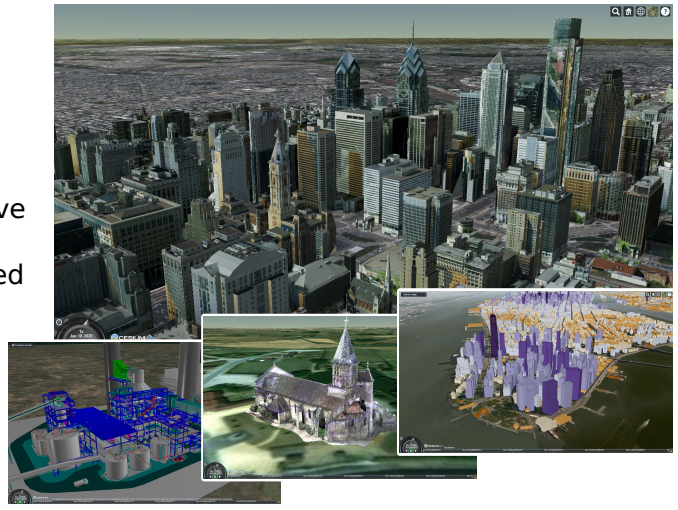
3D Tiles

3D Tiles is an open specification for sharing, visualizing, fusing, interacting with, and analyzing massive heterogeneous 3D geospatial content across desktop, web, and mobile applications. 3D Tiles is built on glTF, an open standard for efficient streaming and rendering of 3D models and scenes.



3D geospatial content, including photogrammetry/massive models, BIM/CAD, 3D buildings, instanced features, and point clouds, can be converted into 3D Tiles and combined into a single dataset for seamless performance and real-time analytics including measurements, visibility analysis, styling and filtering.

The foundation of 3D Tiles is a spatial data structure that enables Hierarchical Level of Detail (HLOD) so only visible tiles are streamed and rendered, improving overall performance.

This overview summarizes the main concepts that are supported by the 3D Tiles specification:

• The general concepts of tilesets and tiles, and how they make it possible to organize massive datasets into elements that can be streamed efficiently	1. At a Glance: An Example Tileset 2. Tilesets and Tiles
• How 3D Tiles implements hierarchical spatial data structures that are used for efficient rendering and interaction.	3. Bounding Volumes 4. Spatial Data Structures
• The concept of a Hierarchical Level of Detail (HLOD), which makes it possible to balance rendering performance and visual quality at any scale.	5. Geometric Error 6. Refinement Strategies
• How the concepts behind 3D Tiles can be implemented for efficient rendering and interaction	7. Optimized Rendering with 3D Tiles 8. Spatial Queries in 3D Tiles
• The technical details of the different tile formats in 3D Tiles	9. Tile Formats: Introduction 10. Tile Formats
• The possibility to extend the base specification with additional features	11. Extensions
• The capabilities for information visualization by styling the content based on metadata	12. Declarative Styling
• How basic elements like geospatial coordinate systems and geometry data compression are integrated in 3D Tiles	13. Common Definitions

If you are looking for 3D Tilesets to get started, Cesium ion (https://cesium.com/ion) hosts curated 3D Tilesets and also allows users to upload their own data to create, host, and stream 3D Tiles.

CESIUM®

1. At a Glance: An Example Tileset

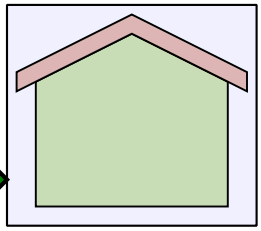
The core element of 3D Tiles is *tilesets*. A tileset is a set of *tiles*, organized in a hierarchical structure. The tileset is described in JSON. The following is a simple example of a tileset, introducing the most important concepts and elements. Each concept is explained in more detail in the following sections.

```
{
  "asset": { ... },
  "properties": { ... },
  "geometricError": 100,
  "root": {
    "geometricError": 20,
    "boundingVolume": {
      "region": [ ... ]
    },
    "refine": "ADD",
    "children": [
      {
        "geometricError": 10,
        "boundingVolume": { ... },
        "content": {
          "uri": "house.b3dm"
        },
        "children": [
          {
            "geometricError": 5,
            "boundingVolume": { ... },
            "content": {
              "uri": "detailsA.b3dm"
            },
          }, {
            "geometricError": 5,
            "boundingVolume": { ... },
            "content": {
              "uri": "detailsB.b3dm"
            },
          }
        ]
      }, {
        "geometricError": 10,
        "boundingVolume": { ... },
        "content": {
          "uri": "tree.pnts"
        },
      }, {
        "geometricError": 10,
        "boundingVolume": { ... },
        "content": {
          "uri": "fence.i3dm"
        },
      }, {
        "geometricError": 10,
        "boundingVolume": { ... },
        "content": {
          "uri": "external.json"
        },
      }
    ]
  }
}
```

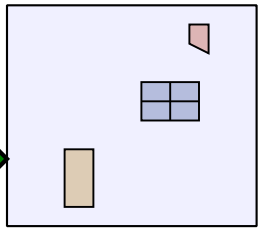
Each tile may refer to an external tileset. This allows combining multiple smaller tilesets into larger ones.

The main tileset JSON contains a basic description of the asset and general properties.
The *geometric error* (Section 5) is used to determine when the root tile should be rendered.

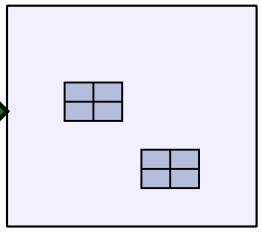
Each tile contains a *bounding volume* (Section 3) that encloses the content of the tile and all children. It also contains a geometric error that determines when the children should be rendered.



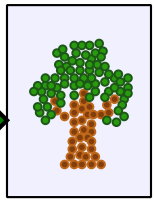
In this example, the first child tile refers to a 3D model with a low level of detail. The tile format of the model in the example is a *Batched 3D Model* (Section 10.2)



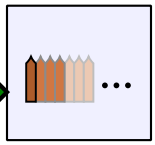
The child tiles contain additional details for the model. They are rendered when a higher level of detail is requested.



In this example, the contents of the child tiles will be added to the low-level representation of the model of the parent node. Alternatively, the child tiles could contain more detailed models that replace the low-level model. These are the two *refinement strategies* (Section 6) that are supported by 3D Tiles.

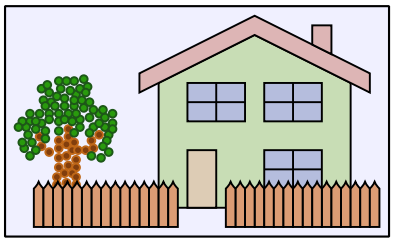


Different tile formats can be combined in one tileset: in this example, the first child tile referred to a Batched 3D Model. Another child tile of the root node refers to a *Point Cloud* (Section 10.4).



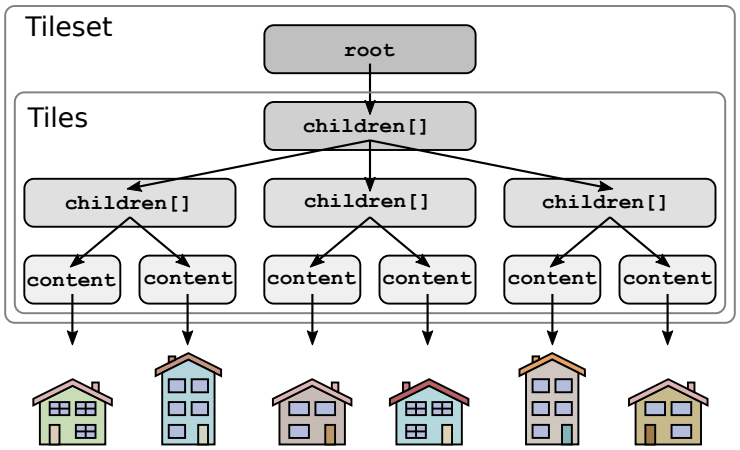
Another tile is an *Instanced 3D Model* (Section 10.3) where a simple geometry is rendered multiple times, at different positions.

When the tileset is rendered, the tiles are combined, at the appropriate level of detail, to generate the final rendered result:



2. Tilesets and Tiles

A tileset is a set of tiles, which are organized as a hierarchical data structure, called a *tree*. The tileset itself contains the root tile, and each tile may have child tiles.
A tile can also refer to another tileset. This makes it possible to flexibly and hierarchically combine tilesets into larger ones.
Each tile may refer to renderable content. This content can have different formats and may represent, for example, textured terrain surfaces, 3D models, or point clouds. The different possible tile formats are explained in Section 9, "Tile Formats: Introduction."



Both tilesets and tiles are described in JSON. The tileset JSON file contains basic information about the tileset itself, and the description of the tiles.

Tileset Properties:

Root tile: The `root` property of a tileset is a tile that represents the root of the tile hierarchy.

Geometric error: The `geometricError` property is used to quantify the visual error that would occur if the tileset was not rendered. When the visual error exceeds a certain threshold, then the tileset and the tiles that it contains are considered for rendering. Details of how the geometric error is used are given in Section 5, "Geometric Error."

Property summaries: The renderable content of tiles may have associated properties. For example, when the tiles contain buildings, then the height of each building can be stored in the tile. The `properties` object of a tileset contains the minimum and the maximum of select property values for all tiles in the tileset.

Metadata: Information about the 3D Tiles version and application-specific version information can be stored in the `asset` property of a tileset.

Tile Properties:

Content: The actual renderable content that is associated with a tile is referred to via a URI in the `content` property.

Children: The hierarchical structure of tiles is modeled as a tree: each tile may have children, which are tiles in an array named `children`.

Bounding Volume: Each tile has an associated bounding volume, and different types of bounding volumes may be stored in the `boundingVolume` property. The possible types of bounding volumes are presented in Section 3, "Bounding Volumes." Each bounding volume encloses the content of the tile and the content of all children, yielding a spatially coherent hierarchy of bounding volumes.

Geometric error: The renderable content of tiles may have different levels of detail. For tiles, the `geometricError` property quantifies the degree of simplification of the content in the tile compared to the highest level of detail. The geometric error is used as described in Section 5, "Geometric Error," to determine when the child tiles should be considered for rendering.

Refinement strategy: When the visual error of a tile with a certain level of detail exceeds a threshold, then the child tiles are considered for rendering. The way the additional detail from the child tiles is incorporated into the rendering process is determined by the `refine` property. The different refinement strategies are explained in Section 6, "Refinement."

3. Bounding Volumes

Each tileset is a set of tiles that are organized in a hierarchical manner, and each tile has an associated *bounding volume*. This yields a hierarchical spatial data structure that can be used for optimized rendering and efficient spatial queries. Additionally, each tile can contain actual renderable content that also has a bounding volume. In contrast to the tile bounding volume, the content bounding volume fits tightly only to the content and can be used for visibility queries and view frustum culling to further increase the rendering performance.

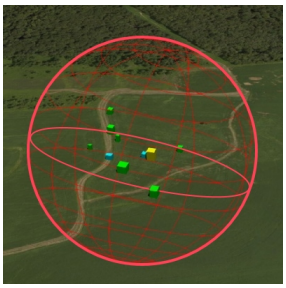
The 3D Tiles format supports different types of bounding volumes, and different strategies for their hierarchical organization.

Types of Bounding Volumes

Different types of bounding volumes are supported in 3D Tiles, making it possible to choose the type that is best suited for the structure of the underlying data:

```
"boundingVolume": {
  "sphere": [
    10, 5, 15,
    140.0
  ]
}
```

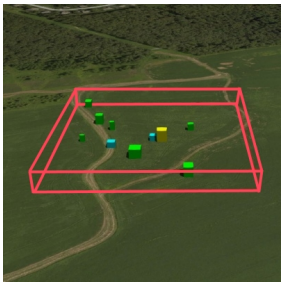
Center
Radius



A *bounding sphere* is a simple bounding volume that allows easy and efficient intersection tests. It is defined by its center position and the radius.

```
"boundingVolume": {
  "box": [
    0,  0,  10,
    20, 0,  0,
    0,  30, 0,
    0,  0,  10
  ]
}
```

Center
x-axis
y-axis
z-axis



An *oriented bounding box* fits the bounding volume more tightly to the geometry, particularly for regular, technical structures like CAD models. It is defined by the center position of the box, and three 3D vectors. The vectors define the directions and half-lengths of the x, y, and z-axis.

```
"boundingVolume": {
  "region": [
    -1.319700,
     0.698858,
    -1.319659,
     0.698889,
     0.0,
    20.0
  ]
}
```

West
South
East
North
Min. height
Max. height

A *region* is particularly well suited to geographical information systems, because the sides of the region are defined with latitude and longitude coordinates, for the west, south, east, and north of the region, and a minimum and maximum height.
Latitudes and longitudes are in the WGS 84 datum as defined in EPSG 4979 and are in radians.
The minimum and maximum height are given in meters above or below the WGS 84 ellipsoid.

