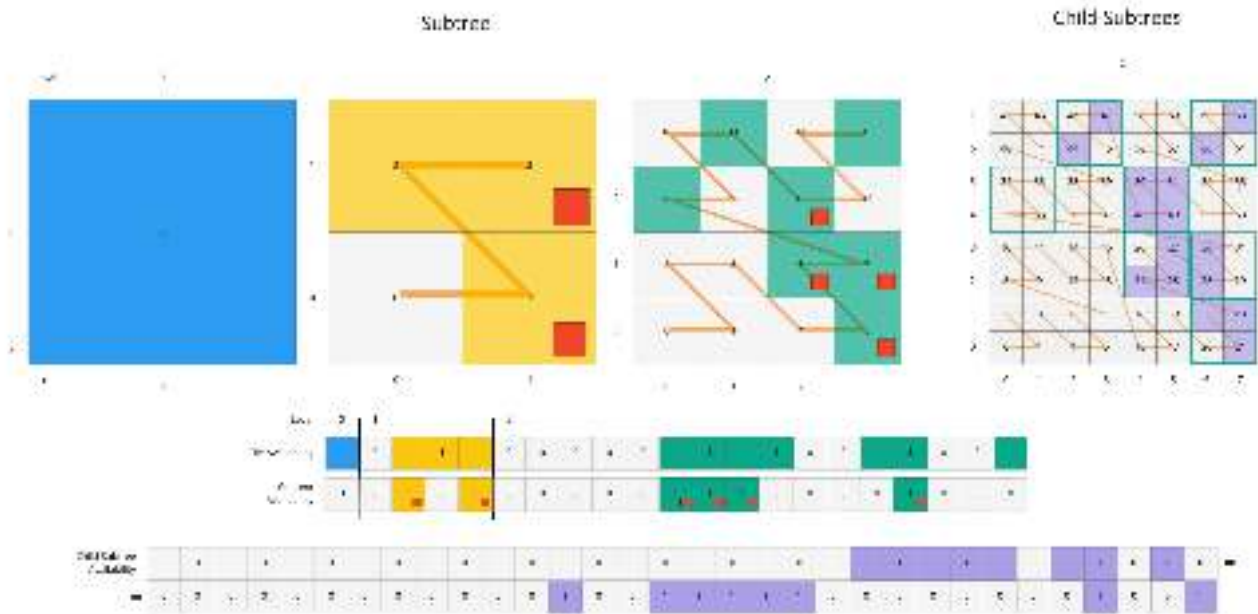


Figure 55. Illustration of a child subtree availability bitstream. Tiles that are the roots of available subtrees are represented by a 1 in the bitstream.

### 3.6.2. Metadata

Subtrees may store metadata at multiple granularities.

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

#### Tile Metadata

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

Tile metadata exists only for available tiles and is tightly packed by an increasing tile index

according to the [Availability Ordering](#). Each available tile must have a value—representation of missing values within a tile is possible only with the `noData` indicator defined by the [Binary Table Format](#) specification.

*Implementation Note*

**NOTE**

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

Tile properties can have [Semantics](#) which define how property values should be interpreted. In particular, `TILE_BOUNDING_BOX`, `TILE_BOUNDING_REGION`, `TILE_BOUNDING_SPHERE`, `TILE_MINIMUM_HEIGHT`, and `TILE_MAXIMUM_HEIGHT` semantics each define a more specific bounding volume for a tile than is implicitly calculated from implicit tiling. If more than one of these semantics are available for a tile, clients may select the most appropriate option based on use case and performance requirements.

### Example

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

NOTE

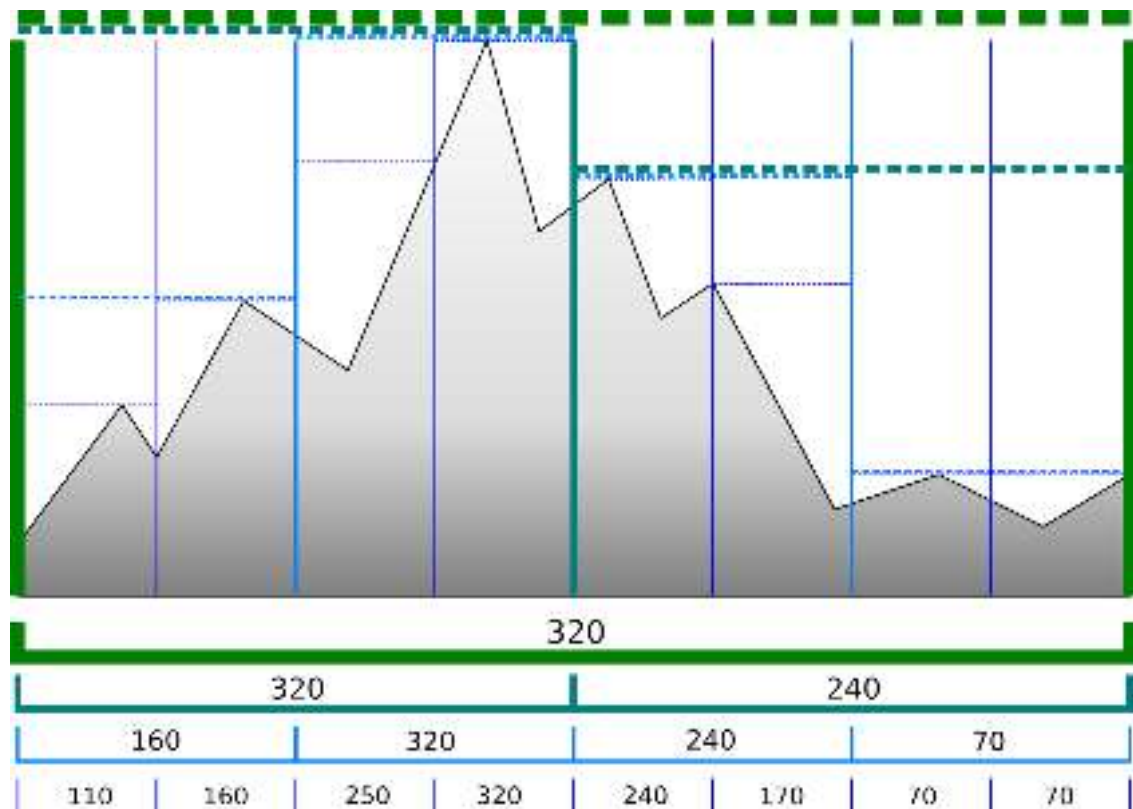


Figure 56. Illustration of storing the actual heights of individual tiles using the `TILE_MAXIMUM_HEIGHT` semantic

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

### Content Metadata

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

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

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

group.

## Subtree Metadata

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

# 3.7. Subtree JSON Format

Defined in [subtree.schema.json](#).

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

## 3.7.1. Buffers and Buffer Views

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

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

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

A **buffer view** is a contiguous subset of a buffer. A buffer view's `buffer` property is an integer index to identify the buffer. A buffer view has a `byteOffset` and a `byteLength` to describe the range of bytes within the buffer. The `byteLength` does not include any padding. There may be multiple buffer views referencing a single buffer.

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

## 3.7.2. Availability

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

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

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

Availability bitstreams are packed in binary using the format described in the [Booleans](#) section of the 3D Metadata Specification.

#### Example

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

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

#### NOTE

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

Only some tiles have content, and `contentAvailability.bufferView` indicates where the bitstream for the content availability is stored: The `bufferView` with index 0 refers to the `buffer` with index 0. This buffer does not have a `uri` property, and

therefore refers to the *internal* buffer that is stored directly in the binary chunk of the subtree binary file. The `byteOffset` and `byteLength` indicate that the content availability bitstream is stored in the bytes `[0...11)` of the internal buffer.

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

### 3.7.3. Metadata

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

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

#### *Example*

The same JSON description of a subtree extended with tile, content, and subtree metadata. The subtree JSON refers to a class ID in the root tileset schema. Tile and content metadata is stored in [property table](#); subtree metadata is encoded directly in JSON.

