Open Geospatial Consortium

Submission Date 2022-11-28

Approval Date: <yyyy-mm-dd>

Publication Date: <yyyy-mm-dd>

External identifier of this OGC® document: <http://www.opengis.net/doc/[{doc-type/}]{standard}/{m.n}>

Internal reference number of this OGC® document: 20-072r3

Version: 1.1

Category: OGC® Community Standard

Editor: Hugo Ledoux, Balázs Dukai

CityJSON 1.1 Community Standard

Copyright notice

Copyright © 2022 Open Geospatial Consortium
To obtain additional rights of use, visit http://www.opengeospatial.org/legal/.

Warning

This document is not an OGC Standard. This document is distributed for review and comment. This document is subject to change without notice and may not be referred to as an OGC Standard.

Recipients of this document are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document type: OGC® Community Standard
Document subtype:
Document stage: Draft
Document language: English

Copyright © 2022 Open Geospatial Consortium

License Agreement

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

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

THIS LICENSE IS A COPYRIGHT LICENSE ONLY, AND DOES NOT CONVEY ANY RIGHTS UNDER ANY PATENTS THAT MAY BE IN FORCE ANYWHERE IN THE WORLD.

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

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

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

Copyright © 2022 Open Geospatial Consortium

CityJSON Specifications 1.1.3

Living Standard, 25 November 2022

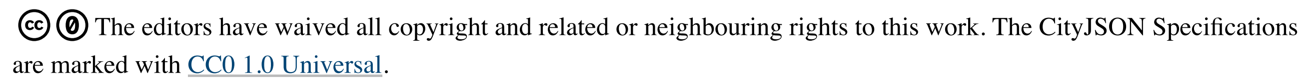
This version:
https://cityjson.org/specs/1.1.3/

Latest published version:
https://cityjson.org/specs/

Previous Versions:
https://cityjson.org/specs/overview/

Feedback:
GitHub

Editors:
Hugo Ledoux (TU Delft)
Balázs Dukai (3DGI)


The editors have waived all copyright and related or neighbouring rights to this work. The CityJSON Specifications are marked with CC0 1.0 Universal.

Abstract

CityJSON is a data exchange format for digital 3D models of cities and landscapes. It aims at being easy-to-use (for reading, processing, and creating datasets), and it was designed with programmers in mind, so that tools and APIs supporting it can be quickly built. The JSON-based encoding of CityJSON is conformant with a subset of the OGC CityGML data model (version 3.0). Using JSON instead of GML allows us to compress files by a factor 6 and at the same time to simplify greatly the structure of the files.

Table of Contents

1 CityJSON Object

2 The different City Objects

2.1 Attributes for all City Objects

2.2 Bridge
2.3 Building
2.4 CityFurniture
2.5 CityObjectGroup
2.6 LandUse
2.7 OtherConstruction
2.8 PlantCover
2.9 SolitaryVegetationObject
2.10 TINRelief
2.11 Transportation
2.12 Tunnel
2.13 WaterBody

3 Geometry Objects
3.1 Coordinates of the vertices
3.2 Arrays to represent boundaries
3.3 Semantics of geometric primitives
3.4 Geometry templates

4 Transform Object

5 Metadata
5.1 geographicalExtent (bbox)
5.2 identifier
5.3 pointOfContact
5.4 referenceDate
5.5 referenceSystem (CRS)
5.6 title

6 Appearance Object
6.1 Geometry Object having material(s)
6.2 Geometry Object having texture(s)
6.3 Material Object
6.4 Texture Object
6.5 Vertices-texture Object

7 Handling large files

7 Handling large files
7.1 Decomposing an area into parts/tiles
7.2 Text sequences and streaming with CityJSONFeature

8 Extensions
8.1 Using an Extension in a CityJSON file
8.2 The Extension file
8.3 Case 1: Adding new properties at the root of a document
8.4 Case 2: Defining attributes on existing City Objects
8.5 Case 3: Defining a new Semantic Object
8.6 Case 4: Creating and/or extending new City Objects
8.7 Rules to follow to define new City Objects

9 CityJSON Schemas

10 CityGML v3.0 conformance

§ 1. CityJSON Object

A CityJSON object represents one 3D city model of a given area, this model may contain features of different types, as defined in the CityGML data model.