Details about WGS84 and the EPSG 4979 can be found in Section 13, "Common Definitions."

4. Spatial Data Structures

The tiles in a tileset are organized in a tree data structure. Each tile has an associated bounding volume. This allows modeling different spatial data structures. The following section will present different types of spatial data structures that can all be modeled with 3D Tiles. The runtime engine can use the spatial information generically in order to optimize the rendering performance, independent of the actual characteristics of the spatial data structure.

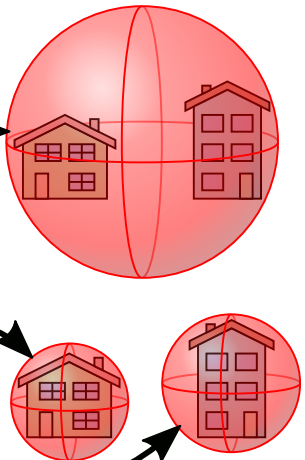
Spatial Coherence

All types of bounding volumes and spatial data structures in 3D Tiles have *spatial coherence*, meaning that the bounding volume of a parent tile always encloses the content of all its child tiles.

Spatial coherence is crucial for conservative visibility tests and intersection tests because it guarantees that when an object does *not* intersect the bounding volume of a tile, then it also does *not* intersect the content of any child tile.

In contrast, when the object *does* intersect a bounding volume, then the bounding volumes of the children are tested for intersections. This can be used to quickly prune large parts of the hierarchy, leaving only a few leaf tiles where the object must be tested for intersections with the actual tile content geometry.

```
"boundingVolume": {
 "sphere": [
  49.3, 23.0, 10.0, 56.5
 ]
}, ...
"children": [
 {
  "boundingVolume": {
   "sphere" : [
    18.1, 14.2, 8.2, 24.0
   ]
  }, ...
 }, {
  "boundingVolume": {
   "sphere" : [
    83.6, 21.8, 9.2, 28.4
   ]
  }, ...
 }
]
```



Types of Spatial Data Structures

There are different approaches for constructing hierarchical spatial data structures with bounding volumes. These approaches generally result in different types of spatial data structures, depending on the exact strategy that is used for the construction.

A simple spatial hierarchy can be constructed by recursively splitting the content of a tile along certain axes, until a stopping criterion is met. When the content is only split along a single axis at each level, then the result is a **k-d tree**. 3D Tiles also supports **multi-way k-d trees**, where each level has multiple splits along one axis. When the content is split along the x, y, and the z-axis at each level, then the result is an **octree**. When the content is split at the center of the content bounding volume, then the result is a **uniform octree**. When the content is split at a different point, then the result is a **non-uniform** octree.

At each level of the hierarchy, the bounding volumes can be created purely spatially from the bounding volume of the parent node. Alternatively, the bounding volumes for each node can be computed from the actual content of the node. This can be particularly useful for sparse datasets because it ensures **tightly-fitting bounding volumes** at each level of the hierarchy.

It is sometimes not possible to divide a tile without splitting a single feature (model) of the content. It is therefore possible to construct **loose octrees** and **loose k-d trees**, where the bounding volumes of children overlap. These trees still maintain spatial coherence, in that the parent bounding volume still encloses the content of all children.

In the most general case, a tileset can be divided into a **non-uniform grid**. Since each node in the spatial hierarchy in 3D Tiles may have an arbitrary number of child nodes, and non-leaf tiles are not required to contain renderable content, the grid cells can still be organized in a hierarchical structure in order to support hierarchical culling.

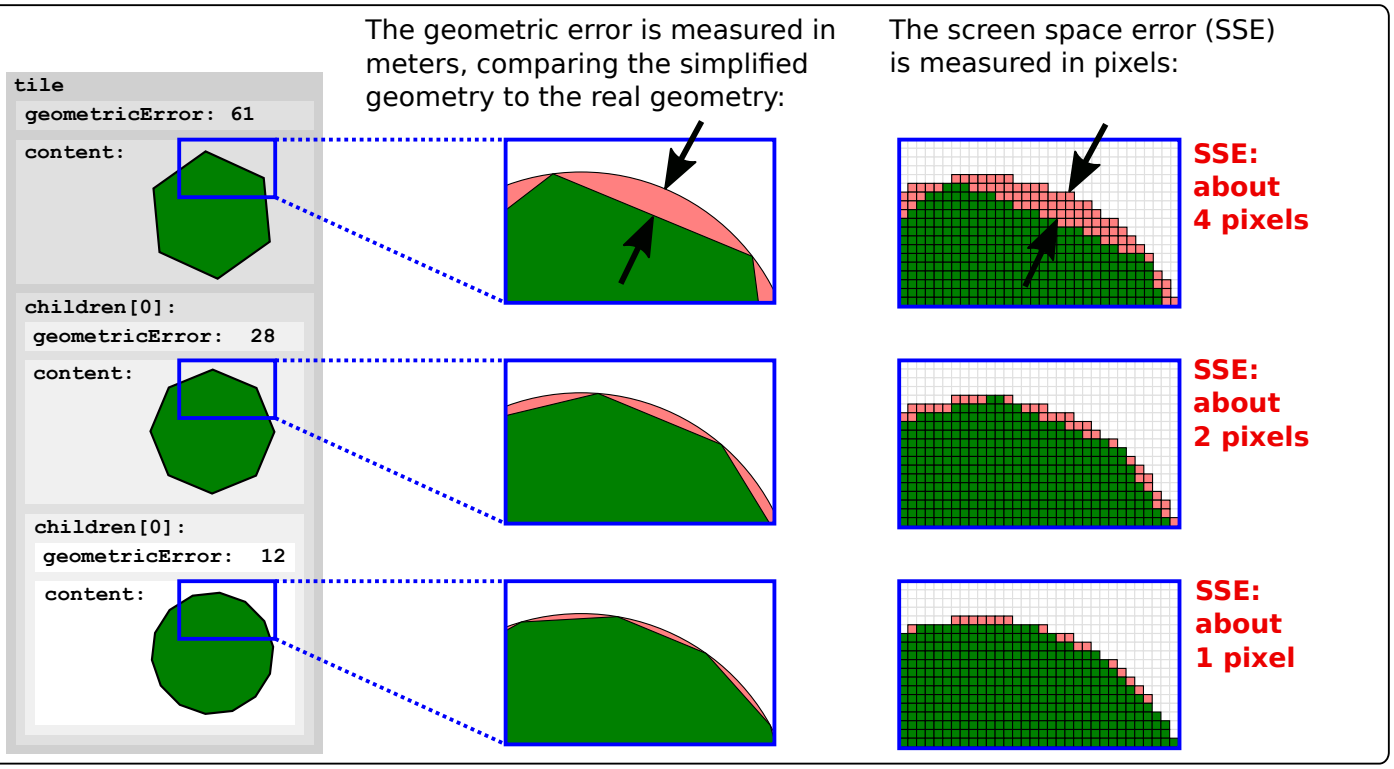
The generic spatial data structures in 3D Tiles allow modeling the results of all these construction methods. Each tile is a node of the hierarchy that may optionally refer to geometry content or store only a bounding volume that encloses the content of all its descendants, and different types of bounding volumes can be stored in a single tileset.

5. Geometric Error

One of the goals of 3D Tiles is to efficiently stream massive datasets to runtime engines, and still allow the runtime to render this content efficiently. The hierarchical structure of tiles in a tileset therefore incorporates the concept of a *Hierarchical Level of Detail* (HLOD): tiles at the top of the hierarchy contain representations of the renderable content with a low level of detail. The child tiles contain content with a higher level of detail. The rendering runtime can dynamically choose the level of detail that offers the best trade-off between performance and rendering quality.

The key for determining which level of detail should be rendered is the *geometric error.* Each tileset and each tile has a **`geometricError`** property that quantifies the error of the simplified geometry compared to the actual geometry.

The runtime translates this geometric error into a *screen-space error* (SSE). The SSE quantifies how much of the geometric error will be visible, in terms of pixels on the screen.



When the SSE exceeds a certain threshold, the runtime will render a higher level of detail. For a tileset, the geometric error is used to determine whether the root tile should be rendered. For a tile, the geometric error is used to determine whether the children of the tile should be rendered.

5.1 Computing the Screen Space Error (SSE)

The runtime can use the information that is provided with the geometric error to find the best trade-off between performance and rendering quality: for any given tileset or tile, the runtime can determine how much the visual quality could improve by rendering the tile, even without actually downloading and rendering the tile content.

To accomplish that, the runtime must compute the screen space error for a given geometric error. The actual screen space error in pixels will depend on the view configuration—that is, the position and orientation of the viewer—and on the resolution at which the final image will be rendered.

The computation of the SSE therefore must take these factors into account. The exact implementation will also depend on the type of view projection that is used. But for a standard perspective projection, one possible way of computing the SSE is as follows:

$$sse = (geometricError \cdot screenHeight) / (tileDistance \cdot 2 \cdot \tan(fovy / 2))$$

where *geometricError* is the geometric error that was stored in the tileset or tile, *screenHeight* is the height of the rendering screen in pixels, *tileDistance* is the distance of the tile from the eye point, and *fovy* is the opening angle of the view frustum in the y-direction.

6. Refinement Strategies

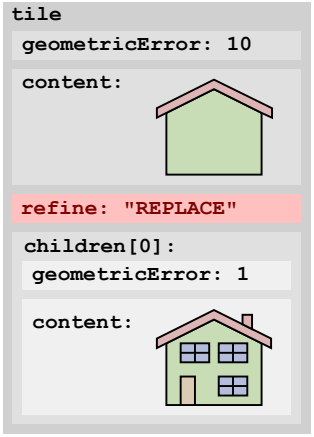
When a tileset is rendered, the runtime traverses the hierarchy of tiles, examines the geometric error of each tile, and computes the associated screen space error. When this screen space error exceeds a certain threshold, then the runtime will recursively consider the child tiles for rendering. The child tiles that contain renderable content have a higher level of detail and a smaller screen space error.

The content of the child tiles can be used to increase the level of detail through one of two refinement strategies. The refinement strategy is determined by the **`refine`** property of a tile, and can either be **`REPLACE`** or **`ADD`**:

Replacement:

The child tiles contain a complete representation of the content with a higher level of detail.

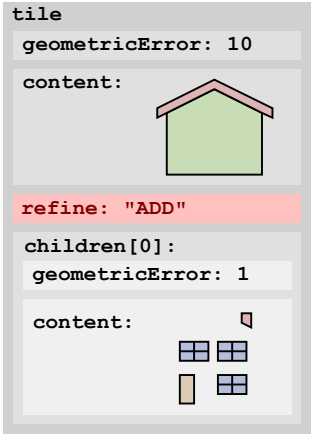
When the child tiles are rendered, this content will replace the content of the parent tile.



Additive:

The child tiles contain additional details for the content of the parent tile.

When the child tiles are rendered, this content will be rendered on top of the content of the parent tile.



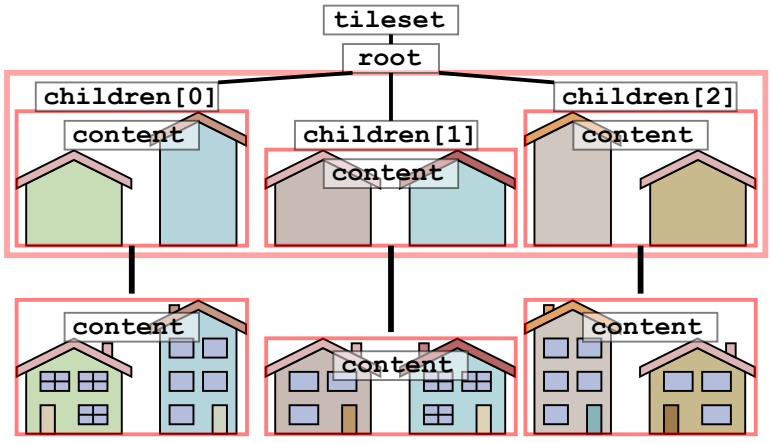
The **`refine`** property can be given for each tile individually but is only required in the root tile of a tileset. When it is not given, the strategy will be inherited from the parent tile.

7. Optimized Rendering with 3D Tiles

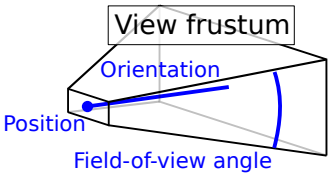
3D Tiles is designed for efficient rendering and streaming of massive heterogeneous datasets. The following is a summary of how the concepts of 3D Tiles can be used by a runtime to interactively render a huge tileset. It shows how the hierarchical structure of tilesets and tiles, the associated bounding volumes, the concept of the geometric error, and the different refinement strategies play together.