#### *Schema in the root tileset JSON*

NOTE

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





```

    { "buffer": 1, "byteOffset": 0, "byteLength": 2040 },
    { "buffer": 1, "byteOffset": 2040, "byteLength": 1530 },
    { "buffer": 1, "byteOffset": 3576, "byteLength": 344 },
    { "buffer": 1, "byteOffset": 3920, "byteLength": 1024 },
    { "buffer": 1, "byteOffset": 4944, "byteLength": 240 },
    { "buffer": 1, "byteOffset": 5184, "byteLength": 122 },
    { "buffer": 1, "byteOffset": 5312, "byteLength": 480 },
    { "buffer": 1, "byteOffset": 5792, "byteLength": 480 },
    { "buffer": 1, "byteOffset": 6272, "byteLength": 240 }
  ],
  "propertyTables": [
    {
      "class": "tile",
      "count": 85,
      "properties": {
        "horizonOcclusionPoint": {
          "values": 2
        },
        "countries": {
          "values": 3,
          "arrayOffsets": 4,
          "stringOffsets": 5,
          "arrayOffsetType": "UINT32",
          "stringOffsetType": "UINT32"
        }
      }
    },
    {
      "class": "content",
      "count": 60,
      "properties": {
        "attributionIds": {
          "values": 6,
          "arrayOffsets": 7,
          "arrayOffsetType": "UINT16"
        },
        "minimumHeight": {
          "values": 8
        },
        "maximumHeight": {
          "values": 9
        },
        "triangleCount": {
          "values": 10,
          "min": 520,
          "max": 31902
        }
      }
    }
  ],
  "tileAvailability": {

```

```

    "constant": 1
  },
  "contentAvailability": [{
    "bitstream": 0,
    "availableCount": 60
  }],
  "childSubtreeAvailability": {
    "bitstream": 1
  },
  "tileMetadata": 0,
  "contentMetadata": [1],
  "subtreeMetadata": {
    "class": "subtree",
    "properties": {
      "attributionStrings": [
        "Source A",
        "Source B",
        "Source C",
        "Source D"
      ]
    }
  }
}
}
}

```

### 3.7.4. Multiple Contents

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

#### *Example*

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

#### *Implicit root tile*

**NOTE**

```

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

```

*Subtree JSON*

```

{
  "propertyTables": [
    {
      "class": "building",
      "count": 85,
      "properties": {
        "height": {
          "values": 1
        },
        "owners": {
          "values": 2,
          "arrayOffsets": 3,
          "stringOffsets": 4
        }
      }
    },
    {
      "class": "tree",
      "count": 52,
      "properties": {
        "height": {
          "values": 5
        },
        "species": {
          "values": 6
        }
      }
    }
  ],
  "tileAvailability": {
    "constant": 1
  },
  "contentAvailability": [
    {
      "constant": 1
    },
    {
      "bitstream": 0,
      "availableCount": 52
    }
  ],
  "childSubtreeAvailability": {
    "constant": 1
  },
  "contentMetadata": [0, 1]
}

```

## 3.8. Subtree Binary Format

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

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



Figure 57. Data layout for the subtree binary format

Header fields:

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

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

- The JSON chunk must be padded with trailing `Space` chars (`0x20`)
- If it exists, the binary chunk must be padded with trailing zeros (`0x00`)

## 4. 3D Metadata Specification

### 4.1. Overview

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

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

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

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

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

## 4.2. Concepts

This specification defines metadata schemas and methods for encoding metadata.

**Schemas** contain a set of **classes** and **enums**. A class represents a category of similar entities, and is defined as a set of **properties**. Each property describes values of a particular type. An enum defines a set of named values representing a single value type, and may be referenced by class properties. Schema definitions do not describe how entities or properties are stored, and may be represented in a file format in various ways. Schemas can be reused across multiple assets or even file formats.

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

### *Implementation Note*

#### **NOTE**

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

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

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

### 4.2.1. Identifiers

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

## 4.3. Schemas

### 4.3.1. Schema

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

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

#### ID

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

#### Version

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

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

#### NOTE

*Example*

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

#### Name

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

#### Description

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



## Enums

Unordered set of [enums](#).

## Classes

Unordered set of [classes](#).

---

### 4.3.2. Enum

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

#### *Example*

A "species" enum with three possible tree species, as well as an "Unknown" value.

- **ID:** "species"
- **Name:** "Species"
- **Description:** "Common tree species identified in the study."
- **Value type:** [INT32](#)

#### NOTE

<b>name</b>	<b>value</b>
"Oak"	0
"Pine"	1
"Maple"	2
"Unknown"	-1

## ID

IDs ([id](#)) are unique [identifiers](#) for an enum within a schema.

## Name

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

## Description

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

## Values

An enum consists of a set of named values, represented as ([string](#), [integer](#)) pairs. The following enum value types are supported: [INT8](#), [UINT8](#), [INT16](#), [UINT16](#), [INT32](#), [UINT32](#), [INT64](#), and [UINT64](#). See the

[Component Type](#) section for definitions of each. Smaller enum types limit the range of possible enum values, and allow more efficient binary encoding. Duplicate names or values within the same enum are not allowed.

---

### 4.3.3. Class

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

#### **ID**

IDs ([id](#)) are unique [identifiers](#) for a class within a schema.

#### **Name**

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

#### **Description**

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

#### **Properties**

Unordered set of [properties](#).

---

### 4.3.4. Property

#### **Overview**

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

### Example

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

### building

property	type	componentType
height	SCALAR	FLOAT32

NOTE

### tree

property	type	componentType
enumType	height	SCALAR
FLOAT32		species
ENUM		species
leafColor	STRING	

## ID

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

## Name

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

### Example

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

NOTE

## Description

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

## Semantic

Property IDs, names, and descriptions do not have an inherent meaning. To provide a machine-readable meaning, properties may be assigned a semantic identifier string (**semantic**), indicating

how the property’s content should be interpreted. Semantic identifiers may be defined by the [3D Metadata Semantic Reference](#) or by external semantic references, and may be application-specific. Identifiers should be uppercase, with underscores as word separators.

*Example*

**NOTE**

Semantic definitions might include temperature in degrees Celsius (e.g. `TEMPERATURE_DEGREES_CELSIUS`), time in milliseconds (e.g. `TIME_MILLISECONDS`), or mean squared error (e.g. `MEAN_SQUARED_ERROR`). These examples are only illustrative.

**Type**

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

<b>name</b>	<b>type</b>
SCALAR	Single numeric component
VEC2	Fixed-length vector with two (2) numeric components
VEC3	Fixed-length vector with three (3) numeric components
VEC4	Fixed-length vector with four (4) numeric components
MAT2	2x2 matrix with numeric components
MAT3	3x3 matrix with numeric components
MAT4	4x4 matrix with numeric components
STRING	A sequence of characters
BOOLEAN	True or false
ENUM	An enumerated type

**Component Type**

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

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

name	componentType
FLOAT64	A number that can be represented as a 64-bit IEEE floating point number

Floating-point properties (`Float32` and `Float64`) must not include values `NaN`, `+Infinity`, or `-Infinity`.

*Implementation Note*

**NOTE**

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

### Enum Type

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

### Arrays

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

### Normalized Values

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

component Type	int to float	float to int
INT8	$f = \max(i / 127.0, -1.0)$	$i = \text{round}(f * 127.0)$
UINT8	$f = i / 255.0$	$i = \text{round}(f * 255.0)$
INT16	$f = \max(i / 32767.0, -1.0)$	$i = \text{round}(f * 32767.0)$
UINT16	$f = i / 65535.0$	$i = \text{round}(f * 65535.0)$
INT32	$f = \max(i / 2147483647.0, -1.0)$	$i = \text{round}(f * 2147483647.0)$
UINT32	$f = i / 4294967295.0$	$i = \text{round}(f * 4294967295.0)$
INT64	$f = \max(i / 9223372036854775807.0, -1.0)$	$i = \text{round}(f * 9223372036854775807.0)$
UINT64	$f = i / 18446744073709551615.0$	$i = \text{round}(f * 18446744073709551615.0)$

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

*Implementation Note*

**NOTE**

Depending on the implementation and the chosen integer type, there may be some loss of precision in values after denormalization. For example, if the implementation uses 32-bit floating point variables to represent the value of a normalized 32-bit integer, there are only 23 bits in the mantissa of the float, and lower bits will be truncated by denormalization. Client implementations should use higher precision floats when appropriate for correctly representing the result.

## Offset and Scale

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

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

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

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

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

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

*Implementation Note*

**NOTE**

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

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

## Minimum and Maximum Values

Properties may specify a minimum (`min`) and maximum (`max`) value. Minimum and maximum

values represent component-wise bounds of the valid range of values for a property. Both values are *inclusive*, meaning that they denote the smallest and largest allowed value, respectively.

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

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

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

*Example*

**NOTE**

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

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

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

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

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

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

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

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

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



### Example

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

#### NOTE

property	componentType	required	noData
height	FLOAT32	Yes	
species	ENUM		"Unknown"
leafColor	STRING	Yes	

## Property Values Structure

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

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

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

## 4.4. Storage Formats

### 4.4.1. Overview

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

- **Binary Table Format** - property values are stored in parallel 1D arrays, encoded as binary data
- **JSON Format** - property values are stored in key/value dictionaries, encoded as JSON objects

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

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

### Implementation Note


#### NOTE

Any specification that references 3D Metadata must state explicitly which storage formats are supported, or define its own serialization. For example, the `EXT_structural_metadata` glTF extension implements the binary table format described below, and defines an additional image-based format for per-textel metadata.

## 4.4.2. Binary Table Format

### Overview

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



Houses

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

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

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

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

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

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

### Scalars

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

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

## Vectors

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

## Matrices

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

## Booleans

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

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

### *Implementation Note*

Example accessing a boolean value for entity ID `i`.

#### NOTE

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

## Strings

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

Because string lengths may vary, a **string offset** buffer is used to identify strings in the buffer. If

there are  $N$  strings in the property array, the string offset buffer has  $N + 1$  elements. The first  $N$  of these point to the first byte of each string, while the last points to the byte immediately after the last string. The number of bytes in the  $i$ -th string is given by `stringOffset[i + 1] - stringOffset[i]`. UTF-8 encodes each character as 1-4 bytes, so string offsets do not necessarily represent the number of characters in the string.