A CityJSON object:

- is a JSON object.
- **must** have one member with the name `"type"`. The value must be `"CityJSON"`.
- **must** have one member with the name `"version"`. The value must be a string with the version (X.Y) of the CityJSON object. Observe that while schemas can have a version with patch version (X.Y.Z), a CityJSON object points only to the minor version (X.Y), and for validation the latest schema of that minor version should be used.
- **must** have one member with the name `"transform"`. The value is a JSON object describing how to *decompress* the integer coordinates of the geometries to obtain real-world coordinates. See § 4 Transform Object for details.
- **must** have one member with the name `"CityObjects"`. The value of this member is a collection of key-value pairs, where the key is the ID of the object, and the value is one City Object. The ID of a City Object should be unique (within one CityJSON Object). See § 2 The different City Objects for details.

- **must** have one member with the name `"vertices"`. The value is an array representing the coordinates of each vertex of the city model. See § 3.1 Coordinates of the vertices.
- **may** have one member with the name `"metadata"`. The value may be a JSON object describing the coordinate reference system used, the extent of the dataset, its creator, etc. See § 5 Metadata for details.
- **may** have one member with the name `"extensions"`, which is used if there are Extensions used in the file. See § 8 Extensions for details.
- **may** have one member with the name `"appearance"`. The value may contain JSON objects representing the textures and/or materials of surfaces. See § 6 Appearance Object for details.
- **may** have one member with the name `"geometry-templates"`, the value is a JSON object containing the templates that can be reused by different City Objects (usually for trees). This is equivalent to the concept of "implicit geometries" in CityGML. See § 3.4 Geometry templates for details.
- **may** have other members, and their value is not prescribed. Because these are not standard members in CityJSON, they might be ignored by parsers.

> **Suggested convention:** A file containing one CityJSON object may have the extension `'.city.json'`

The minimal valid CityJSON object is:

```
{
  "type": "CityJSON",
  "version": "1.1",
  "transform": {
    "scale": [1.0, 1.0, 1.0],
    "translate": [0.0, 0.0, 0.0]
  },
  "CityObjects": {},
  "vertices": []
}
```