The tileset in this example contains a set of buildings. The bounding volume of each element is shown in red.

The tileset contains the root tile, which in turn contains three child tiles. Each child tile contains renderable content. In the example, the content of each tile consists of two buildings, stored as a Batched 3D Model. At this level, the content is stored with a low level of detail, which means that the tiles have a high geometric error. Each child additionally refers to a child where the same content is stored with a higher level of detail, and thus, with a lower geometric error.



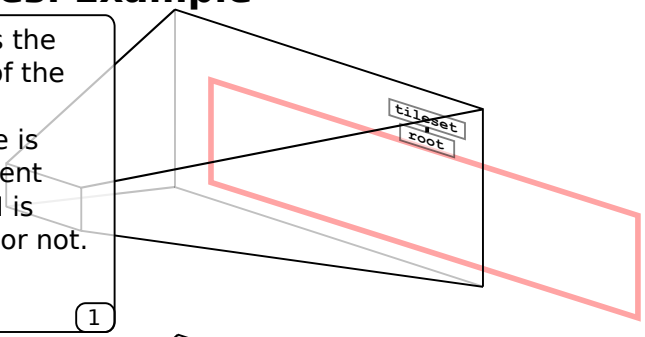
During rendering, the runtime maintains a view frustum, which is defined by the camera position, orientation, and the field-of-view angle. The view frustum can be tested for intersections with the bounding volumes of tilesets and tiles.



7.1. Optimized Rendering with 3D Tiles: Example

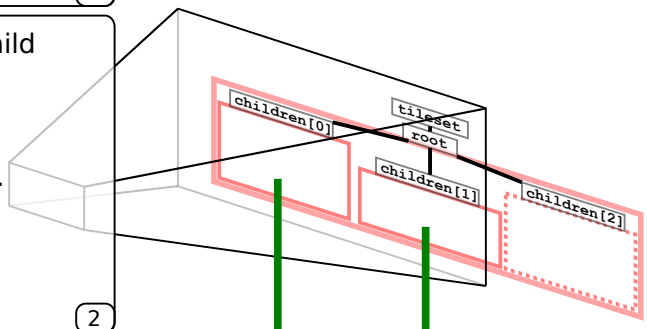
Initially, the runtime loads the main tileset JSON, and tests the view frustum for intersections with the bounding volume of the root tile. In this example, the frustum does intersect the bounding volume of the root tile, which means that the tile is considered for rendering. At this point, no renderable content has been loaded yet: the information from the tileset JSON is sufficient to determine whether anything will be rendered or not.

1



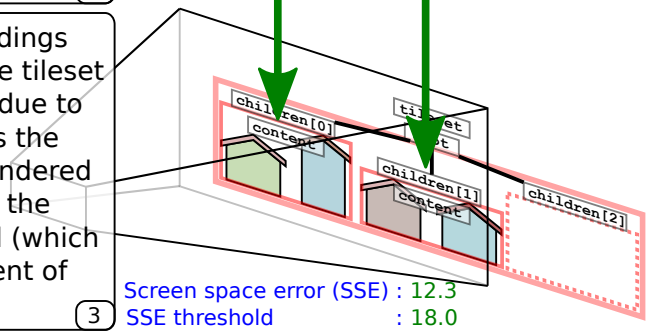
The runtime can then test the bounding volumes of the child tiles for intersections with the view frustum.
In the example, the bounding volumes of two of the three child tiles do intersect the view frustum. This means that the content of these child tiles is considered for rendering. Only the content of these child tiles must be loaded.

2



In the example, the content is a representation of the buildings with a low level of detail. So even though large parts of the tileset are visible, the runtime can efficiently render the content due to its low complexity. During rendering, the runtime observes the effect of the geometric error that is associated with the rendered tiles: it is translated into a screen space error, to estimate the quality of the visual representation. As long as a threshold (which may be 18.0 in the example) is not exceeded, no refinement of the rendered content is necessary.

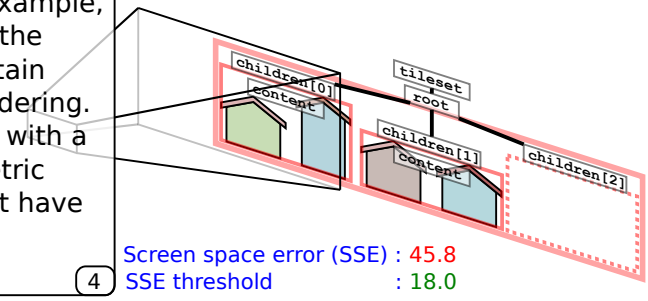
3



Screen space error (SSE) : 12.3
SSE threshold : 18.0

The user can then interact with the rendered tileset. For example, the user can zoom in to a certain building. This will cause the screen space error to increase, and when it exceeds a certain threshold, then the next level of tiles is considered for rendering. These tiles contain a representation of the same content, with a higher level of detail, and therefore, with a smaller geometric error. The runtime only has to load the content of tiles that have bounding volumes that intersect the new view frustum.

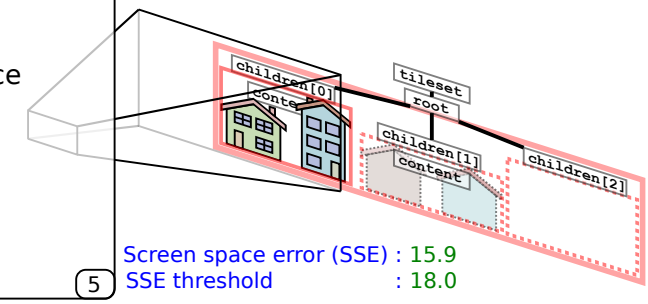
4



Screen space error (SSE) : 45.8
SSE threshold : 18.0

The content with the higher level of detail is loaded and rendered according to the selected refinement strategy. Due to its lower geometric error, it causes the screen space error to fall below the threshold, which implies a higher visual quality. But now that only small parts of the tileset are visible, the content with a high level of detail can still be rendered efficiently.

5



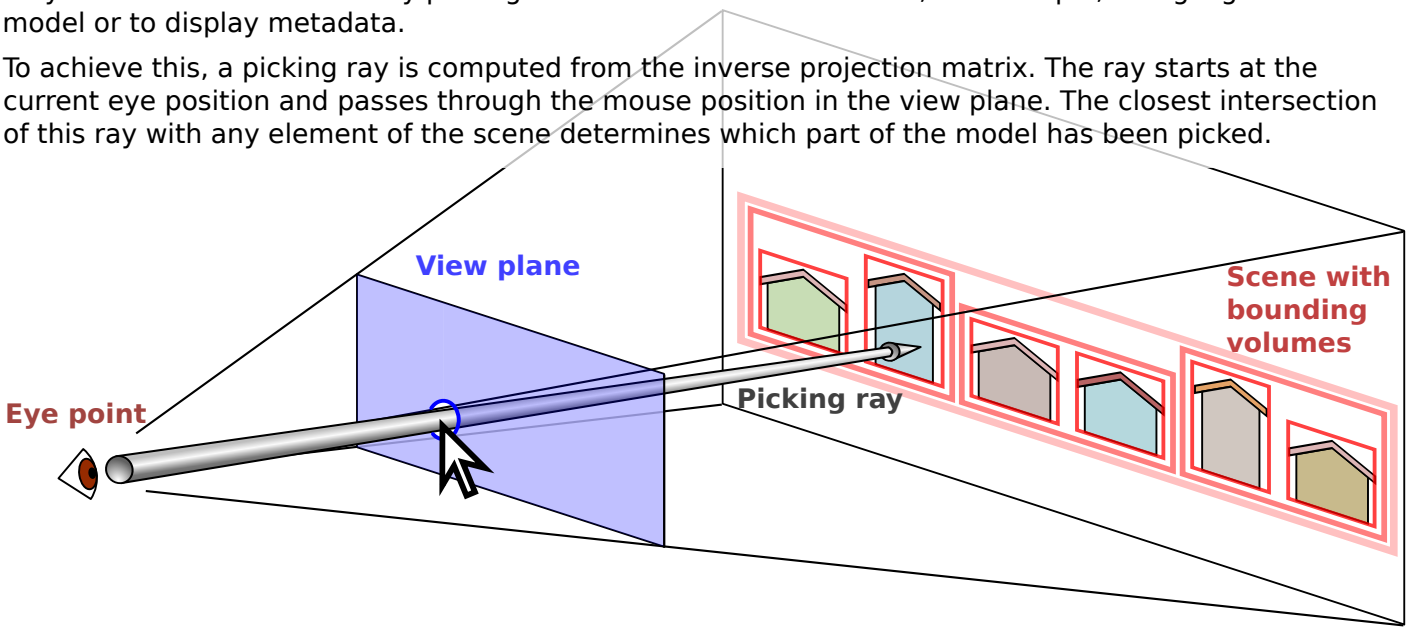
Screen space error (SSE) : 15.9
SSE threshold : 18.0

This example shows how 3D Tiles balances the rendering performance and visual quality at any scale. The runtime can detect whether the view frustum intersects the bounding volume of a tile, and only if this is the case will tile content be downloaded. The runtime can initially display the tile content with the lowest level of detail, and only download and render the content with a higher level of detail on demand, when the screen space error exceeds a threshold—for example, when the user zooms closely to one feature of the dataset.

8. Spatial Queries in 3D Tiles

The hierarchical structure of tiles with bounding volumes in 3D Tiles allows for efficient spatial queries. An example of such a spatial query is *ray casting*: When a tileset is rendered by the runtime, the user may interact with the scene by picking individual models or features, for example, to highlight the model or to display metadata.

To achieve this, a picking ray is computed from the inverse projection matrix. The ray starts at the current eye position and passes through the mouse position in the view plane. The closest intersection of this ray with any element of the scene determines which part of the model has been picked.

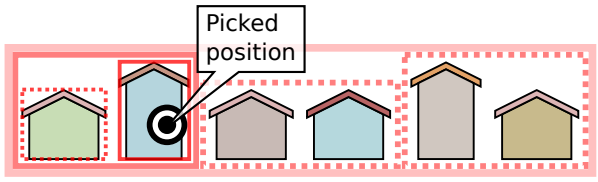


The test for intersections starts at the root tile of the rendered tileset. When the ray intersects the bounding volume of this tile, it is tested for intersections with each of the child tiles. When an intersection with a leaf tile is found, the actual geometry data of the content of this tile can be tested for intersections with the ray, to determine the actual picking position.

During the intersection tests, the spatial coherence of the tree makes it possible to quickly prune parts of the hierarchy that are not intersected: when the bounding volume of a tile is not intersected, then the intersection tests for the children of this tile can be skipped. In the example, the bounding volumes that are shown with dashed lines do not intersect the picking ray, and the traversal may stop there.

Pseudocode for finding the tile that was picked with a ray:

```
function intersect(ray, tile) {
  if (ray.intersects(tile.boundingVolume)) {
    if (tile.isLeaf) {
      return tile;
    }
    for (child in tile.children) {
      intersection = intersect(ray, child);
      if (intersection) return intersection;
    }
    return undefined;
}
```



For certain types of tiles, knowing that the content of a certain tile was hit with a mouse click may not be sufficient: In tile formats like Batched 3D Models (Section 10.2) or batched Point Clouds (Section 10.4), multiple distinct models or parts of models (features) may be combined into a single geometry. In these cases, the vertices of the geometry are extended with a *batch ID*, which identifies the feature. This makes it possible to determine the actual feature that was hit with the mouse click.

9. Tile Formats: Introduction

The renderable content of a tile is referred to by a URI that is part of the tile JSON. This renderable content can be stored in different formats for different model types:

- **Batched 3D Model**: Heterogenous models like textured terrain or 3D buildings
- **Instanced 3D Model**: Multiple instances of the same 3D model
- **Point Clouds**: A massive number of points

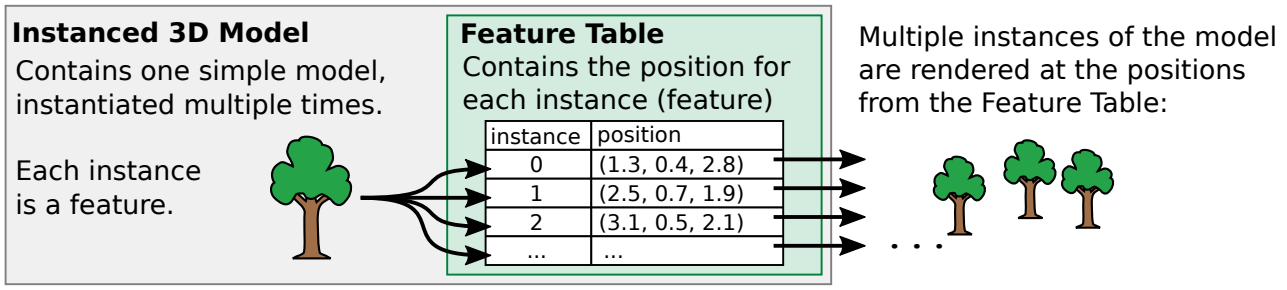
A tileset can contain any combination of tile formats. Additional flexibility is achieved by allowing tiles with different formats to be combined as a *Composite Tile* (Section 10.5).