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

#### Example

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

#### NOTE

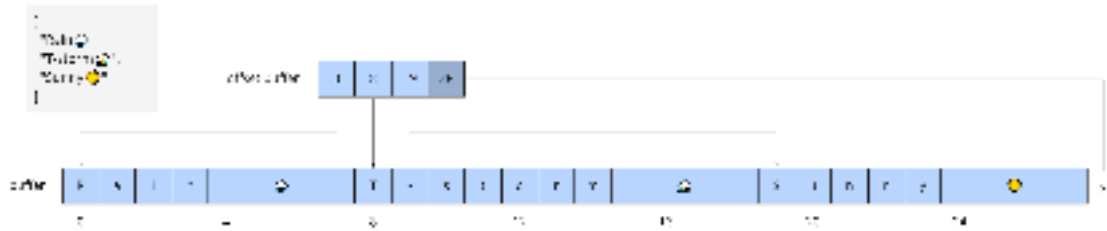


Figure 59. Data layout for the buffers storing string metadata

## Enums

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

## Fixed-Length Arrays

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

## Variable-Length Arrays

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

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

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

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



### 4.4.3. JSON Format

#### Overview

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

Each entity is represented as a JSON object with its `class` identified by a string ID. Property values are defined in a key/value `properties` dictionary, having property IDs as its keys. Property values are encoded as corresponding JSON types: numeric types are represented as `number`, booleans as `boolean`, strings as `string`, enums as `string`, vectors and matrices as `array` of `number`, and arrays as `array` of the containing type.

#### Example

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

An enum, "basicEnum", composed of three (name: value) pairs:

name	value
"Enum A"	0
"Enum B"	1
"Enum C"	2

A class, "basicClass", composed of ten properties. `stringArrayProperty` count is undefined and therefore variable-length.

#### NOTE

id	type	componentType	array	count	enumType	required
floatProperty	SCALAR	FLOAT64	false			Yes
integerProperty	SCALAR	INT32	false			Yes
vectorProperty	VEC2	FLOAT32	false			Yes
floatArrayProperty	SCALAR	FLOAT32	true	3		Yes
vectorArrayProperty	VEC2	FLOAT32	true	2		Yes
booleanProperty	BOOLEAN		false			Yes
stringProperty	STRING		false			Yes
enumProperty	ENUM		false		basicEnum	Yes
stringArrayProperty	STRING		true			Yes
optionalProperty	STRING		false			

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

```

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

```

## Scalars

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

### *Implementation Note*

#### NOTE

For numeric types the size in bits is made explicit. Even though JSON only has a single `number` type for all integers and floating point numbers, the application that consumes the JSON may make a distinction. For example, C and C++ have several different integer types such as `uint8_t`, `uint32_t`. The application is responsible for interpreting the metadata using the type specified in the property definition.

## Vectors

Vectors are encoded as a JSON array of numbers.

## Matrices

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

## Booleans

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

## Strings

Strings are encoded as JSON strings.



## Enums

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

## Arrays

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

# 5. Styling

## 5.1. Overview

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

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

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

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

The following example assigns a color based on building height.

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

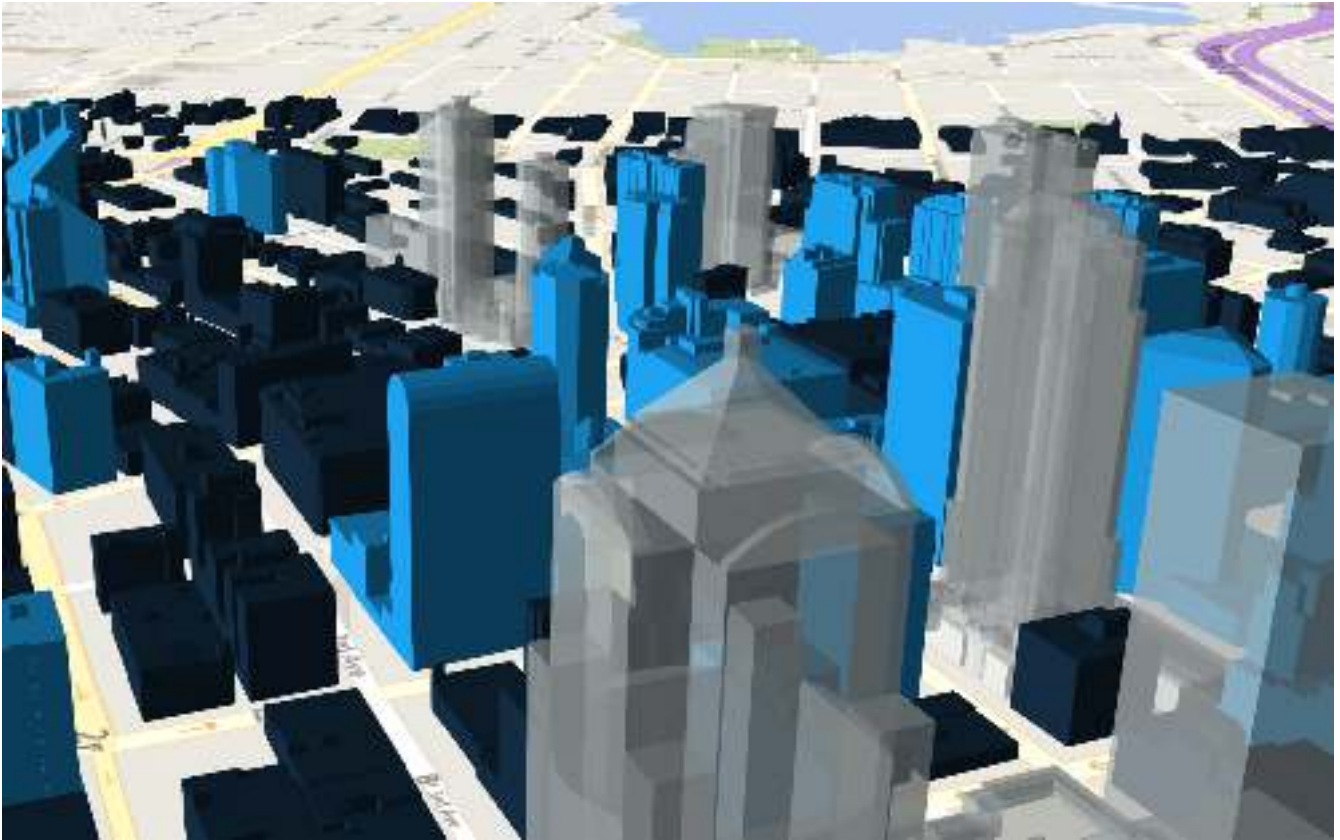


Figure 62. Rendering of buildings with different colors, based on the height of the buildings

## 5.2. Concepts

### 5.2.1. 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))"
}
```

### 5.2.2. 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')"]
  ]
}

```

### 5.2.3. 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)"]
    ]
  }
}

```

### 5.2.4. 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}"
  }
}

```

## 5.3. Expressions

The language for expressions is a small subset of JavaScript ([EMCAScript 5](#)), plus native vector and regular expression types and access to tileset feature properties in the form of readonly variables.

#### *Implementation Note*

**NOTE** CesiumJS uses the [jsep](#) JavaScript expression parser library to parse style expressions into an [abstract syntax tree \(AST\)](#).

### 5.3.1. Semantics

Dot notation is used to access properties by name, e.g., `building.name`.

Bracket notation (`[]`) is also used to access properties, e.g., `building['name']`, or arrays, e.g., `temperatures[1]`.

Functions are called with parenthesis (`()`) and comma-separated arguments, e.g., `(isNaN(0.0), color('cyan', 0.5))`.

### 5.3.2. 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.

### 5.3.3. 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')`

## Number

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

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

## String

Strings are encoded in UTF-8.