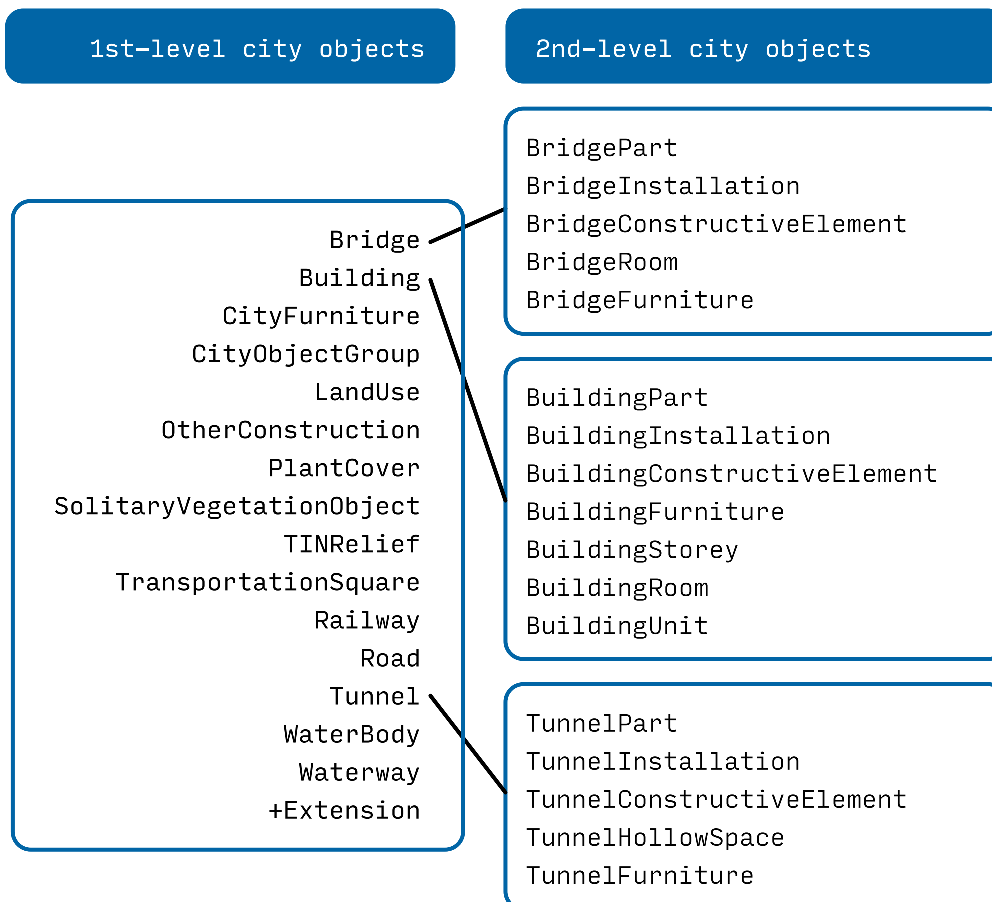
An "empty" but complete CityJSON object will look like this:

```
{
  "type": "CityJSON",
  "version": "1.1",
  "extensions": {},
```

```json
  "transform": {
    "scale": [1.0, 1.0, 1.0],
    "translate": [0.0, 0.0, 0.0]
  },
  "metadata": {},
  "CityObjects": {},
  "vertices": [],
  "appearance": {},
  "geometry-templates": {}
}
```

> While the order of the CityJSON member values should preferably be as above, not all JSON generators support this, therefore the order is not prescribed.

§ 2. The different City Objects



> There are 2 kinds of City Objects:
>
> 1. **1st-level**: City Objects that can "exist by themselves".
> 2. **2nd-level**: City Objects that need to have a `"parents"` to exist.
>
> This is because the schema of CityGML has been flattened. For example, a `"BuildingInstallation"` cannot be present in a dataset without being the `"children"` of a `"Building"`, but a `"Building"` can be present by itself.

A City Object:

- **must** have one member `"type"`. The value is one of the possibilities in the figure above (of type string). If an Extension is used, then the type can be any string starting with a `"+"`, as explained in § 8 Extensions.
- **may** have one member with the name `"geometry"`. The value is an array containing 0 or more Geometry Objects. More than one Geometry Object is used to represent several different levels-of-detail (LoDs) for the same object. However, the different Geometry Objects of a given City Object do not have to be of different LoDs.
- **may** have one member with the name `"attributes"`. The value is an object where the attributes of the City Object are listed.
- **may** have one member with the name `"geographicalExtent"` (the axis-aligned bounding box of the City Object). The value is an array with 6 values: `[minx, miny, minz, maxx, maxy, maxz]`
- **may** have one member `"children"`. The value is an array of the IDs (of type string) of the 2nd-level City Objects that are part of the 1st-level City Object. Only the direct children of the City Object are listed, not the grandchildren. Also, a City Object can have different types of City Objects as children, eg a `"Building"` can have both as children `"BuildingPart"` and `"BuildingInstallation"`. The order of the children in the array is not relevant.

A City Object of type 2nd-level:

- **must** have one member `"parents"`. The value is an array of the IDs (of type string) of the City Objects that are its parents. For the City Objects in the CityJSON core module, this array will always be of size 1 (only one parent). New City Objects defined in extensions can have more than one parents.
- **may** have one member `"children"`, and the `"children"` array only references the City Objects that are one level below the current one in the hierarchy. For instance a `"BuildingPart"` that contains a `"BuildingInstallation"`.

```
"CityObjects": {
  "id-1": {
    "type": "Building",
    "geographicalExtent": [ 84710.1, 446846.0, -5.3, 84757.1, 446944.0, 40.
    "attributes": {
      "measuredHeight": 22.3,
      "roofType": "gable",
      "owner": "Elvis Presley"
    },
    "children": ["id-2"],
    "geometry": [{...}]
  },
  "id-2": {
    "type": "BuildingPart",
    "parents": ["id-1"],
    "children": ["id-3"],
    ...
  },
  "id-3": {

    "type": "BuildingInstallation",
    "parents": ["id-2"],
    ...
  },
  "id-4": {
    "type": "LandUse",
    ...
  }
}
```