The renderable content of a tile contains different *features*. For Batched 3D Models, each part of the geometry may be a feature. For example, when several buildings are combined in a Batched 3D Model, then each building may be a feature. For Instanced 3D Models, each instance is a feature. For Point Clouds, there are two options: a feature can either be a single point, or a group of points that represent an identifiable part of the model.

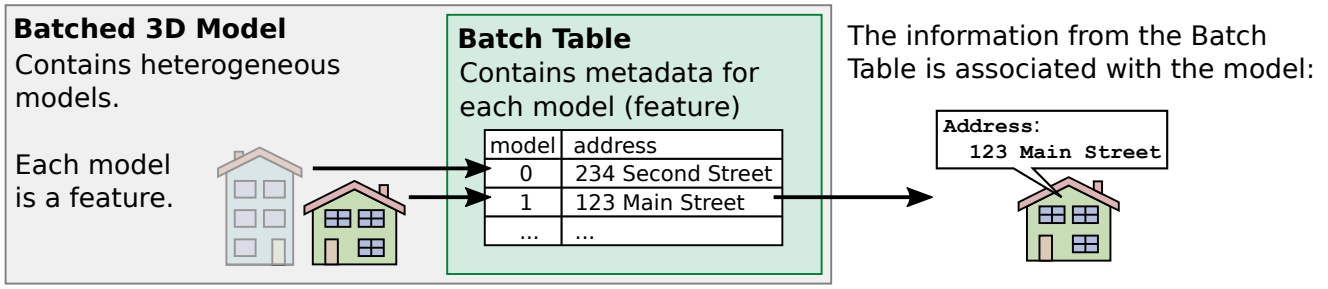
Feature Table and Batch Table

A common element of all tile formats (except for composite tiles) is the *Feature Table* and the *Batch Table*. The exact contents of the Feature Table and the Batch Table depend on the tile format, but their structure and layout are the same for all tile formats.

The Feature Table contains properties that are required for rendering the features. For example, in an Instanced 3D Model, the positions of the instances are stored in the Feature Table:



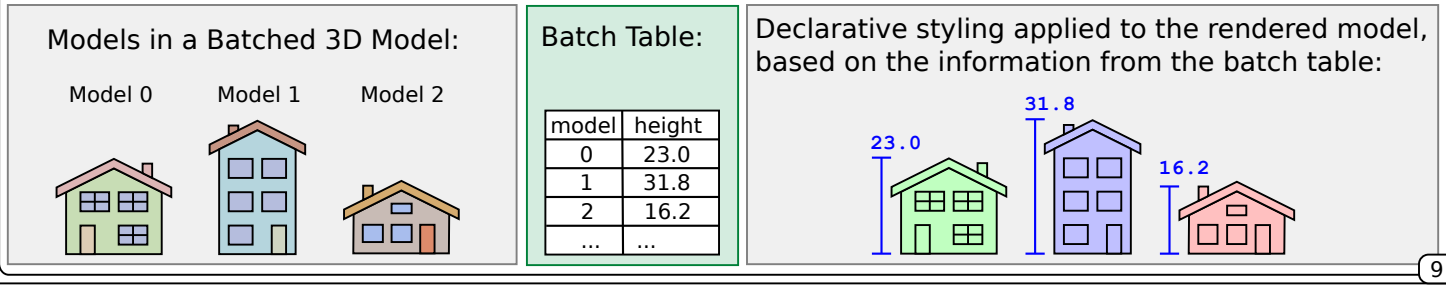
The Batch Table can contain additional, application-specific properties for each feature. For example, in a Batched 3D Model, metadata that is associated with each model is stored in the Batch Table:



Outlook: Declarative Styling

The information that is stored in a Batch Table is not directly necessary for rendering. But the Batch Table usually contains metadata that can be used for declarative styling, as shown in Section 11.

For example, the Batch Table may contain information about the heights of a set of buildings that appear in the tile. This information can be accessed in the 3D Tiles styling language to modify the appearance of the model. For example, the buildings can be rendered with different colors, depending on their height:



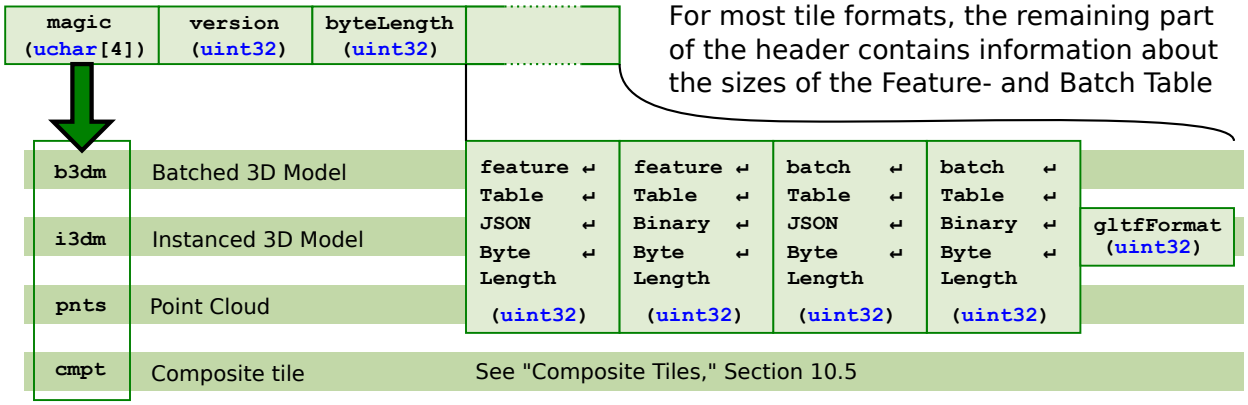
10. Tile Formats

The actual renderable content of a tile is stored as a binary blob. This blob consists of a header with structural information, and a body, which contains the actual payload:

Tile Format Header

The header of each tile format starts with a sequence of four magic bytes. These bytes represent a 4-character string that determines the tile format. They are followed by the version number of the tile format, and the total length of the tile data, including the header, in bytes.

The exact size, contents and structure of the header depends on the tile format and is therefore known after the magic bytes have been read. The exact data layout for the tile formats and their headers are shown in the following sections that explain each individual tile format.

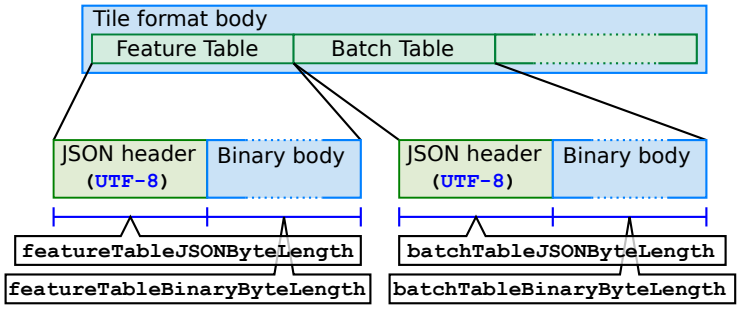


The sizes **`featureTableJSONByteLength`**, **`featureTableBinaryByteLength`**, **`batchTableJSONByteLength`**, and **`batchTableBinaryByteLength`** describe the size (in bytes) of the respective parts of the Feature Table and Batch Table and refer to the tile format body that contains the actual table data.

Tile Format Body

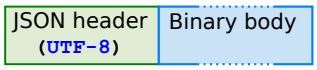
The tile format body contains the actual payload of the tile data. For all tile formats (except for composite tiles) this body may contain a Feature Table and a Batch Table, as well as other binary data that is specific for the tile format.

The Feature Table and Batch Table each consist of a JSON header and a binary body. The length of each element is determined by the information in the tile format header.

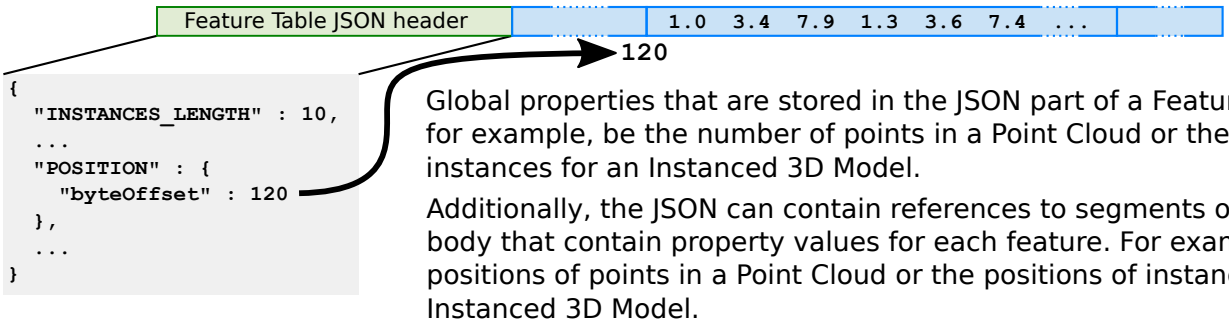


10.1. Tile Formats: Feature Table and Batch Table

The Feature Table and the Batch Table are stored in the tile format body. Both kinds of tables have the same structure: they consist of a header part, which is interpreted as a JSON string, and a binary body:



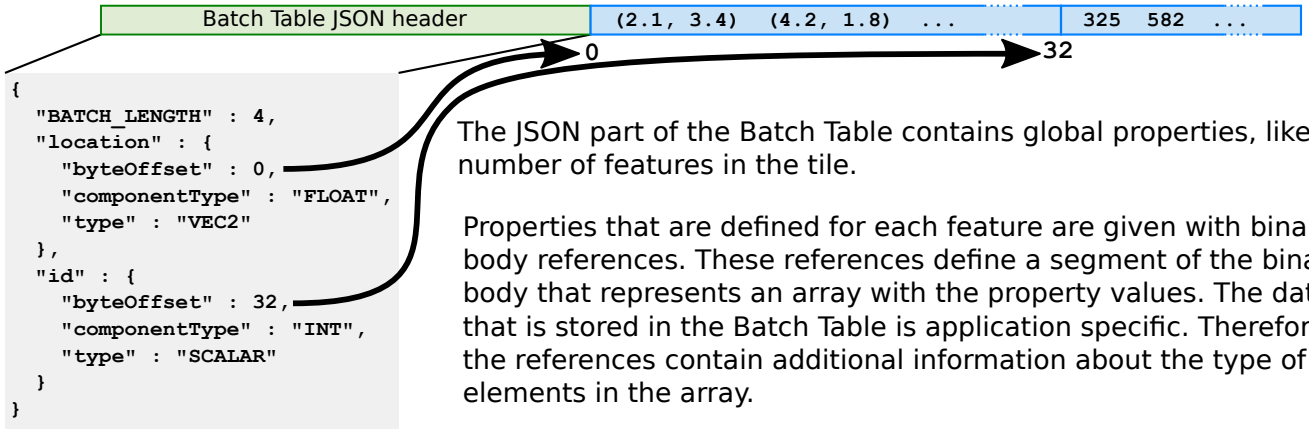
The JSON part may contain global properties that refer to the whole tile. Additionally, the JSON part may contain references to the binary body. The exact set of properties that are supported depend on the tile format, but all properties define segments of the binary body that contain arrays of values for each feature that appears in the tile.



Global properties that are stored in the JSON part of a Feature Table may, for example, be the number of points in a Point Cloud or the number of instances for an Instanced 3D Model.

Additionally, the JSON can contain references to segments of the binary body that contain property values for each feature. For example, the positions of points in a Point Cloud or the positions of instances in an Instanced 3D Model.

The section of the binary body that one property refers to is given by the **`byteOffset`** inside the body. The type of the data depends on the property semantics. For example, the **`POSITION`** property refers to a section of the body that represents an array of 32-bit floating-point values, representing the x, y, and z coordinates of the positions.



The JSON part of the Batch Table contains global properties, like the number of features in the tile.

Properties that are defined for each feature are given with binary body references. These references define a segment of the binary body that represents an array with the property values. The data that is stored in the Batch Table is application specific. Therefore, the references contain additional information about the type of elements in the array.

The location of the array data in the binary body is given by the **`byteOffset`**. The **`type`** says whether the elements of the array are scalars or vectors. The **`componentType`** defines the type of the scalar or vector components.