## 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:

### `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`

### `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

#### `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

#### 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.

## 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)`

## 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')
```

### 5.3.4. Operator rules

- Unary operators `+` and `-` operate only on number and vector expressions.
- Unary operator `!` operates only on boolean expressions.
- Binary operators `<`, `<=`, `>`, and `>=` operate only on number expressions.
- Binary operators `||` and `&&` operate only on boolean expressions.
- Binary operator `+` operates on the following expressions:
  - Number expressions
  - Vector expressions of the same type
  - If at least one expressions 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.

### 5.3.5. 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.

### 5.3.6. 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/"

### 5.3.7. Constants

The following constants are supported by the styling language:

- `Math.PI`
- `Math.E`

#### PI

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

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

#### E

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

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

### 5.3.8. Variables

Variables are used to retrieve the property values of individual features in a tileset. Variables are identified using the ES 6 (ECMAScript 2015) template literal syntax, i.e., `{feature.identifier}` or `{feature['identifier']}`, where the identifier is the case-sensitive property name. Variable names are encoded in UTF-8. `feature` is implicit and can be omitted in most cases. If the identifier contains non-alphanumeric characters, such as `:`, `-`, `#`, or spaces, the `{feature['identifier']}` form should be used.

Variables can be used anywhere a valid expression is accepted, except inside other variable identifiers. For example, the following is not allowed:

```
{foo[{bar}]}
```

If a feature does not have a property with the specified name, the variable evaluates to `undefined`. Note that the property may also be `null` if `null` was explicitly stored for a property.

Variables may be any of the supported native JavaScript types:

- Boolean
- Null
- Undefined
- Number
- String
- Array

For example:

```
{
  "enabled" : true,
  "description" : null,
  "order" : 1,
  "name" : "Feature name"
}
```

```
${enabled} === true
${description} === null
${order} === 1
${name} === 'Feature name'
```

Additionally, variables originating from vector properties stored in the [Batch Table binary](#) are treated as vector types:

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]
```

### 5.3.9. Built-in functions

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

- `abs`
- `sqrt`
- `cos`
- `sin`
- `tan`
- `acos`
- `asin`
- `atan`
- `atan2`
- `radians`
- `degrees`
- `sign`
- `floor`
- `ceil`

- `round`
- `exp`
- `log`
- `exp2`
- `log2`
- `fract`
- `pow`
- `min`
- `max`
- `clamp`
- `mix`
- `length`
- `distance`
- `normalize`
- `dot`
- `cross`

Many of the built-in functions take either scalars or vectors as arguments. For vector arguments the function is applied component-wise and the resulting vector is returned.

### **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"
}
```

### **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)"]
    ]
  }
}
```

## 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"
}
```

## 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"
}
```

## 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"  
}
```

## 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"  
}
```

## 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"  
}
```

## 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"  
}
```

## 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"  
}
```

## 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"  
}
```

## 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"  
}
```

## 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"
}
```

## 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"
}
```

## 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"
}
```

## 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"
}
```

## 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"
}
```

## 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"
}
```

## 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"
}
```

## 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"
}
```

## 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})"
}
```

## 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"
}
```

## 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"
}
```

## 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"  
}
```

## 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)"  
}
```

## 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"  
}
```

## 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"
}
```

## 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., `length(x - y)`.

```
{
  "show" : "distance(${BottomRight}, ${UpperLeft}) > 50.0"
}
```

## 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"
}
```

## 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"
}
```

### 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)"
}
```

### 5.3.10. Notes

Comments are not supported.

## 5.4. 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:

#### **NOTE**

- 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

## 5.5. File extension and MIME type

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

# Appendix A: Migration From Legacy Tile Formats

This section describes how legacy tile formats can be converted into equivalent glTF content.

## Batched 3D Model (b3dm)

[Batched 3D Model](#) is a wrapper around a binary glTF that includes additional information in its Feature Table and Batch Table. Batched 3D Model content can be converted into glTF content with the following changes:

- The `RTC_CENTER` can be added to the translation component of the root node of the glTF asset.
- Batch IDs and Batch Tables can be represented using `EXT_mesh_features` and `EXT_structural_metadata`.



Figure 63. Batched 3D Models in 3D Tiles 1.0, and the corresponding representation in 3D Tiles 1.1

## Instanced 3D Model (i3dm)

[Instanced 3D Model](#) instances a glTF asset (embedded or external) and provides per-instance transforms and batch IDs.

- The `RTC_CENTER` can be added to the translation component of the root node of the glTF asset.
- glTF can leverage GPU instancing with the `EXT_mesh_gpu_instancing` extension.
- Batch IDs and Batch Tables can be represented using `EXT_instance_features` and `EXT_structural_metadata`.
- `EAST_NORTH_UP` is not directly supported, but can be represented using per-instance rotations.



Figure 64. Instanced 3D Models in 3D Tiles 1.0, and the corresponding representation in 3D Tiles 1.1

## Point Cloud (pnts)

[Point Cloud](#) can be represented as a glTF using the primitive mode `0` (`POINTS`).

- The `RTC_CENTER` can be added to the translation component of the root node of the glTF asset.

- Feature table properties like **POSITION**, **COLOR**, and **NORMAL** may be stored as glTF attributes.
- **EXT\_meshopt\_compression** and **KHR\_mesh\_quantization** may be used for point cloud compression. **3DTILES\_draco\_point\_compression** is not directly supported in glTF because **KHR\_draco\_mesh\_compression** only supports triangle meshes.
- Batch IDs and Batch Tables can be represented using **EXT\_mesh\_features** and **EXT\_structural\_metadata**.
- **CONSTANT\_RGBA** is not directly supported in glTF, but can be achieved with materials or per-point colors.



Figure 65. Point Clouds in 3D Tiles 1.0, and the corresponding representation in 3D Tiles 1.1

## Composite (cmpt)

All inner contents of a [Composite](#) may be combined into the same glTF as separate nodes, meshes, or primitives, at the tileset author's discretion. Alternatively, a tile may have [multiple contents](#).

# Appendix B: Availability Indexing

## Converting from Tile Coordinates to Morton Index

A [Morton index](#) is computed by interleaving the bits of the  $(x, y)$  or  $(x, y, z)$  coordinates of a tile. Specifically:

```
quadtreeMortonIndex = interleaveBits(x, y)
octreeMortonIndex = interleaveBits(x, y, z)
```

For example:

```
// Quadtree
interleaveBits(0b11, 0b00) = 0b0101
interleaveBits(0b1010, 0b0011) = 0b01001110
interleaveBits(0b0110, 0b0101) = 0b00110110

// Octree
interleaveBits(0b001, 0b010, 0b100) = 0b100010001
interleaveBits(0b111, 0b000, 0b111) = 0b101101101
```



Figure 66. An example for the computation of the Morton index based on the tile coordinates on three levels of a quadtree

## Availability Bitstream Lengths

Availability Type	Length (bits)	Description
Tile availability	$(N^{\text{subtreeLevels}} - 1) / (N - 1)$	Total number of nodes in the subtree
Content availability	$(N^{\text{subtreeLevels}} - 1) / (N - 1)$	Since there is at most one content per tile, this is the same length as tile availability
Child subtree availability	$N^{\text{subtreeLevels}}$	Number of nodes one level deeper than the deepest level of the subtree

Where  $N$  is 4 for quadtrees and 8 for octrees.

These lengths are in number of bits in a bitstream. To compute the length of the bitstream in bytes, the following formula is used:

$$\text{lengthBytes} = \text{ceil}(\text{lengthBits} / 8)$$

## Accessing Availability Bits

For tile availability and content availability, the Morton index only determines the ordering within a single level of the subtree. Since the availability bitstream stores bits for every level of the subtree, a level offset must be computed.



Given the (`level`, `mortonIndex`) of a tile relative to the subtree root, the index of the corresponding bit can be computed with the following formulas:

Quantity	Formula	Description
<code>levelOffset</code>	$(N^{\text{level}} - 1) / (N - 1)$	This is the number of nodes at levels 1, 2, ... ( <code>level - 1</code> )
<code>tileAvailabilityIndex</code>	<code>levelOffset + mortonIndex</code>	The index into the buffer view is the offset for the tile's level plus the morton index for the tile

Where `N` is 4 for quadtrees and 8 for octrees.

Since child subtree availability stores bits for a single level, no `levelOffset` is needed, i.e. `childSubtreeAvailabilityIndex = mortonIndex`, where the `mortonIndex` is the Morton index of the desired child subtree relative to the root of the current subtree.

## Global and Local Tile Coordinates

When working with tile coordinates, it is important to consider which tile the coordinates are relative to. There are two main types used in implicit tiling:

- **global coordinates** - coordinates relative to the implicit root tile.
- **local coordinates** - coordinates relative to the root of a specific subtree.

Global coordinates are used for locating any tile in the entire implicit tileset. For example, template URIs use global coordinates to locate content files and subtrees. Meanwhile, local coordinates are used for locating data within a single subtree file.

In binary, a tile's global Morton index is the complete path from the implicit root tile to the tile. This is the concatenation of the path from the implicit root tile to the subtree root tile, followed by the path from the subtree root tile to the tile. This can be expressed with the following equation:

```
tile.globalMortonIndex = concatBits(subtreeRoot.globalMortonIndex,  
tile.localMortonIndex)
```

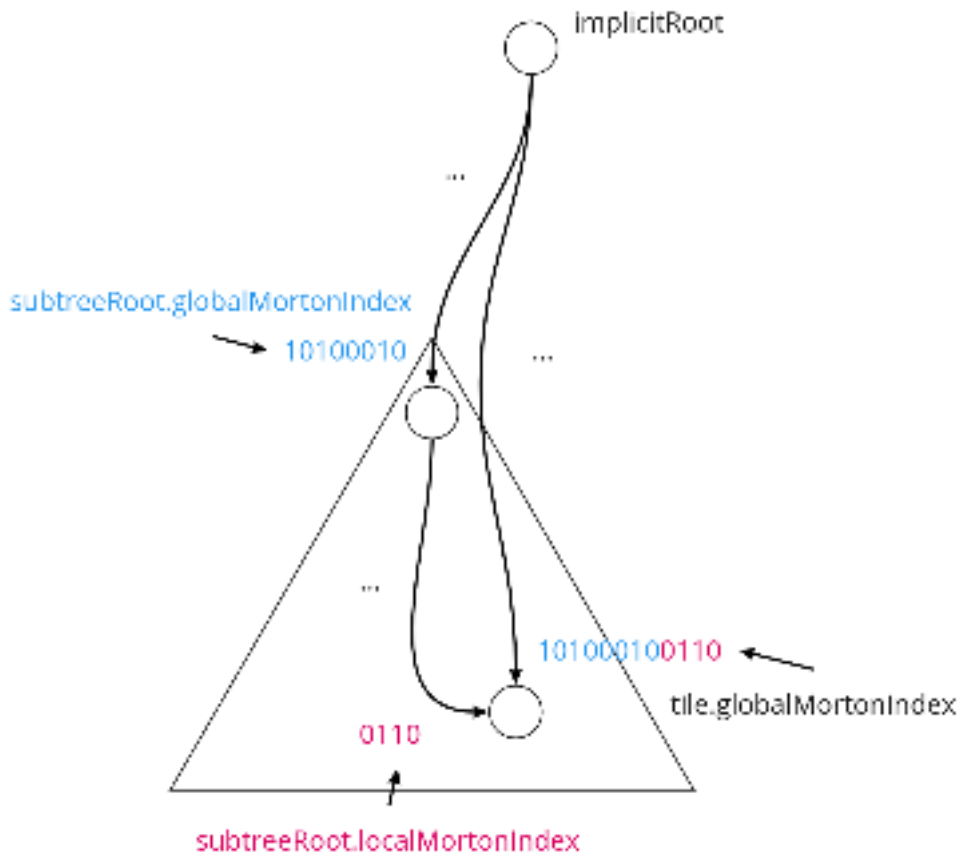


Figure 67. Illustration of how to compute the global Morton index of a tile, from the global Morton index of the root of the containing subtree, and the local Morton index of the tile in this subtree

Similarly, the global level of a tile is the length of the path from the implicit root tile to the tile. This is the sum of the subtree root tile's global level and the tile's local level relative to the subtree root tile:

$$\text{tile.globalLevel} = \text{subtreeRoot.globalLevel} + \text{tile.localLevel}$$

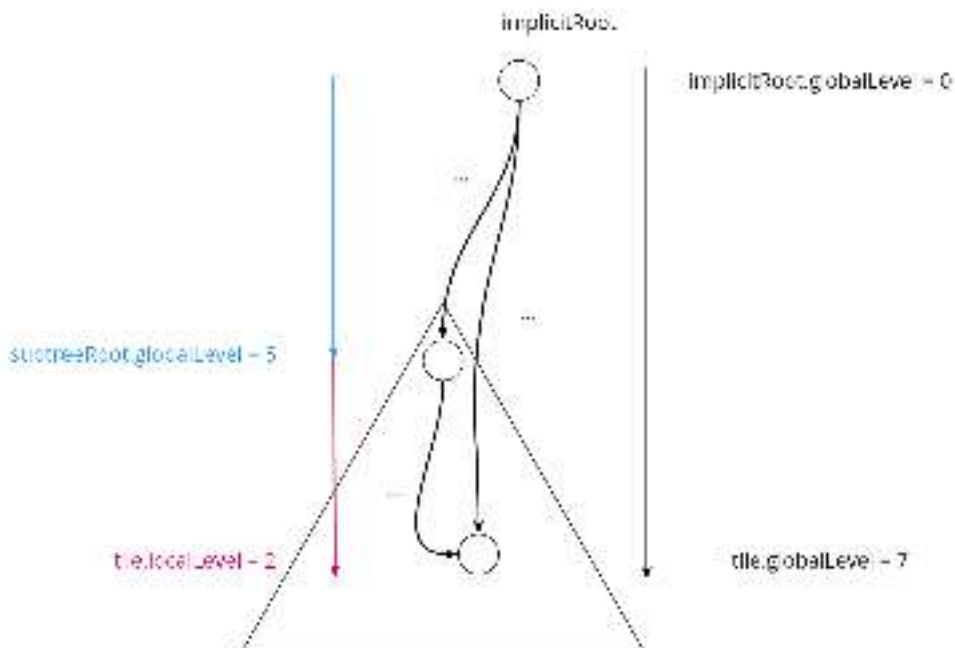


Figure 68. Illustration of how to compute the global level of a tile, from the global level of the root of the containing subtree, and the local level of the tile in this subtree

$(x, y, z)$  coordinates follow the same pattern as Morton indices. The only difference is that the concatenation of bits happens component-wise. That is:

```

tile.globalX = concatBits(subtreeRoot.globalX, tile.localX)
tile.globalY = concatBits(subtreeRoot.globalY, tile.localY)

// Octrees only
tile.globalZ = concatBits(subtreeRoot.globalZ, tile.localZ)

```

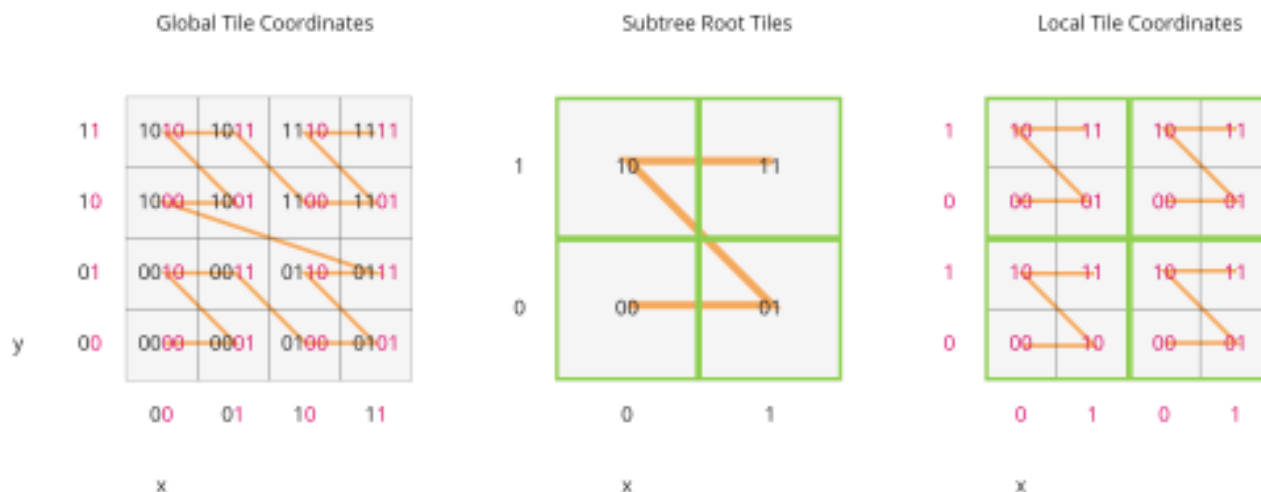


Figure 69. Illustration of the computation of the global tile coordinates, from the global coordinates of the containing subtree, and the local coordinates of the tile in this subtree.

## Finding Parent and Child Tiles

The coordinates of a parent or child tile can also be computed with bitwise operations on the Morton index. The following formulas apply for both local and global coordinates.

```

childTile.level = parentTile.level + 1
childTile.mortonIndex = concatBits(parentTile.mortonIndex, childIndex)
childTile.x = concatBits(parentTile.x, childX)
childTile.y = concatBits(parentTile.y, childY)