An example of a minimal valid City Object is:

```
{
  "type": "Building"
}
```

The above example is for a `"Building"` City Object, but any 1st-level City Object can be encoded the same way.

An example of a minimal 2nd-level valid City Object is:

```
{
  "type": "BuildingPart",
  "parents": ["id-parent"]
}
```

The above example is for a `"BuildingPart"`, but any 2nd-level City Object can be encoded this way.

§ 2.1. Attributes for all City Objects

The attributes for a given City Object are not prescribed (unlike in CityGML). This means that the `"attributes"` is a JSON object and its content is a JSON key-value pair ("owner" in the example above is one such attribute). Note that any valid JSON value (including an array and/or object) is a valid attribute value.

```
"CityObjects": {
  "id-1": {
    "type": "LandUse",
    "attributes": {
      "function": "Industry and Business",
      "area-parcel": {
        "value": 437,
        "uom": "m2"
      },
    },
    "geometry": [{...}]
  },
  "id-2": {
    "type": "WaterBody",
    "attributes": {
      "name": "Lake Black",
      "some-list": ["a", "b", "c"]
    },
    "geometry": [{...}]
  }
}
```

§ 2.2. Bridge

> See the CityGML v3.0.0 Bridge module for more details.

Six City Objects are related to bridges:

- `"Bridge"`
- `"BridgePart"`
- `"BridgeInstallation"`
- `"BridgeConstructiveElement"`
- `"BridgeRoom"`
- `"BridgeFurniture"`

The geometry of both `"Bridge"` and `"BridgePart"` can only be represented with these Geometry Objects: (1) `"Solid"`, (2) `"CompositeSolid"`, (3) `"MultiSurface"`, (4) `"CompositeSurface"`. The geometry of the four other objects can be represented with any of the Geometry Objects.

A City Object of type `"Bridge"` or `"BridgePart"` may have a member `"address"`, whose value is an array of JSON objects listing the potentially several addresses of that bridge . The properties of an address JSON object are free, to accommodate the different ways addresses are structured in different countries.

```
"CityObjects": {
  "LondonTower": {
    "type": "Bridge",
    "address": [
      {
        "City": "London",
        "Country": "UK"
      }
    ],
    "children": ["Bext1", "Bext2", "Inst-2017-11-14"],
    "geometry": [{
      "type": "MultiSurface",
      "lod": "2",
      "boundaries": [
        [[0, 3, 2, 1]],
        [[4, 5, 6, 7]],
        [[0, 1, 5, 4]],
        [[1, 2, 6, 5]],
        [[2, 3, 7, 6]],
        [[3, 0, 4, 7]]
```

```
        ]
      }]
    }
  }
```

§ 2.3. Building

> See the CityGML v3.0.0 Building module for more details.

Eight City Objects are related to buildings:

- `"Building"`
- `"BuildingPart"`
- `"BuildingInstallation"`
- `"BuildingConstructiveElement"`
- `"BuildingFurniture"`
- `"BuildingStorey"`
- `"BuildingRoom"`
- `"BuildingUnit"`

The geometry of `"Building"`, `"BuildingPart"`, `"BuildingStorey"`, `"BuildingRoom"`, and `"BuildingUnit"` can only be represented with these Geometry Objects: (1) `"Solid"`, (2) `"CompositeSolid"`, (3) `"MultiSurface"`, (4) `"CompositeSurface"`.

All eight City Objects, except `"Building"`, must have a `"parents"` property. The installations, furnitures, and subdivisions can have as parents a `"Building"`, a `"BuildingPart"`, or a `"BuildingRoom"`.