The following tables contain the number of bytes that each component type consists of, and the number of elements that each type consists of. This can be used to compute the byte size of the property data in the binary body:

componentType	Size in bytes
"BYTE"	1
"UNSIGNED BYTE"	1
"SHORT"	2
"UNSIGNED SHORT"	2
"INT"	4
"UNSIGNED INT"	4
"FLOAT"	4
"DOUBLE"	8

type	Number of components
"SCALAR"	1
"VEC2"	2
"VEC3"	3
"VEC4"	4

10.2. Tile Formats: Batched 3D Models (`b3dm`)

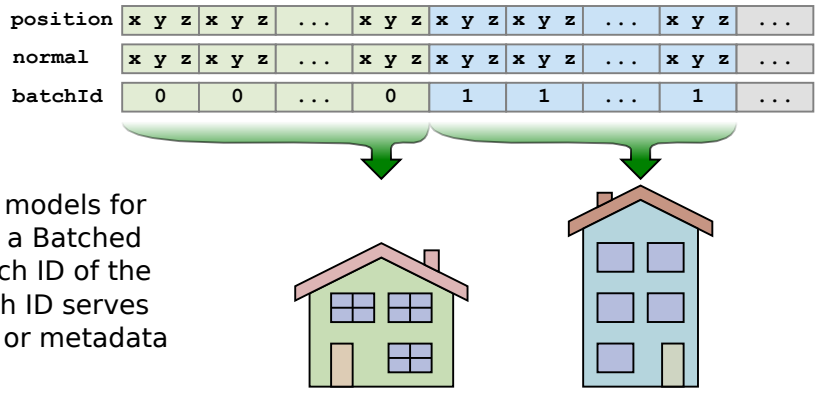
A Batched 3D Model tile contains the rendering data of heterogeneous models. These models may be terrain or 3D buildings, for example, for Building Information Management (BIM) or engineering applications.

The actual rendering data in a Batched 3D Model is stored as *Binary glTF* - a binary form of the GL Transmission Format. This format allows individual models or even complete 3D scenes with textures and animations to be stored in a compact form that can be efficiently transferred over networks and directly rendered by runtime engines. Geometry data in glTF is stored in a buffer that is structured by dividing the buffer into parts that represent different attributes, like the vertex positions or normals. A summary and links to further resources about glTF are given in Section 13, "Common Definitions."

The term "batched" refers to the fact that the geometry data of multiple models, each consisting of vertex positions and optional normals and texture coordinates, can be combined into a single buffer, in order to improve the rendering performance: the data of a single buffer can be copied directly into GPU memory, resulting in fewer copying operations, and it can be structured in a form that minimizes the number of draw calls.

When multiple models are combined into a single buffer, it must still be possible to support styling and interaction for the individual models. This can refer to highlighting a model by rendering it with a different color or determining which model has been picked with a mouse click. In 3D Tiles, this is achieved by extending the buffer with an additional vertex attribute.

The geometry data is extended with the **`batchId`** attribute. It stores the batch ID for each vertex as an integral number. Vertices with the same ID are part of the same model.



The batch ID can then be used to identify the models for interaction or styling: when the user clicks on a Batched 3D Model, the runtime can determine the batch ID of the part of the model that was selected. The batch ID serves as an index for looking up styling information or metadata in the Batch Table.

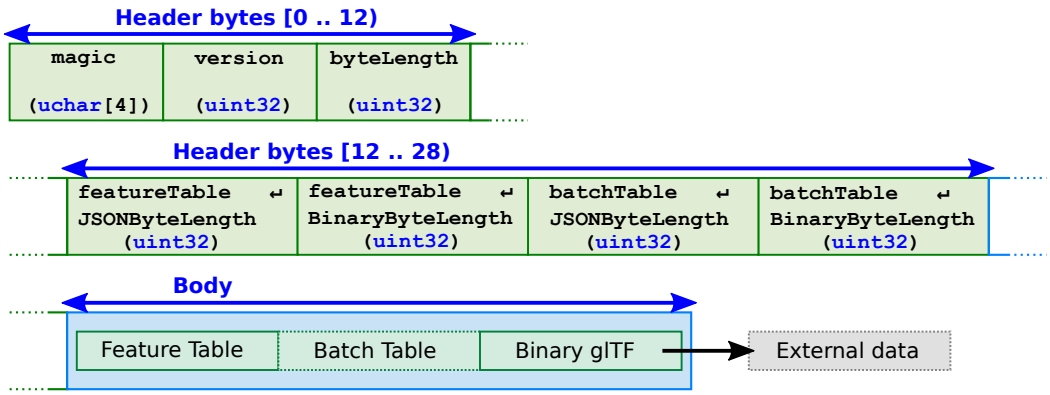
Batched 3D Models: Properties

The Feature Table of Batched 3D Models only contains global properties. The size of the Batch Table is given as the **`BATCH_LENGTH`** property, and the **`RTC_CENTER`** may store the center point in the case that the positions are given relative to the center.

Property	Type	Description
`BATCH_LENGTH`	**`uint32`**	The number of distinguishable models (features) in the batch. If the binary glTF does not have a **`batchId`** attribute, this field must be 0.
`RTC_CENTER`	**`float32[3]`**	The center position when positions are defined relative-to-center

Batched 3D Models: Data Layout

The following diagram shows the layout and data types for the information that is contained in the header and body of Batched 3D Models:

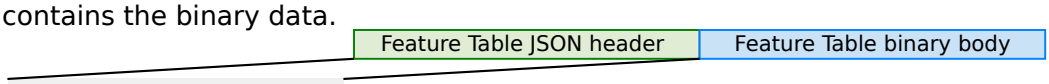


10.3. Tile Formats: Instanced 3D Models (`i3dm`)

In many application scenarios, complex scenes contain the same model multiple times, but with small variations. Examples of such models may be trees, CAD features like bolts, or elements of interior design like furniture.

The 3D Tiles format therefore supports *Instanced 3D Models*, where a single model is rendered multiple times. Each appearance of this model is an instance (or feature), and the instances can be rendered with different transformations—for example, at different positions.

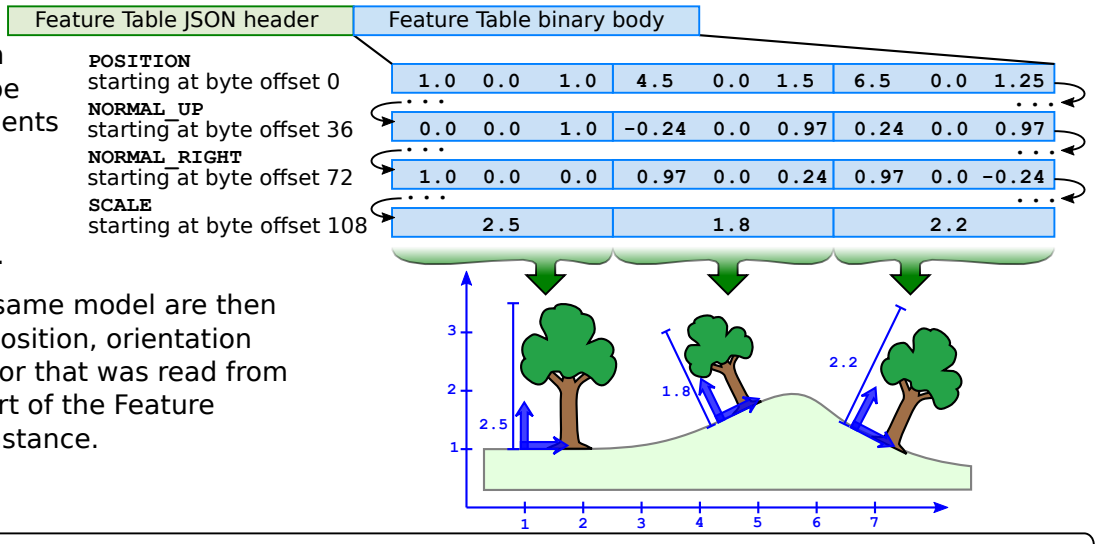
The actual model is stored as a binary glTF (see Section 13, "Common Definitions"). It can be stored directly in the binary body. Alternatively, the body can contain a URI for the binary glTF file. Information about how many instances will be rendered, and the positions and orientations of these instances, are stored in the Feature Table. The first part of the Feature Table contains JSON data, and the second part contains the binary data.




```
{
  "INSTANCES_LENGTH" : 3,
  "POSITION" : {
    "byteOffset" : 0
  },
  "NORMAL_UP" : {
    "byteOffset" : 36
  },
  "NORMAL_RIGHT" : {
    "byteOffset" : 72
  },
  "SCALE" : {
    "byteOffset" : 108
  }
}
```

The JSON part of the Feature Table of an Instanced 3D Model contains the global **`INSTANCES_LENGTH`** property, which determines the number of instances that will be rendered.

It further contains properties for each feature (instance), given as references into the binary body of the Feature Table. The section of the binary body that starts at the respective **`byteOffset`** represents an array. The array length is given by the number of instances. The type of the array is determined by the property: in the example, the properties are **`POSITION`**, **`NORMAL_UP`**, and **`NORMAL_RIGHT`**, each given by three floating-point values for the x, y, and z coordinates, and a **`SCALE`** factor for each instance, given as single floating-point values.

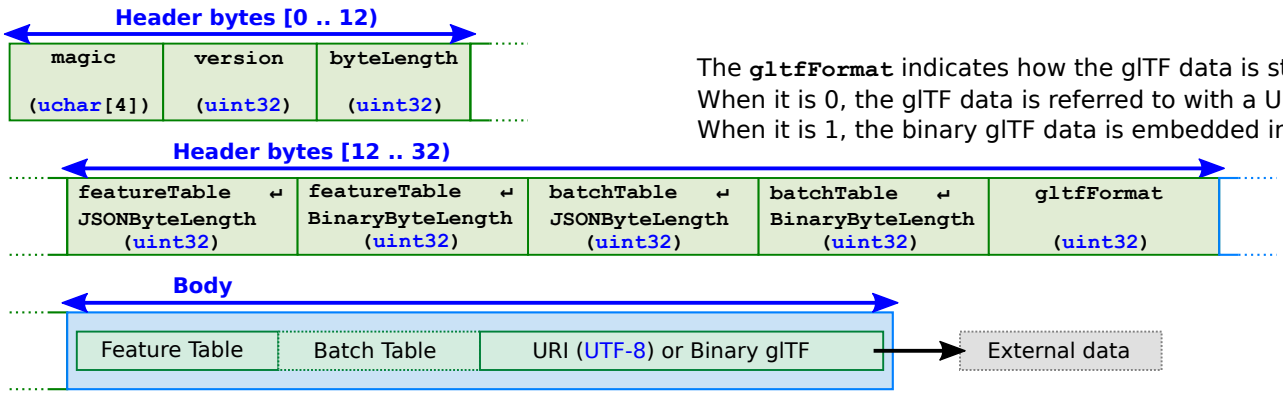


The property information from the JSON part can be used to access the segments of the binary body that represent the arrays containing the property values for each instance.

Multiple instances of the same model are then rendered, each with the position, orientation (normal), and scaling factor that was read from the array in the binary part of the Feature Table for the respective instance.

Instanced 3D Models: Data Layout

The following diagram shows the layout and data types for the information that is contained in the header and body of Instanced 3D Models:



The **`gltfFormat`** indicates how the glTF data is stored:
When it is 0, the glTF data is referred to with a URI.
When it is 1, the binary glTF data is embedded in the body.

Instanced 3D Models: Properties

The Feature Table of an Instanced 3D Model may contain the following properties:

Number of instances: The number of instances that will be rendered, stored in the `INSTANCES_LENGTH` property. This determines the length of the arrays that store the per-instance properties. It also determines the size of the Batch Table in the case that the Batch Table stores metadata for each instance.

Batch ID: An identifier for each instance, stored in the `BATCH_ID` attribute. This batch ID serves as an index that can be used to look up properties for each instance in the Batch Table.

Position: The positions of the instances, given as cartesian x, y, and z coordinates. The positions can be stored directly in a `POSITION` attribute, or using a compressed, quantized representation in the `POSITIONS_QUANTIZED` attribute.

Relative-to-center point: The center point for relative-to-center positions, can be stored in the `RTC_CENTER` attribute. When it is defined, then the positions are given relative to this point.

Quantized volume: The volume that is used for quantization when the positions are given in their compressed form in the `POSITIONS_QUANTIZED` property. The volume is defined by the `QUANTIZED_VOLUME_OFFSET`, which defines the offset of the volume, and the `QUANTIZED_VOLUME_SCALE`, which defines the scale for the quantized volume.