// Octrees only
childTile.z = concatBits(parentTile.z, childZ)

```

Where:

- `childIndex` is an integer in the range  $[0, N)$  that is the index of the child tile relative to the parent.
- `childX`, `childY`, and `childZ` are single bits that represent which half of the parent's bounding volume the child is in in each direction.

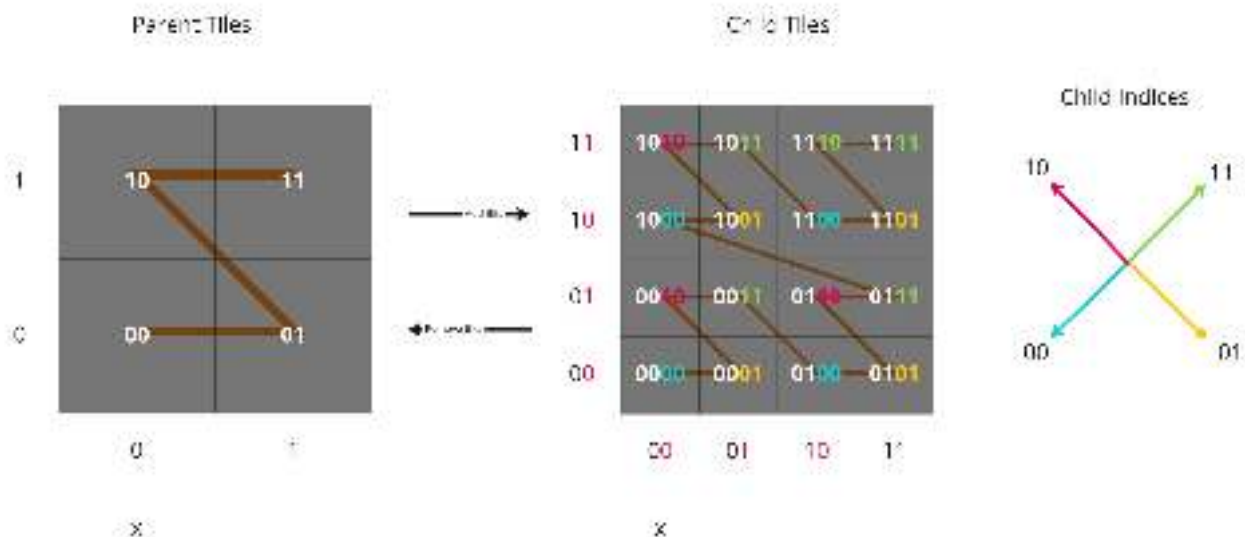


Figure 70. Illustration of the computation of the coordinates of parent- and child tiles

## Appendix C: 3D Metadata Reference Implementation

This document defines a reference implementation of the concepts defined in the [3D Metadata Specification](#). The 3D Metadata Specification itself defines a standard format for structured metadata in 3D content in a way that is language- and format agnostic. The reference implementation described here is an implementation of these concepts:

- The [Schema](#) is a JSON-based representation of [3D Metadata Schemas](#) that describe the structure and types of metadata
- The [PropertyTable](#) is one form of a [Binary Table Format](#). It is a JSON-based description of how

large amounts of metadata can be stored compactly in a binary form.

These serialization formats are used as a common basis for different implementations of the 3D Metadata Specification:

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

## Property Table Implementation

The [3D Metadata Specification](#) defines schemas as a description of the structure of metadata, and different storage formats for the metadata. One form of storing metadata is that of a [Binary Table Format](#), where the data is stored in a binary representation of a table. Each column of such a table represents one of the properties of a class. Each row represents a single entity conforming to the class. The following is the description of such a binary table format, referred to as **property table**. It is used as the basis for defining the metadata storage in the following implementations:

- [3D Tiles Metadata Implicit Tilesets](#) - Assigns metadata to tilesets, tiles, groups, and contents in a 3D Tiles tileset. A property table is defined for subtrees of an implicit tile hierarchy, and stores metadata that is associated with the nodes of such a subtree.
- [EXT\\_structural\\_metadata](#) — Assigns metadata to vertices, texels, and features in a glTF asset. A property table is defined in the top-level extension object. The property values are stored in standard glTF buffer views.

The full JSON schema definition for this implementation can be found in [the PropertyTable directory of the specification](#).

### Property Tables

Defined in [propertyTable.schema.json](#).

A property table must specify the following elements:

- Its class (**class**), which refers to a class ID in a [Schema](#).
- A dictionary of properties (**properties**), where each entry describes the binary storage of the data for the corresponding class property.
- A count (**count**) for the number of elements (rows) in the property table.

The property table may provide value arrays for only a subset of the properties of its class, but class properties that are marked **required: true** in the schema must not be omitted.

### Example

A `tree_survey_2021-09-29` property table, implementing the `tree` class defined in the [Schema](#) examples. The table contains observations for 10 trees. Details about the class properties will be given in later examples.

#### NOTE

```
"schema": { ... },
"propertyTables": [{
  "name": "tree_survey_2021-09-29",
  "class": "tree",
  "count": 10,
  "properties": {
    "species": { ... },
    "age": { ... },
    "height": { ... },
    // "diameter" is not required and has been omitted from this table.
  }
}]
```

## Property Table Properties

Defined in `propertyTable.property.schema.json`.

Each property definition in a property table represents one column of the table. This column data is stored in binary form, using the encoding defined in the [Binary Table Format](#) section of the 3D Metadata Specification. The actual data is stored in a binary buffer, and the property refers to a section of this buffer that is called a *buffer view*.

- In the [3D Tiles Metadata](#) implementation, a buffer view is defined as part of [subtrees in implicit tilesets](#).
- In the [3DTILES\\_metadata](#) extension for 3D Tiles 1.0, a buffer view is defined as part of [subtrees the 3DTILES\\_implicit\\_tiling extension](#).
- In the [EXT\\_structural\\_metadata](#), a buffer view is a standard glTF buffer view.

The exact structure of each property entry depends on the [property type](#):

- Every property definition must define the `values` that store the raw data of the actual values.
- Properties that have the `STRING` component type must define the `stringOffsets`, as defined in [Strings](#).
- Properties that are variable-length arrays must define the `arrayOffsets`, as defined in [Arrays](#).

For variable-length arrays of strings, both the `stringOffsets` and the `arrayOffsets` are required.

### Example

The property table from the previous example, with details about the binary storage of the property values

#### NOTE

```
{
  "propertyTables": [{
    "name": "tree_survey_2021-09-29",
    "class": "tree",
    "count": 10,
    "properties": {
      "species": {
        "values": 2,
        "stringOffsets": 3
      },
      "age": {
        "values": 1
      },
      "height": {
        "values": 0
      },
      // "diameter" is not required and has been omitted from this
      table.
    }
  ]
}
```

Each buffer view `byteOffset` must be aligned to a multiple of the size of the `componentType` of the respective property.

### Implementation Note

#### NOTE

Authoring tools may choose to align all buffer views to 8-byte boundaries for consistency, but client implementations should only depend on 8-byte alignment for buffer views containing 64-bit component types.

For the `arrayOffsets` and `stringOffsets` buffer views, the property can also define the `arrayOffsetType` and `stringOffsetType`, which describe the storage type for array- and string offsets, respectively. Allowed types are `UINT8`, `UINT16`, `UINT32`, and `UINT64`. The default is `UINT32`.

A property may override the `minimum and maximum values` and the `offset and scale` from the property definition in the class, to account for the actual range of values that is stored in the property table.

## Schema Implementation

The [3D Metadata Specification](#) defines `schemas` as a description of the structure of metadata, consisting of classes with different properties, and enum types. The following is the description of a JSON-based representation of such a schema and its elements. It is used as the basis for defining the metadata structure in the following implementations:

- [3D Tiles Metadata Implicit Tilesets](#) - Assigns metadata to tilesets, tiles, groups, and contents in a 3D Tiles tileset. A property table is defined for subtrees of an implicit tile hierarchy, and stores metadata that is associated with the nodes of such a subtree.
- [EXT\\_structural\\_metadata](#) — Assigns metadata to vertices, texels, and features in a glTF asset. A property table is defined in the top-level extension object. The property values are stored in standard glTF buffer views.

The full JSON schema definition for this implementation can be found in [the Schema directory of the specification](#).

## Schema

Defined in [schema.schema.json](#).

A schema defines a set of classes and enums. Classes serve as templates for entities. They provide a list of properties and the type information for those properties. Enums define the allowable values for enum properties.

### Example

Schema with a `tree` class, and a `speciesEnum` enum that defines different species of trees. Later examples show how these structures in more detail.

NOTE

```
{
  "schema": {
    "classes": {
      "tree": { ... },
    },
    "enums": {
      "speciesEnum": { ... }
    }
  }
}
```

## Class

Defined in [class.schema.json](#).

A class is a template for metadata entities. Classes provide a list of property definitions. Every entity must be associated with a class, and the entity's properties must conform to the class's property definitions. Entities whose properties conform to a class are considered instances of that class.

Classes are defined as entries in the `schema.classes` dictionary, indexed by class ID. Class IDs must be [identifiers](#) as defined in the 3D Metadata Specification.



### Example

A "Tree" class, which might describe a table of tree measurements taken in a park. Property definitions are abbreviated here, and introduced in the next section.

NOTE

```
{
  "schema": {
    "classes": {
      "tree": {
        "name": "Tree",
        "description": "Woody, perennial plant.",
        "properties": {
          "species": { ... },
          "age": { ... },
          "height": { ... },
          "diameter": { ... }
        }
      }
    }
  }
}
```

## Class Property

Defined in [class.property.schema.json](#).

Class properties are defined abstractly in a class. The class is instantiated with specific values conforming to these properties. Class properties support a rich variety of data types. Details about the supported types can be found in the [3D Metadata Specification](#).

Class properties are defined as entries in the `class.properties` dictionary, indexed by property ID. Property IDs must be [identifiers](#) as defined in the 3D Metadata Specification.

### Example

A "Tree" class, which might describe a table of tree measurements taken in a park. Properties include species, age, height, and diameter of each tree.

NOTE