The geometry of `"BuildingInstallation"`, `"BuildingConstructiveElement"`, or `"BuildingFurniture"` objects can be represented with any of the Geometry Objects.

A City Object of type `"Building"`, `"BuildingPart"` or `"BuildingUnit"` may have a member `"address"`, consisting of an array of JSON objects listing one or more addresses of that building (an apartment building could contain several for instance). The properties of an address JSON object are free format, to accommodate the different ways addresses are described in different countries. If a location is necessary (eg to locate the position of the front door) then a property `"location"` should be used, and it should contain a `"MultiPoint"` geometry.

```
"CityObjects": {
  "id-1": {
    "type": "Building",
    "attributes": {
      "roofType": "gabled roof"
    },
    "geographicalExtent": [ 84710.1, 446846.0, -5.3, 84757.1, 446944.0, 40.
    "children": ["id-56", "id-832", "mybalcony"]
  },
  "id-56": {
    "type": "BuildingPart",
    "parents": ["id-1"],
    ...
  },
  "mybalcony": {
    "type": "BuildingInstallation",
    "parents": ["id-1"],
    ...


  }
  ...
}
```

```
"myroom": {
  "type": "BuildingRoom",
  "attributes": {
    "usage": "living room"
  },
  "parents": ["id-1"],
  "geometry": [{
    "type": "Solid",
    "lod": "2",
    "boundaries": [
      [ [[0, 3, 2, 1]], [[4, 5, 6, 7]], [[0, 1, 5, 4]], ... ]
    ]
  }]
}
```

```
{
  "type": "Building",
  "address": [
    {
      "Country": "Canada",
      "Locality": "Chibougamau",
      "ThoroughfareNumber": "1",
      "ThoroughfareName": "rue de la Patate",
      "Postcode": "H0H 0H0",
      "location": {
        "type": "MultiPoint",
        "lod": "1",
        "boundaries": [231]
      }
    }
  ]
}
```

§ 2.4. CityFurniture

> See the CityGML v3.0.0 CityFurniture module for more details.

The geometry of a City Object of type `"CityFurniture"` can be any Geometry Object.

```
"mystopsign": {
  "type": "CityFurniture",
  "attributes": {
    "function": "bus stop"
  },
  "geometry": [{
    "type": "MultiSurface",
    "lod": "2",
    "boundaries": [
      [[0, 3, 2, 1]], [[4, 5, 6, 7]], [[0, 1, 5, 4]]
    ]
  }]
}
```

§ 2.5. CityObjectGroup

> See the CityGML v3.0.0 CityObjectGroup module for more details.

The CityGML concept of *groups*, where City Objects are aggregated based on certain criteria (think of a neighbourhood in a city for instance), is possible in CityJSON. The group is a City Object, and it can contain, if needed, a geometry (the polygon representing the neighbourhood for instance).

Since a `"CityObjectGroup"` is also a City Object, it can be part of another group.

A City Object of type `"CityObjectGroup"`:

- **must** have a member `"children"`. The value is an array of the IDs of the City Objects that the group contains. As for other City Objects, the City Objects must have the ID of the group in `"parents"`.
- **may** have a member `"children_roles"`, whose value is an array of strings describing the role of each City Object in the group. This member must be of the same length as that of `"children"`.
- **may** have a member `"attributes"`.
- **may** have a member `"geometry"`. Notice that since the `"CityObjectGroup"` is a container of different City Objects, the concept of Level of Detail does not apply to it. Nevertheless, the `"lod"` property is still used for enforcing uniformity with all the other geometries.

```
"CityObjects": {
  "my-neighbourhood": {
    "type": "CityObjectGroup",
    "children": ["building1", "building2", "building666"]
  }
}
```

```
"CityObjects": {
  "my-neighbourhood": {
    "type": "CityObjectGroup",
    "attributes": {
      "location": "Chibougamau Sud"
    },
    "children": ["building1", "building666"],
    "children_roles": ["residential building", "voting location"],
    "geometry": [{
      "type": "MultiSurface",
```

```
      "lod": "2",
      "boundaries": [ [[2, 41, 5, 77]] ]
    }]
  }
}
```

§ 2.6. LandUse

> See the CityGML v3.0.0 LandUse module for more details.