Orientation: The orientation of each instance can be defined with two vectors: one vector defining the up-direction, and one vector defining the right direction. These normals can be stored in the `NORMAL_UP` and `NORMAL_RIGHT` attribute, or using a compressed, oct-encoded representation of the normals, where the normals are stored in the `NORMAL_UP_OCT16P` and `NORMAL_RIGHT_16P` attribute.

Default orientation: When the orientation is not given for the individual instances, then the `EAST_NORTH_UP` can be used to indicate that each instance will default to the east/north/up reference frame's orientation on the WGS84 ellipsoid.

Scale: The scaling factors for the instances. The scaling for each instance can either be a uniform scaling, which is a single value stored in the `SCALE` attribute, or scaling factors along the x, y, and z-axes, given via the `SCALE_NON_UNIFORM` attribute.

Further information about the quantized position and oct-encoded normal representations, about the concept of relative-to-center positions, and the WGS84 ellipsoid, can be found in Section 13, "Common Definitions."

Instanced 3D Models: Properties Summary

The following tables summarize the properties that may be contained in the Feature Table of Instanced 3D Models.

Global Properties

Property	Type	Description
`INSTANCES_LENGTH`	`uint32`	The number of instances
`QUANTIZED_VOLUME_OFFSET`	`float32[3]`	The offset of the quantized volume in x, y, and z-direction
`QUANTIZED_VOLUME_SCALE`	`float32[3]`	The scale of the quantized volume in x, y, and z-direction
`EAST_NORTH_UP`	`boolean`	Whether the instance orientation should default to the east/north/up reference frame on the WGS84 ellipsoid
`RTC_CENTER`	`float32[3]`	The center position when positions are defined relative-to-center

Per-instance Properties

Each of these properties is a reference to a section of the binary body of the Feature Table. This section contains the data of an array with the actual property values. The data type of the array elements is given by the type of the property. The length of the array is determined by the number of instances.

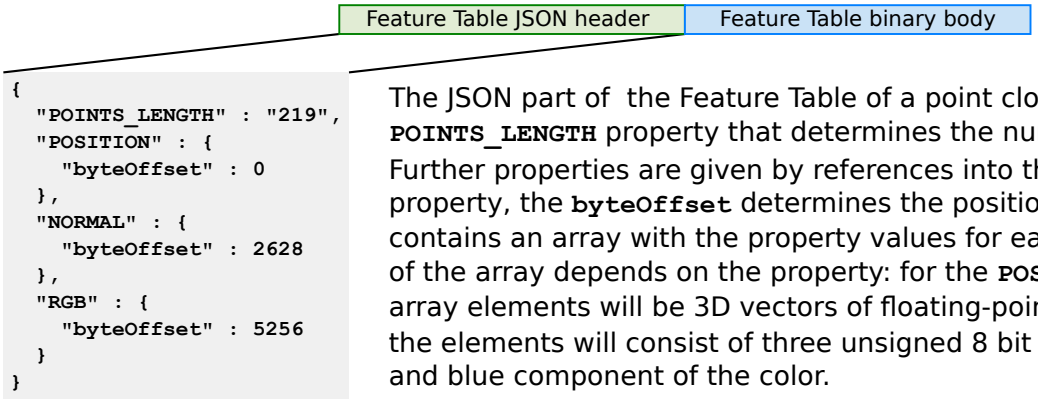
Property	Type	Description
`POSITION` `POSITION_QUANTIZED`	`float32[3]` `uint16[3]`	The x,y, and z coordinates for the position of the instance
`NORMAL_UP` `NORMAL_UP_OCT32P`	`float32[3]` `uint16[2]`	A unit vector defining the up-direction of the instance
`NORMAL_RIGHT` `NORMAL_RIGHT_OCT32P`	`float32[3]` `uint16[2]`	A unit vector defining the right-direction of the instance
`SCALE`	`float32`	A scaling factor for all axes of the instance
`SCALE_NON_UNIFORM`	`float32[3]`	The scaling factors for the x, y, and z-axis of the instance
`BATCH_ID`	`uint8/16/32`	The batch ID, to look up metadata for the instance in the batch table

10.4. Tile Formats: Point Clouds (`pnts`)

A common way of obtaining 3D data from existing structures like buildings or the environment is via photogrammetry or LIDAR scanning. The result of this acquisition process is a point cloud, where each point is defined by its position, and additional properties that define its appearance.

The *Point Clouds* format is a format in 3D Tiles that allows streaming massive point clouds for 3D visualization.

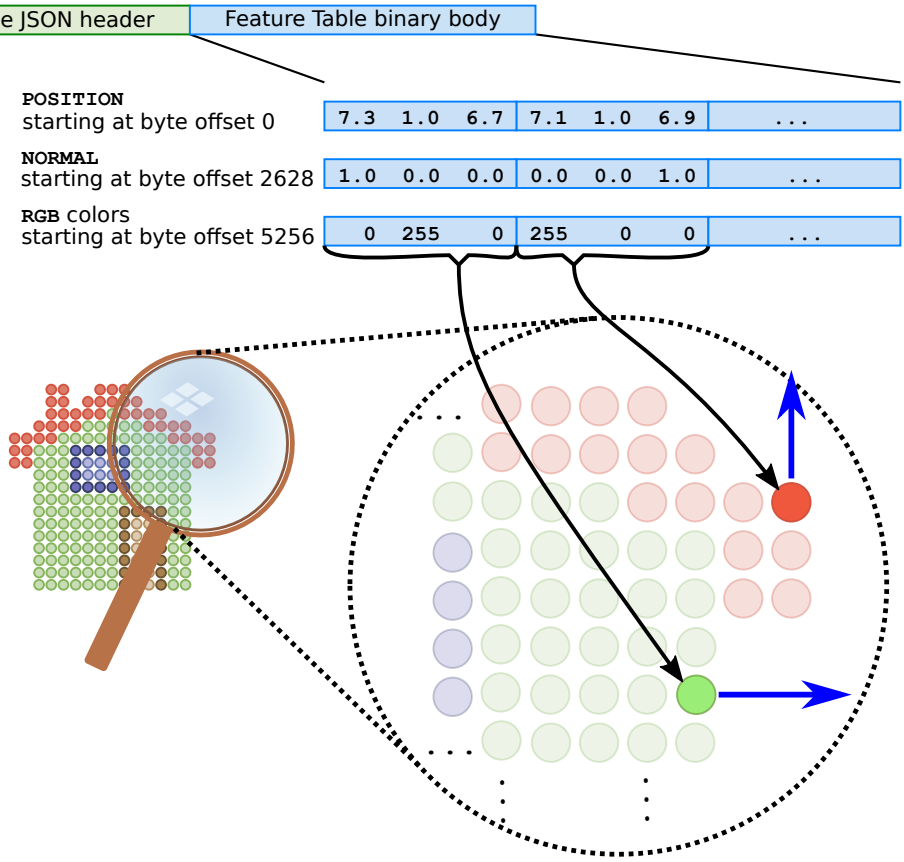
The information about the positions and other visual properties of the points is stored in the Feature Table, which consists of a JSON header and a binary body.



The JSON part of the Feature Table of a point cloud contains the global **`POINTS_LENGTH`** property that determines the number of points.

Further properties are given by references into the binary body: for each property, the **`byteOffset`** determines the position in the binary body that contains an array with the property values for each of the points. The type of the array depends on the property: for the **`POSITION`** and **`NORMAL`**, the array elements will be 3D vectors of floating-point values. For the **`RGB`** color, the elements will consist of three unsigned 8 bit values for the red, green, and blue component of the color.

Each section of the binary part of the Feature Table represents an array of property values. The type of the array elements depends on the property. The lengths of the arrays are given by the number of points.



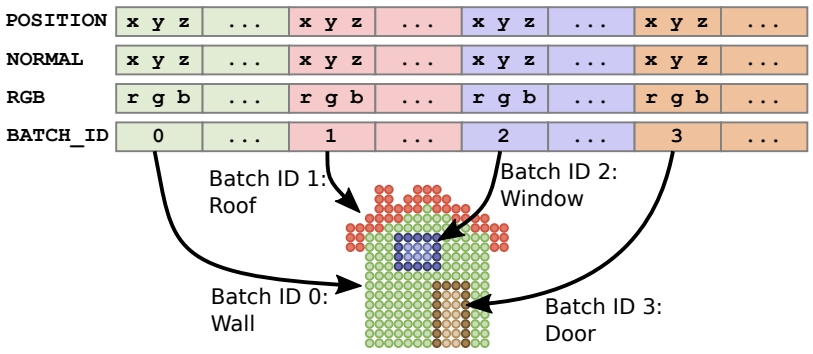
Batched Point Clouds

In a Point Cloud, it is possible to define groups of points that represent distinct features. For example, there may be groups of points that represent a door, a window, or the roof of a house.

In 3D Tiles, these groups are defined by assigning a batch ID to the points, similar to the concept that is used for Batched 3D Models.

For batched Point Clouds, the JSON part of the Feature Table contains a **`BATCH_LENGTH`** property that defines the number of groups of points, and a **`BATCH_ID`** property that refers to a section of the binary body that contains the array with batch IDs for the points, stored as 8, 16, or 32-bit integer values. The batch ID can then be used as an index to look up metadata for that group of points in the Batch Table.

```
{
  "POINTS_LENGTH" : "219",
  "BATCH_LENGTH" : 4,
  "POSITION" : {
    "byteOffset" : 0
  },
  ...
  "BATCH_ID" : {
    "byteOffset": 5913,
    "componentType" :
      "UNSIGNED_BYTE"
  }
}
```



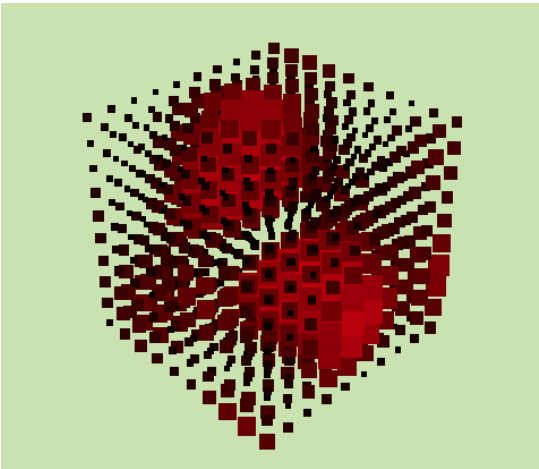
Outlook: Declarative Styling of Point Clouds

Declarative styling, which is shown in Section 11, can be applied to Point Clouds to facilitate different kinds of information visualization.

For example, a Point Cloud may consist of a regular grid of points that fills a certain volume. The Batch Table can then contain measurements that have been taken at the respective positions.

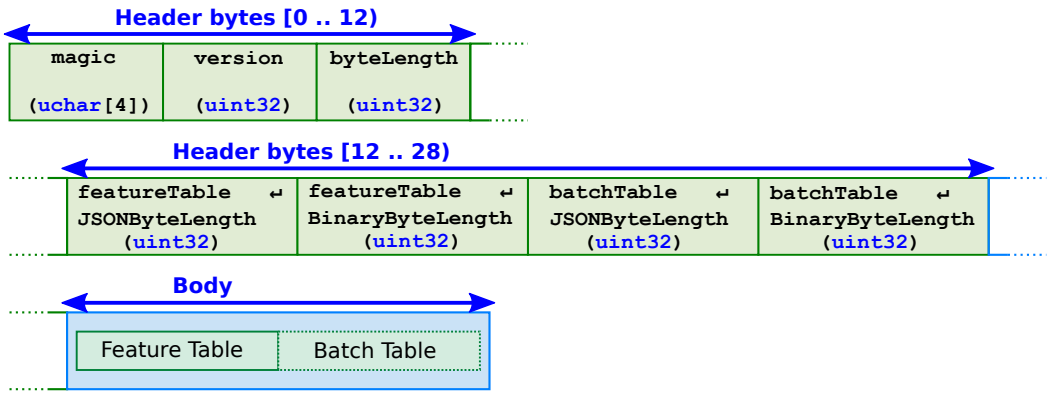
In the example shown here, each point has an associated **`temperature`** value stored in the Batch Table. This domain-specific information from the Batch Table can then be mapped to visual attributes of the points, using the 3D Tiles styling language. Here, the **`temperature`** property affects the color and the size of the rendered points:

```
{
  "color" : "color('red') * ${temperature}",
  "pointSize" : "5 + ${temperature} * 30"
}
```



Point Clouds: Data Layout

The following diagram shows the layout and data types for the information that is contained in the header and body of Point Clouds:



Point Clouds: Properties

The Feature Table of a Point Cloud may contain the following properties:

Number of points: The number of points in the Point Cloud is given by the **POINTS_LENGTH** property. This determines the length of the arrays that store the per-point properties.