```
{
  "schema": {
    "classes": {
      "tree": {
        "name": "Tree",
        "description": "Woody, perennial plant.",
        "properties": {
          "species": {
            "description": "Type of tree.",
            "type": "ENUM",
            "enumType": "speciesEnum",
            "required": true
          },
          "age": {
            "description": "The age of the tree, in years",
            "type": "SCALAR",
            "componentType": "UINT8",
            "required": true
          },
          "height": {
            "description": "Height of tree measured from ground level,
in meters.",
            "type": "SCALAR",
            "componentType": "FLOAT32"
          },
          "diameter": {
            "description": "Diameter at trunk base, in meters.",
            "type": "SCALAR",
            "componentType": "FLOAT32"
          }
        }
      }
    }
  }
}
```

## Enum

Defined in [enum.schema.json](#).

A set of categorical types, defined as (*name*, *value*) pairs. Enum properties use an enum as their type.

Enums are defined as entries in the `schema.enums` dictionary, indexed by an enum ID. Enum IDs must be [identifiers](#) as defined in the 3D Metadata Specification.

### Example

A "Species" enum defining types of trees. An "Unspecified" enum value is optional, but when provided as the `noData` value for a property (see: [3D Metadata - No Data Values](#)) may be helpful to identify missing data.

#### NOTE