The geometry of a City Object of type `"LandUse"` can be of type `"MultiSurface"` or `"CompositeSurface"`.

```
"oneparcel": {
  "type": "LandUse",
  "geometry": [{
    "type": "MultiSurface",
    "lod": "1",
    "boundaries": [
      [[0, 3, 2, 1]], [[4, 5, 6, 7]], [[0, 1, 5, 4]]
    ]
  }]
}
```

§ 2.7. OtherConstruction

> See the CityGML v3.0.0 Construction module for more details (OtherConstruction is one class).

This is used for constructions that are not buildings, bridges, or tunnels. Examples are:

- electricity pylon
- fence
- permanent water tank
- pontoon
- railway platform

- shed
- windmill

The geometry of a City Object of type "OtherConstruction" can be any Geometry Object.

```
"mypylon": {
  "type": "OtherConstruction",
  "attributes": {
    "class": "windmill",
    "conditionOfConstruction": "underConstruction"
  },
  "geometry": [{
    "type": "MultiSurface",
    "lod": "2",
    "boundaries": [
       [[0, 3, 2, 1]], [[4, 5, 6, 7]], [[0, 1, 5, 4]], ...
    ]
  }]
}
```

§ 2.8. PlantCover

> See the CityGML v3.0.0 Vegetation module for more details (PlantCover is one class).

The geometry of a City Object of type "PlantCover" can be of type: (1) "Solid", (2) "CompositeSolid", (3) "MultiSolid", (4) "MultiSurface", (5) "CompositeSurface".

```
"myplants": {
  "type": "PlantCover",
  "attributes": {
    "averageHeight": 11.05
  },
  "geometry": [{
    "type": "MultiSolid",
    "lod": "2",
    "boundaries": [
```

```
        [
          [ [[0, 3, 2, 1]], [[4, 5, 6, 7]], [[0, 1, 5, 4]], [[10, 13, 22, 31]
        ],
        [
          [ [[5, 34, 31, 12]], [[44, 54, 62, 74]], [[111, 123, 922, 66]] ]
        ]
      ]
    }]
  }
```

2.9. SolitaryVegetationObject

> See the CityGML v3.0.0 Vegetation module for more details (SolitaryVegetationObject is one class).

The geometry of a City Object of type `"SolitaryVegetationObject"` can be any Geometry Object.

```
"onebigtree": {
  "type": "SolitaryVegetationObject",
  "attributes": {
    "trunkDiameter": 5.3,
    "crownDiameter": 11.0
  },
  "geometry": [{
    "type": "MultiPoint",
    "lod": "1",
    "boundaries": [1]
  }]
}
```

2.10. TINRelief

> See the CityGML v3.0.0 Relief module for more details (TINRelief is one class).

The geometry of a City Object of type `"TINRelief"` can only be of type `"CompositeSurface"`.

CityJSON does not define a specific Geometry Object for a TIN (triangulated irregular network), and is simply a CompositeSurface for which every surface is a triangle (thus a polygon having 3 vertices, and no interior ring).

Notice that in practice any `"CompositeSurface"` is allowed for encoding a terrain, and that arbitrary polygons could also be used (not just triangles).

```
"myterrain01": {
  "type": "TINRelief",
  "geographicalExtent": [ 84710.1, 446846.0, -5.3, 84757.1, 446944.0, 40.9
  "geometry": [{
    "type": "CompositeSurface",
    "lod": "1",
    "boundaries": [
       [[0, 3, 2]], [[4, 5, 6]], [[1, 2, 6]], [[2, 3, 7]], [[3, 0, 4]]
    ]
  }]
}
```

§ 2.11. Transportation

> See the CityGML v3.0.0 Transportation module for more details.