Number of batches: The number of batches (groups of points) in the Point Cloud, stored in the **BATCH_LENGTH** property. It is therefore the number of unique values in the **BATCH_ID** attribute, and the number of entries in the Batch Table.

Batch ID: An identifier for the batch (group) that the point belongs to is stored in the **BATCH_ID** attribute. All points with the same batch ID belong to one batch. The batch ID serves as an index to look up properties for this group of points in the Batch Table.

Position: The positions of the points, given as cartesian x, y, and z coordinates. The positions can be stored directly in a **POSITION** attribute, or in a compressed, quantized representation in the **POSITIONS_QUANTIZED** attribute.

Relative-to-center point: The center point for relative-to-center positions can be stored in the **RTC_CENTER** attribute. When it is defined, then the positions are given relative to this point.

Quantized volume: The volume that is used for quantization when the positions are given in their compressed form in the **POSITIONS_QUANTIZED** property. The volume is defined by the **QUANTIZED_VOLUME_OFFSET**, which defines the offset of the volume, and the **QUANTIZED_VOLUME_SCALE**, which is the scale for the quantized volume.

Normal: Unit vectors defining the normals of the points. These can be stored in the **NORMAL** attribute, or using a compressed, oct-encoded representation of the normal, using the **NORMALS_OCT16P** attribute.

Color: The color for each point can be given as RGB or RGBA colors that are stored in the **RGB** or **RGBA** properties. Alternatively, a compressed representation of the colors can be stored, in the **RGB565** property.

Constant color: A color for all the points in the tile can be given in the **CONSTANT_RGBA** property. This color will be used when no colors for the individual points are defined.

Further information about the quantized position and oct-encoded normal representations, and about the concept of relative-to-center positions, can be found in Section 13, "Common Definitions."

Point Clouds: Properties Summary

The following tables summarize the properties that may be contained in the Feature Table of Point Clouds.

Global Properties

Property	Type	Description
POINTS_LENGTH	**uint32**	The number of points
QUANTIZED_VOLUME_OFFSET	**float32[3]**	The offset of the quantized volume in x, y, and z-direction
QUANTIZED_VOLUME_SCALE	**float32[3]**	The scale of the quantized volume in x, y, and z-direction
CONSTANT_RBGA	**uint8[4]**	The RGBA color components for all points in the tile
BATCH_LENGTH	**uint32**	The number of batches for the points
RTC_CENTER	**float32[3]**	The center position when positions are defined relative-to-center

Per-point Properties

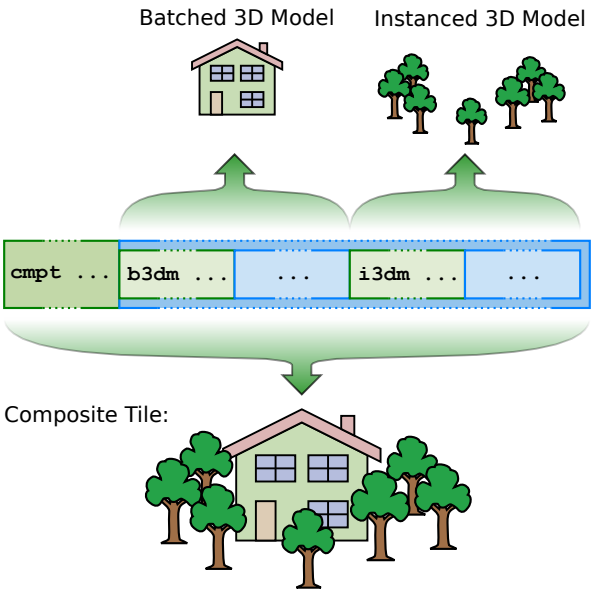
Each of these properties is a reference to a section of the binary body of the Feature Table. This section contains the data of an array with the actual property values. The data type of the array elements is given by the type of the property. The length of the array is determined by the number of points.

Property	Type	Description
POSITION **POSITION_QUANTIZED**	**float32[3]** **uint16[3]**	The x,y, and z coordinates for the position of the point
RGBA	**uint8[4]**	The RGBA color components of the point
RGB	**uint8[3]**	The RGB color components of the point
RGB565	**uint16**	A compressed, 16bit representation of RGB colors, with 5 bits for red, 6 bits for green, and 5 bits for blue
NORMAL **NORMAL_OCT16P**	**float32[3]** **uint8[2]**	A unit vector defining the normal of the point
BATCH_ID	**uint8/16/32**	The batch ID, to look up metadata for the points the batch table

10.5. Tile Formats: Composite Tiles (`cmpt`)

3D Tiles supports streaming heterogenous datasets. Multiple tile formats can be combined in one tileset. It is also possible to combine multiple tiles of different formats in a *Composite Tile* for additional flexibility.

One example for the use of Composite tiles can be found in geospatial applications: A set of buildings could be stored in a Batched 3D Model, and a set of trees could be stored as an Instanced 3D Model. When these elements appear at the same geographic location, it is useful to combine these models in a single Composite tile: The renderable content for a given geolocation can then be obtained as a single tile, with a single request.



The tiles that are combined in a single Composite tile are referred to as *inner tiles*. Composite tiles can also be nested, meaning that each inner tile can again be a Composite tile.

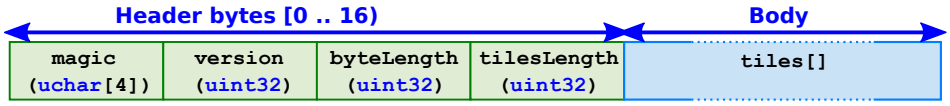
As for other tile formats, the header of a Composite tile starts with the magic bytes that indicate the tile format (**`cmpt`**), followed by a version number, and the total length of the tile data, including the header, in bytes.

The common part of the header is followed by an integer that indicates the number of tiles that have been combined in the Composite tile.

The body of a Composite tile just consists of an array of inner tiles. Each inner tile is a binary blob representing a tile: It starts with the common header, indicating the format, version, and length of the inner tile data, followed by the body for the respective tile format.

Composite Tiles: Data Layout

The following diagram shows the layout and data types for the information that is contained in the header and body of Composite tiles:



11. Extensions

3D Tiles offers a mechansim to extend the base specification with new features: each JSON object in 3D Tiles may contain an **`extensions`** dictionary. The keys of this dictionary are the names of the extensions, and the values may be extension specific objects. Vendors can propose extensions and provide a specification for the structure and semantics of the extension objects that are contained in the dictionary.

```
{
  ...
  "extensions": {
    "VENDOR_collision_volume": {
      "sphere": [ 5.0, 3.0, 7.0, 10.0 ]
    }
  }
}
```

This example shows the JSON that may be added to a tile JSON in the context of a hypothetical vendor extension that defines a collision volume for the tile. The name of the extension is **`VENDOR_collision_volume`**, and it defines a collision sphere via its center and radius, similar to the bounding volumes that are already supported in 3D Tiles.

The names of extensions that are used in a tileset or one of its descendants must be listed in the top-level **`extensionsUsed`** dictionary of the tileset. When a certain extension is required in order to properly load and render the tileset, it must also be listed in the **`extensionsRequired`** dictionary. Implementations can inspect these dictionaries when the tileset is loaded, and check whether they support the extensions that are used or required by the tileset.

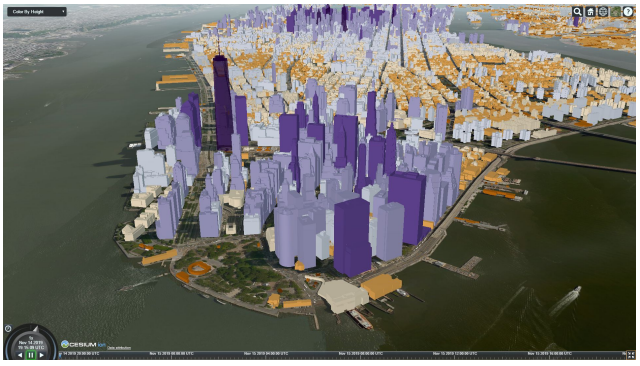
12. Declarative Styling

Depending on the tile format, the renderable content of a tile may contain different features. For Batched 3D Models, each model that is identified with a **batchId** is a feature. For Instanced 3D Models, each instance is a feature. For Point Clouds, there are two options: when the points are batched together, each group of points that has the same **batchId** is a feature. Otherwise, each point is a feature.

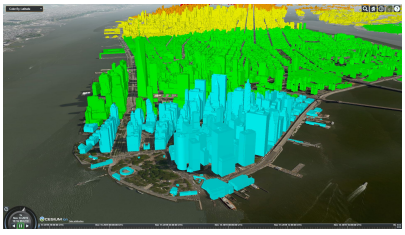
3D Tiles allows modifying the appearance of features at runtime, using *declarative styling*: a style is defined with JSON that contains a set of expressions. The values of these expressions determine the visibility or color of the features. The runtime engine can evaluate these expressions and apply styling to the features based on user interaction and the feature properties that are stored in the Batch Table.

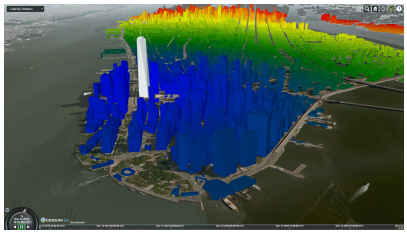
For example, in a Batched 3D Model where each building is a feature, the color of the buildings may be modified at runtime, based on different criteria. The following is an example of a style JSON that causes buildings to be rendered with different colors, depending on their height:

```
{
  "color": {
    "conditions": [
      ["${height} >= 300", "rgba(45, 0, 75, 0.5)"],
      ["${height} >= 200", "rgb(102, 71, 151)"],
      ["${height} >= 100", "rgb(170, 162, 204)"],
      ["${height} >= 50", "rgb(224, 226, 238)"],
      ["${height} >= 25", "rgb(252, 230, 200)"],
      ["${height} >= 10", "rgb(248, 176, 87)"],
      ["${height} >= 5", "rgb(198, 106, 11)"],
      ["true", "rgb(127, 59, 8)"]
    ]
}
```



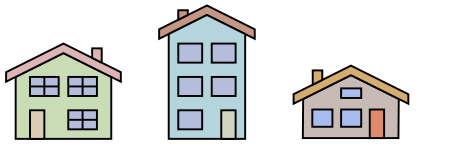
Other examples of declarative styling may apply a color to the buildings based on other properties, like their latitude (left) or their distance to a certain landmark (right) :





Styling a Batched 3D Model

In this example, a Batched 3D Model contains multiple buildings. Each building has its own **batchId**.

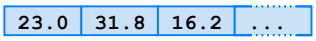


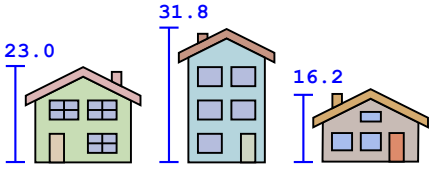
The model also has a Batch Table. The JSON header of the Batch Table contains a property **"height"**. It points to a segment of the binary body of the Batch Table. The segment represents an array that contains the heights of the buildings. The **batchId** is used to look up the height of a building in this array.

Batch table JSON:

```
"height" : {
  "byteOffset" : 0,
  "componentType" : "FLOAT",
  "type" : "SCALAR"
}
```

Batch table binary:



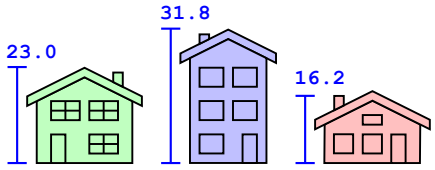


When the model is rendered, a style JSON is applied to the model. The style determines the color of each feature (building). In this example, the color depends on the value of the **"height"** property:

Style JSON:

```
{
 "color" : {
  "conditions" : [
    ["(${height} < 20)", "color('#FFC0C0')"],
    ["(${height} < 30)", "color('#C0FFC0')"],
    ["(${height} < 40)", "color('#C0C0FF')"]
  ]
 }
}
```

The model is rendered in red, green, or blue when the height is lower than 20, 30, or 40, respectively:



Style JSON

The style description is passed to the runtime engine as JSON. It contains properties that determine the appearance of the features in the tile:

```
{
  "show" : "true",
  "color" : "color('red')",
  "pointSize" : "3.0"
}
```

For Batched 3D Models and Instanced 3D Models, the style consists of a **show** property that determines the visibility of the feature, and a **color** property that determines the color.
For Point Clouds, the style may also contain a **pointSize** property that determines the size of the points when they are rendered.

The values for the properties of a style are written in an expression language that is a subset of JavaScript. The types of this language also include vector types that are useful for geometry computations and for representing colors. These vector types are derived from GLSL, and the expressions that can be represented with these types can therefore be implemented easily and efficiently in shaders.