```
{
  "schema": {
    "enums": {
      "speciesEnum": {
        "name": "Species",
        "description": "An example enum for tree species.",
        "values": [
          {"name": "Unspecified", "value": 0},
          {"name": "Oak", "value": 1},
          {"name": "Pine", "value": 2},
          {"name": "Maple", "value": 3}
        ]
      }
    }
  }
}
```

### Enum Value

Defined in `enum.value.schema.json`.

Pairs of (`name`, `value`) entries representing possible values of an enum property.

Enum values are defined as entries in the `enum.values` array. Duplicate names or duplicate integer values are not allowed.

## Appendix D: 3D Metadata Semantic Reference

### Overview

This document provides common definitions of meaning ("semantics") used by metadata properties in 3D Tiles and glTF. Tileset authors may define their own application- or domain-specific semantics separately.

Semantics describe how properties should be interpreted. For example, an application that encounters the `ID` or `NAME` semantics while parsing a dataset may use these values as unique identifiers or human-readable labels, respectively.

Each semantic is defined in terms of its meaning, and the datatypes it may assume. Datatype specifications include "type" as defined by the [3D Metadata Specification](#). When applicable they

may also include "component type", "array", and "count".

For use of semantics, see:

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

## General

### Overview

Throughout this section, the term "entity" refers to any conceptual object with which a property value (as defined in the [3D Metadata Specification](#)) may be associated. Examples of entities include tilesets, tiles, and tile contents in 3D Tiles, or groups of vertices and texels in glTF 2.0 assets. Additional types of entities may be defined by other specifications or applications.

### General Semantics

Semantic	Type	Description
<a href="#">ID</a>	<ul style="list-style-type: none"><li>• Type: <a href="#">STRING</a></li></ul>	The unique identifier for the entity.
<a href="#">NAME</a>	<ul style="list-style-type: none"><li>• Type: <a href="#">STRING</a></li></ul>	The name of the entity. Names should be human-readable, and do not have to be unique.
<a href="#">DESCRIPTION</a>	<ul style="list-style-type: none"><li>• Type: <a href="#">STRING</a></li></ul>	Description of the entity. Typically at least a phrase, and possibly several sentences or paragraphs.
<a href="#">ATTRIBUTION_IDS</a>	<ul style="list-style-type: none"><li>• Type: <a href="#">SCALAR</a></li><li>• Component type: <a href="#">UINT8</a>, <a href="#">UINT16</a>, <a href="#">UINT32</a>, or <a href="#">UINT64</a></li><li>• Array: <a href="#">true</a></li></ul>	List of attribution IDs that index into a global list of attribution strings. This semantic may be assigned to metadata at any level of granularity including tileset, group, subtree, tile, content, feature, vertex, and texel granularity. The global list of attribution strings is located in a tileset or subtree with the property semantic <a href="#">ATTRIBUTION_STRINGS</a> . The following precedence order is used to locate the attribution strings: first the containing subtree (if applicable), then the containing external tileset (if applicable), and finally the root tileset.

Semantic	Type	Description
<code>ATTRIBUTION_STRINGS</code>	<ul style="list-style-type: none"> <li>Type: <code>STRING</code></li> <li>Array: <code>true</code></li> </ul>	List of attribution strings. Each string contains information about a data provider or copyright text. Text may include embedded markup languages such as HTML. This semantic may be assigned to metadata at any granularity (wherever <code>STRING</code> property values can be encoded). When used in combination with <code>ATTRIBUTION_IDS</code> it is assigned to subtrees and tilesets.

## 3D Tiles

### Overview

Semantics for 3D Tiles are assigned in relationship to a tileset, subtree, tile, or tile content, as defined by the 3D Tiles specification. When associated with other types of entities, these semantics may have invalid or undefined meanings.

Units for all linear distances are meters, and all angles are radians.

### Tileset

#### Overview

`TILESET_*` semantics provide meaning for properties associated with a tileset.

#### Tileset Semantics

Semantic	Type	Description
<code>TILESET_FEATURE_ID_LABELS</code>	<ul style="list-style-type: none"> <li>Type: <code>STRING</code></li> <li>Array: <code>true</code></li> </ul>	The union of all the feature ID labels in glTF content using the <code>EXT_mesh_features</code> and <code>EXT_instance_features</code> extensions.

Semantic	Type	Description
TILESET_CRS_GEOCENTRIC	<ul style="list-style-type: none"> <li>Type: <b>STRING</b></li> </ul>	<p>The geocentric coordinate reference system (CRS) of the tileset. Values include, but are not limited to:</p> <ul style="list-style-type: none"> <li>"EPSG:4978" - WGS 84</li> <li>"EPSG:7656" - WGS 84 (G730)</li> <li>"EPSG:7658" - WGS 84 (G873)</li> <li>"EPSG:7660" - WGS 84 (G1150)</li> <li>"EPSG:7662" - WGS 84 (G1674)</li> <li>"EPSG:7664" - WGS 84 (G1762)</li> <li>"EPSG:9753" - WGS 84 (G2139)</li> <li>"UNKNOWN" - CRS is unknown</li> </ul> <p><b>region</b> bounding volumes are assumed to use the same reference ellipsoid as the geocentric coordinate reference system specified here.</p> <p>For more details on coordinate reference systems in 3D Tiles, see <a href="#">Coordinate Reference System (CRS)</a>.</p>
TILESET_CRS_COORDINATE_EPOCH	<ul style="list-style-type: none"> <li>Type: <b>STRING</b></li> </ul>	<p>The coordinate epoch for coordinates that are referenced to a dynamic CRS such as WGS 84. Coordinates include glTF vertex positions after transforms have been applied—see <a href="#">glTF transforms</a>. Expressed as a decimal year (e.g. "2019.81"). See <a href="#">WKT representation of coordinate epoch and coordinate metadata</a> for more details.</p>

## Tile

### Overview

**TILE\_\*** semantics provide meaning for properties associated with a particular tile, and should take precedence over equivalent metadata on parent objects, as well as over values derived from subdivision schemes in [Implicit Tiling](#).

If property values are missing, either because the property is omitted or the property table contains **noData** values, the original tile properties are used, such as those explicitly defined in tileset JSON or implicitly computed from subdivision schemes in [Implicit Tiling](#).

In particular, **TILE\_BOUNDING\_BOX**, **TILE\_BOUNDING\_REGION**, and **TILE\_BOUNDING\_SPHERE** semantics each define a more specific bounding volume for a tile than is implicitly calculated from [Implicit Tiling](#).

If more than one of these semantics are available for a tile, clients may select the most appropriate option based on use case and performance requirements.

## Tile Semantics

Semantic	Type	Description
TILE_BOUNDING_BOX	<ul style="list-style-type: none"> <li>Type: SCALAR</li> <li>Component type: FLOAT32 or FLOAT64</li> <li>Array: true</li> <li>Count: 12</li> </ul>	The bounding volume of the tile, expressed as a <a href="#">box</a> . Equivalent to <code>tile.boundingVolume.box</code> .
TILE_BOUNDING_REGION	<ul style="list-style-type: none"> <li>Type: SCALAR</li> <li>Component type: FLOAT64</li> <li>Array: true</li> <li>Count: 6</li> </ul>	The bounding volume of the tile, expressed as a <a href="#">region</a> . Equivalent to <code>tile.boundingVolume.region</code> .
TILE_BOUNDING_SPHERE	<ul style="list-style-type: none"> <li>Type: SCALAR</li> <li>Component type: FLOAT32 or FLOAT64</li> <li>Array: true</li> <li>Count: 4</li> </ul>	The bounding volume of the tile, expressed as a <a href="#">sphere</a> . Equivalent to <code>tile.boundingVolume.sphere</code> .
TILE_BOUNDING_S2_CELL	<ul style="list-style-type: none"> <li>Type: SCALAR</li> <li>Component type: UINT64</li> </ul>	The bounding volume of the tile, expressed as an <a href="#">S2 Cell ID</a> using the 64-bit representation instead of the hexadecimal representation. Only applicable to <code>`3DTILES_bounding_volume_S2</code> .
TILE_MINIMUM_HEIGHT	<ul style="list-style-type: none"> <li>Type: SCALAR</li> <li>Component type: FLOAT32 or FLOAT64</li> </ul>	The minimum height of the tile above (or below) the WGS84 ellipsoid. Equivalent to minimum height component of <code>TILE_BOUNDING_REGION</code> and <code>tile.boundingVolume.region</code> . Also equivalent to minimum height component of <code>`3DTILES_bounding_volume_S2</code> .
TILE_MAXIMUM_HEIGHT	<ul style="list-style-type: none"> <li>Type: SCALAR</li> <li>Component type: FLOAT32 or FLOAT64</li> </ul>	The maximum height of the tile above (or below) the WGS84 ellipsoid. Equivalent to maximum height component of <code>TILE_BOUNDING_REGION</code> and <code>tile.boundingVolume.region</code> . Also equivalent to maximum height component of <code>`3DTILES_bounding_volume_S2</code> .

Semantic	Type	Description
<code>TILE_HORIZON_OCCLUSION_POINT</code> <sup>1</sup>	<ul style="list-style-type: none"> <li>Type: <code>VEC3</code></li> <li>Component type: <code>FLOAT32</code> or <code>FLOAT64</code></li> </ul>	The horizon occlusion point of the tile expressed in an ellipsoid-scaled fixed frame. If this point is below the horizon, the entire tile is below the horizon. See <a href="#">Horizon Culling</a> for more information.
<code>TILE_GEOMETRIC_ERROR</code>	<ul style="list-style-type: none"> <li>Type: <code>SCALAR</code></li> <li>Component type: <code>FLOAT32</code> or <code>FLOAT64</code></li> </ul>	The geometric error of the tile. Equivalent to <code>tile.geometricError</code> .
<code>TILE_REFINE</code>	<ul style="list-style-type: none"> <li>Type: <code>SCALAR</code></li> <li>Component type: <code>UINT8</code></li> </ul>	The tile refinement type. Valid values are <code>0</code> ("ADD") and <code>1</code> ("REPLACE"). Equivalent to <code>tile.refine</code> .
<code>TILE_TRANSFORM</code>	<ul style="list-style-type: none"> <li>Type: <code>MAT4</code></li> <li>Component type: <code>FLOAT32</code> or <code>FLOAT64</code></li> </ul>	The tile transform. Equivalent to <code>tile.transform</code> .

<sup>1</sup> `TILE_HORIZON_OCCLUSION_POINT` should account for all content in a tile and its descendants, whereas `CONTENT_HORIZON_OCCLUSION_POINT` should only account for content in a tile. When the two values are equivalent, only `TILE_HORIZON_OCCLUSION_POINT` should be specified.

## Content

### Overview

`CONTENT_*` semantics provide meaning for properties associated with the content of a tile, and may be more specific to that content than properties of the containing tile.

`CONTENT_BOUNDING_BOX`, `CONTENT_BOUNDING_REGION`, and `CONTENT_BOUNDING_SPHERE` semantics each define a more specific bounding volume for tile contents than the bounding volume of the tile. If more than one of these semantics are available for the same content, clients may select the most appropriate option based on use case and performance requirements.

### Content Semantics

Semantic	Type	Description
<code>CONTENT_BOUNDING_BOX</code>	<ul style="list-style-type: none"> <li>Type: <code>SCALAR</code></li> <li>Component type: <code>FLOAT32</code> or <code>FLOAT64</code></li> <li>Array: <code>true</code></li> <li>Count: <code>12</code></li> </ul>	The bounding volume of the content of a tile, expressed as a <code>box</code> . Equivalent to <code>tile.content.boundingVolume.box</code> .



Semantic	Type	Description
<code>CONTENT_BOUNDING_REGION</code>	<ul style="list-style-type: none"> <li>Type: <code>SCALAR</code></li> <li>Component type: <code>FLOAT64</code></li> <li>Array: <code>true</code></li> <li>Count: <code>6</code></li> </ul>	The bounding volume of the content of a tile, expressed as a <a href="#">region</a> . Equivalent to <code>tile.content.boundingBoxVolume.region</code> .
<code>CONTENT_BOUNDING_SPHERE</code>	<ul style="list-style-type: none"> <li>Type: <code>SCALAR</code></li> <li>Component type: <code>FLOAT32</code> or <code>FLOAT64</code></li> <li>Array: <code>true</code></li> <li>Count: <code>4</code></li> </ul>	The bounding volume of the content of a tile, expressed as a <a href="#">sphere</a> . Equivalent to <code>tile.content.boundingBoxVolume.sphere</code> .
<code>CONTENT_BOUNDING_S2_CELL</code>	<ul style="list-style-type: none"> <li>Type: <code>SCALAR</code></li> <li>Component type: <code>UINT64</code></li> </ul>	The bounding volume of the content of a tile, expressed as an <a href="#">S2 Cell ID</a> using the 64-bit representation instead of the hexadecimal representation. Only applicable to <code>`3DTILES_boundingBoxVolume_S2</code> .
<code>CONTENT_MINIMUM_HEIGHT</code>	<ul style="list-style-type: none"> <li>Type: <code>SCALAR</code></li> <li>Component type: <code>FLOAT32</code> or <code>FLOAT64</code></li> </ul>	The minimum height of the content of a tile above (or below) the WGS84 ellipsoid. Equivalent to minimum height component of <code>CONTENT_BOUNDING_REGION</code> and <code>tile.content.boundingBoxVolume.region</code> . Also equivalent to minimum height component of <code>`3DTILES_boundingBoxVolume_S2</code> .
<code>CONTENT_MAXIMUM_HEIGHT</code>	<ul style="list-style-type: none"> <li>Type: <code>SCALAR</code></li> <li>Component type: <code>FLOAT32</code> or <code>FLOAT64</code></li> </ul>	The maximum height of the content of a tile above (or below) the WGS84 ellipsoid. Equivalent to maximum height component of <code>CONTENT_BOUNDING_REGION</code> and <code>tile.content.boundingBoxVolume.region</code> . Also equivalent to maximum height component of <code>`3DTILES_boundingBoxVolume_S2</code> .
<code>CONTENT_HORIZON_OCCLUSION_POINT<sup>1</sup></code>	<ul style="list-style-type: none"> <li>Type: <code>VEC3</code></li> <li>Component type: <code>FLOAT32</code> or <code>FLOAT64</code></li> </ul>	The horizon occlusion point of the content of a tile expressed in an ellipsoid-scaled fixed frame. If this point is below the horizon, the entire content is below the horizon. See <a href="#">Horizon Culling</a> for more information.
<code>CONTENT_URI</code>	<ul style="list-style-type: none"> <li>Type: <code>STRING</code></li> </ul>	The content uri. Overrides the implicit tile's generated content uri. Equivalent to <code>tile.content.uri</code> .

Semantic	Type	Description
<code>CONTENT_GROUP_ID</code>	<ul style="list-style-type: none"> <li>Type: <code>SCALAR</code></li> <li>Component type: <code>UINT8</code>, <code>UINT16</code>, <code>UINT32</code>, or <code>UINT64</code></li> </ul>	The content group ID. Equivalent to <code>tile.content.group</code> .

<sup>1</sup>`TILE_HORIZON_OCCLUSION_POINT` should account for all content in a tile and its descendants, whereas `CONTENT_HORIZON_OCCLUSION_POINT` should only account for content in a tile. When the two values are equivalent, only `TILE_HORIZON_OCCLUSION_POINT` should be specified.

## Appendix E: Acknowledgements

Editors:

- Patrick Cozzi, [@pjcozzi](#), [patrick@cesium.com](mailto:patrick@cesium.com)
- Sean Lilley, [@lilleyse](#), [sean@cesium.com](mailto:sean@cesium.com)
- Gabby Getz, [@gabbygetz](#), [gabby@cesium.com](mailto:gabby@cesium.com)

Acknowledgements:

- Matt Amato, [@matt\\_amato](#)
- Erik Andersson, [@e-andersson](#)
- Dan Bagnell, [@bagnell](#)
- Ray Bentley
- Jannes Bolling, [@jbo023](#)
- Dylan Brown, [@Dylan-Brown](#)
- Sarah Chow, [cesium.com/team/SarahChow](http://cesium.com/team/SarahChow)
- Paul Connelly
- Volker Coors
- Tom Fili, [@CesiumFili](#)
- Leesa Fini, [@LeesaFini](#)
- Ralf Gutbell
- Frederic Houbie
- Christopher Mitchell, Ph.D., [@KermMartian](#)
- Claus Nagel
- Jean-Philippe Pons
- Carl Reed
- Kevin Ring, [www.kotachrome.com/kevin](http://www.kotachrome.com/kevin)

- Scott Simmons
- Rob Taglang, [@lasalvavida](#)
- Stan Tillman
- Piero Toffanin, [@pierotofy](#)
- Pano Voudouris
- Dave Wesloh

## Appendix F: License

Copyright 2016 - 2022 Cesium GS, Inc.

This Specification is licensed under a [Creative Commons Attribution 4.0 International License \(CC BY 4.0\)](#).

The companies listed above have granted the Open Geospatial Consortium (OGC) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version under a Attribution 4.0 International (CC BY 4.0) license.

Some parts of this Specification are purely informative and do not define requirements necessary for compliance and so are outside the Scope of this Specification. These parts of the Specification are marked as being non-normative, or identified as **Implementation Notes**.