Expressions

The styling language supports most of the unary and binary operators that are supported by JavaScript, as well as the ternary conditional operator and common built-in mathematical functions like **abs**, **max**, or **cos**.
The types of the language include the vector types **vec2**, **vec3**, and **vec4**. The **vec4** type is used to represent RGBA colors.

Vector components of the **vec2**, **vec3**, and **vec4** types can be accessed using the dot notation. For example, the elements of a 4D vector **v** can be interpreted as x, y, z, and w-coordinates and accessed as **v.x**, **v.y**, **v.z**, and **v.w**. Equivalently, they can be interpreted as RGBA color components, and accessed as **v.r**, **v.g**, **v.b**, and **v.a**.
Alternatively, the components can be accessed using the array notation, where the elements of a 4D vector **v** can be accessed as **v[0]**, **v[1]**, **v[2]**, and **v[3]**.
Colors are represented with the **vec4** type, containing the RGBA color components. There are different convenience functions for creating colors from different arguments. These functions resemble the ways colors can be defined in CSS. For example:

- `color('red')` From a keyword
- `color('blue', 0.5)` From a keyword, with an opacity (alpha) of 0.5
- `color('#00FFFF')` From a hex RGB string
- `rgb(100, 255, 90)` From red, green and blue components in [0, 255]
- `hsl(1.0, 0.6, 0.7)` From hue, saturation and lightness, each in [0, 1]

Several built-in functions are defined that can be applied to numbers as well as certain vector types: the **clamp** function can be used to constrain numbers or vector components to a certain range. The **mix** function allows linearly interpolating between numbers or vectors. The **length**, **distance**, **normalize**, and **dot** functions may be used for geometric computations on vectors. The **cross** function allows computing the cross product of 3D vectors.
Expressions can be parsed into an abstract syntax tree (AST) and be evaluated. The type of an evaluated expression must match the type of the style property.

Variables

The expressions that are used in a style may contain variables. These variables refer to properties of the features in a tile. The actual values of these properties are stored in the Batch Table.

Variables are written using the ECMAScript 2015 template literal syntax: the literal **${property}** refers to a property of a feature, where **property** is the case-sensitive property name.

The following is an example of a style that assigns a color to a feature, based on the height of the feature that is stored in the Batch Table: features with a height greater than 50 will be colored red, and all others will be white:

```
{
  "color" : "(${height} > 50) ? color('red') : color('white')"
}
```

Conditions

The style properties may also be written using conditions. A condition consists of two expressions: an expression that evaluates to a boolean value, and an expression that is evaluated when the boolean value is **`true`**. Multiple conditions can be combined as an array.

```
{
  "color" : {
    "conditions" : [
      [ "${temperature} > 100)", "color('red')" ],
      [ "${temperature} > 50)", "color('yellow')" ],
      [ "true", "color('green')" ]
    ]
  }
}
```

An example of the use of conditions: if the temperature that is associated with a feature is greater than 100, then red will be returned. If it is greater than 50, then yellow will be returned. Otherwise, green will be returned. The conditions are evaluated in order. If no condition is met, then the style property will be **`undefined`**.

Defining Variables

In addition to the variables that refer to properties in the Batch Table, a style may also define its own variables. These definitions are given in the **`defines`** property of a style. Each definition consists of the name of the new variable, which is mapped to an expression. The expression may not refer to other defines, but it may refer to existing variables from the Batch Table.

```
{
  "defines" : {
    "distanceToPoint" :
      "distance(vec2(${x}, ${y}), vec2(1, 2))"
  },
  "color" : {
    "conditions" : [
      ["${distanceToPoint} > 1.0", "color('red')"],
      ["true", "color('0x0000FF', ${distanceToPoint}"]
    ]
  }
}
```

The newly defined **`distanceToPoint`** variable represents the distance of a feature to the point (1,2), based on variables **`${x}`** and **`${y}`** that are stored in the Batch Table.
The new variable is used to determine the **`color`** for the style: When the distance exceeds a threshold, the feature is rendered in red. Otherwise, the distance is used to control the opacity of the color.

Regular Expressions

Styles support regular expressions, so that it is possible to formulate conditions for rendering based on property values from the Feature Table that are stored as strings.

```
{
  "show" : "regExp('Building\s\d').test(${name})"
}
```

A style that will only show features that have a **`${name}`** property that matches the given pattern: the string **`"Building"`**, followed by a space and an integer.

Styling Point Clouds

When the declarative styling is applied to Point Clouds, the style may also refer to semantics that are stored in the Feature Table, like position, color, and normal.
The positions of the points may be referred to as **`${POSITION}`**. When the positions are stored in the quantized form, this refers to the positions after the quantization scaling is applied, but before the quantization offset is added. In contrast to that, **`${POSITION_ABSOLUTE}`** refers to the positions after the quantization scaling and offset have been applied. The colors may be accessed using the **`${COLOR}`** keyword. The normals of the points may be accessed as **`${NORMAL}`**. If the normals are stored in the oct-encoded representation, then this refers to the decoded normal.

Further information about the quantized position and oct-encoded normal representations can be found in Section 13, "Common Definitions."

13. Common Definitions

13.1. glTF - The GL Transmission Format

Batched 3D Models and Instanced 3D Models in 3D Tiles may embed models that are stored as Binary glTF. This is the binary form of the GL Transmission Format, which is an open specification for the efficient transmission of 3D content, maintained by the Khronos Group.

https://github.com/KhronosGroup/glTF

13.2. World Geodetic System Ellipsoid (WGS84)

The "region" bounding volume type is intended for geospatial applications. The extents of these bounding volumes are given as latitudes and longitudes, measured in radians, in the WGS 84 datum as defined in EPSG 4979.

https://spatialreference.org/ref/epsg/4979/

The minimum and maximum height of the contents of a region bounding volume are given in meters above or below the WGS 84 ellipsoid.

https://earth-info.nga.mil/GandG/publications/tr8350.2/wgs84fin.pdf

13.3. Relative-To-Center Positions (`RTC_CENTER`)

(The information presented here is based on http://help.agi.com/AGIComponents/html/BlogPrecisionsPrecisions.htm)

3D Tiles was created to make it possible to visualize huge amounts of 3D content, especially geospatial data, in a high-quality, interactive way. This means that it must be possible to render 3D objects with high accuracy, even when they have a large distance from the center of the virtual world.

In common graphics APIs like OpenGL, Direct3D, or Vulkan, the vertex positions of 3D objects are usually stored as single-precision (32-bit) floating-point values. Due to this limited precision, objects that have a large distance to the origin of the rendering coordinate system cannot be represented accurately: different vertex coordinates will have the same, internal representation as a single-precision value:

Theoretical value	Actual 32-bit single-precision value
131072.01	131072.0156250
131072.02	131072.0156250
131072.03	131072.0312500

Despite the different values of 131072.01 and 131072.02, the representations of these values as 32-bit single-precision values are equal.

This lack of accuracy can result in rendering artifacts—most noticeably, in visual jitter when zooming closely to one of the rendered objects. To alleviate this problem, 3D Tiles supports a technique for compensating the error that results from large coordinate values. This technique is called *Relative To Center* (RTC) rendering.

The vertex positions for Batched 3D Models are stored as one attribute in the glTF representation of the model. For Instanced 3D Models and Point Clouds, the Feature Table contains a position property for the instances and points. The coordinate values of these positions are usually stored as single precision, 32-bit floating-point values. To support RTC rendering, the respective tile formats allow specifying an **`RTC_CENTER`** property in their Feature Table. This property defines the center of an application-specific coordinate system, and if it is present, all positions are assumed to be given relative to this center.

To take into account the relative positions of the vertices, the **`RTC_CENTER`** is used to modify the *Model-View-Matrix* that is used for rendering. Initially, this is a matrix MV_{GPU}, stored in double-precision on the CPU. The **`RTC_CENTER`** is transformed into eye coordinates using the original Model-View matrix:

$$\text{RTC\_CENTER}_{Eye} = \text{MV}_{GPU} * \text{RTC\_CENTER}$$

The Model-View-Matrix that is used for rendering then is a matrix MV_{GPU} that is stored in single-precision, and which is created by replacing the last column of the original Model-View matrix with the resulting RTC_CENTER_{Eye}.

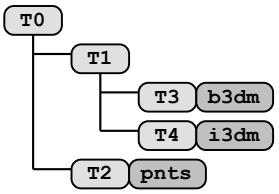
With this technique, it is possible to avoid the appearance of large values in the positions of the model, which would cause rendering artifacts due to the limited precision of the subsequent rendering pipeline: the positions can be given as small values, relative to the **`RTC_CENTER`**, and the modified Model-View matrix properly transforms them into the eye coordinate system for rendering.

13.4. Transforms in 3D Tiles

The renderable content in 3D Tiles may be given in different coordinate systems. For example, a tileset for a city could contain a nested tileset for a single building, and the latter could be given in its own coordinate system.

In order to convert between local coordinate systems, each tile has an optional **`transform`** property. This property is a column-major 4×4 affine transformation matrix and transforms the coordinate system of the tile into the coordinate system of the parent, defaulting to the identity matrix when it is not defined.

The transform matrix of a tile affects the positions, normals, and the bounding volumes of the tile and its content. The positions of features (like instances in an Instanced 3D Model, or points in a Point Cloud) are multiplied with the transform matrix to bring them from the local coordinate system into the coordinate system of the parent tile. The normals are multiplied with the top-left 3×3 matrix of the transpose of the inverse of the matrix, to properly take non-uniform scaling into account. The bounding volumes are transformed with the matrix, except the "region" bounding volumes, which are explicitly defined to be in EPSG:4979 coordinates.



An example of a tileset with tiles T0...T4, and different content types. The diagram shows the hierarchical structure of the tiles. The right-hand side shows the transform that is computed for each tile: the transform for each tile is computed as the product of all transforms on the path from the root tile to the respective tile.

If the content of a tile uses relative-to-center positions, then the **`RTC_CENTER`** must be considered as an additional translation for the vertices.

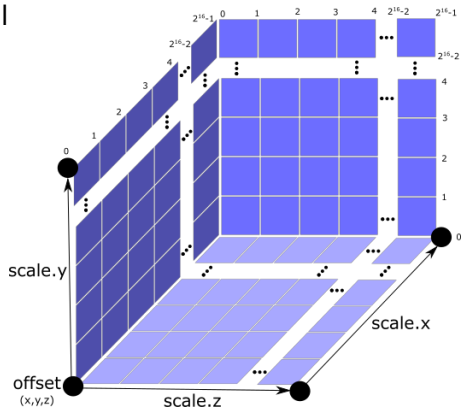
13.4.1 Coordinate Systems in 3D Tiles and glTF

3D Tiles defines the z-axis as up for local Cartesian coordinate systems. In contrast, glTF considers the y-axis as up. When glTF assets are embedded in Batched 3D Models and Instanced 3D Models, these different conventions must be considered by transforming the glTF asset at runtime so that the z-axis points upwards. Additionally, each glTF asset has its own node hierarchy with transforms. Details about how these different transforms play together, and the order in which they are applied to the renderable content, can be found in the 3D Tiles specification.

https://github.com/CesiumGS/3d-tiles/tree/master/specification#transforms

13.5. Quantized Positions and Oct-encoded Normals

The positions of instances in the Instanced 3D Model tile format as well as the positions of points in the Point Cloud tile format can be given in different forms: when they are given with the **`POSITION`** attribute, then they are stored as three 32-bit floating-point values, containing the x, y, and z coordinate of the instance or point.



For massive numbers of instances or points, 3D Tiles offers a way to store the positions more compactly: when the positions are defined using the **`POSITION_QUANTIZED`** attribute, then they are represented with three 16-bit unsigned integer values. This is achieved by storing the positions relative to the quantization volume which is defined by the **`QUANTIZED_VOLUME_OFFSET`** and **`QUANTIZED_VOLUME_SCALE`** attributes that are stored in the respective Feature Table.

https://github.com/CesiumGS/3d-tiles/blob/master/specification/TileFormats/Instanced3DModel/README.md#quantized-positions

Similarly, the normals of points and instances can be given as vectors consisting of three floating-point values, in the **`NORMAL`**, **`NORMAL_UP`**, and **`NORMAL_RIGHT`** attribute. For larger numbers of points or instances, these normals can be stored in a compressed form, indicated by the **`_OCT16P`** suffix of these attribute names. This compressed form consists of a bidirectional mapping: the normal vectors are mapped from the octants of a unit sphere to the faces of an octahedron, which are then projected to the plane and unfold into a unit square. Using this compression, a 3D normal vector can be represented with a single 16-bit value.

https://github.com/CesiumGS/3d-tiles/blob/master/specification/TileFormats/PointCloud/README.md#oct-encoded-normal-vectors