Four City Objects are related to transportation:

- `"Road"`
- `"Railway"`
- `"Waterway"`
- `"TransportSquare"` (to model for instance parking lots and squares)

Observe that the "Section", "Intersection", and "Track" classes from CityGML are omitted because they can be easily specified using specific attributes.

```
"ma_rue": {
  "type": "Road",
  "attributes": {
    "class": "backwards",
```

```
      "clearanceSpace": 2.23,
      "clearanceSpaceUnit": "meter"
    },
    "children": ["sect1", "sect2"],
    "geometry": [...]
  }
  "sect1": {
    "type": "Road",
    "attributes": {
      "class": "section"
    },
    "parents": ["ma_rue"],
    "geometry": [...],
  }
```

Similarly, the CityGML classes "TrafficArea", "AuxiliaryTrafficArea", "Marking", and "Hole" are implemented as semantic surface (see § 3.3 Semantics of geometric primitives). That is, the surface representing a road should be split into sub-surfaces (therefore forming a `"MultiSurface"` or a `"CompositeSurface"`) in which each of the sub-surfaces has semantics.

```
  "ma_rue": {
    "type": "Road",
    "geometry": [{
      "type": "MultiSurface",
      "lod": "2",
      "boundaries": [
         [[0, 3, 2, 1, 4]], [[4, 5, 6, 666, 12]], [[0, 1, 5]], [[20, 21, 75]]
      ]
    }],
    "semantics": {
      "surfaces": [
        {
          "type": "TrafficArea",
          "surfaceMaterial": ["asphalt"],
          "function": "road"
        },
        {
          "type": "AuxiliaryTrafficArea",
          "function": "green areas"
        },
        {

          "type": "TrafficArea"
```

```
        "type": "TrafficArea",
        "surfaceMaterial": ["dirt"],
        "function": "road"
      }
    ],
    "values": [0, 1, null, 2]
  }
}
```

§ 2.12. Tunnel

> See the CityGML v3.0.0 Tunnel module for more details.

Six City Objects are related to tunnels:

- `"Tunnel"`
- `"TunnelPart"`
- `"TunnelInstallation"`
- `"TunnelConstructiveElement"`
- `"TunnelHollowSpace"`
- `"TunnelFurniture"`

The geometry of both `"Tunnel"` and `"TunnelPart"` can only be represented with these Geometry Objects: (1) `"Solid"`, (2) `"CompositeSolid"`, (3) `"MultiSurface"`, (4) `"CompositeSurface"`.

The geometry of the other four objects can be represented with any of the Geometry Objects.

```
"CityObjects": {
  "Lærdalstunnelen": {
    "type": "Tunnel",
    "attributes": {
      "yearOfConstruction": 2000,
      "length": "24.5km"
    },
    "children": ["stoparea1"],
    "geometry": [{
      "type": "Solid",
```

```
        "lod": "2",
        "boundaries": [
          [ [[0, 3, 2, 1]], [[4, 5, 6, 7]], [[0, 1, 5, 4]] ]
        ]
      }]
    }
  }
```

§ 2.13. WaterBody

> See the CityGML v3.0.0 WaterBody module for more details.

The geometry of a City Object of type `"WaterBody"` can be of types: `"MultiLineString"`, `"MultiSurface"`, `"CompositeSurface"`, `"Solid"`, or `"CompositeSolid"`.

```
"mygreatlake": {
  "type": "WaterBody",
  "attributes": {
    "usage": "leisure",
  },
  "geometry": [{
    "type": "Solid",
    "lod": "2",
    "boundaries": [
      [ [[0, 3, 2, 1]], [[4, 5, 6, 7]], [[0, 1, 5, 4]] ]
    ]
  }]
}
```

§ 3. Geometry Objects

CityJSON defines the following 3D geometric primitives, all of which are embedded in 3D space (and therefore their vertices have *(x, y, z)* coordinates). The indexing mechanism of the format Wavefront OBJ is reused, that is a geometry does not store the locations of its vertices, but points to a vertex in a list (property `"vertices"` in the CityJSON Object).

As is the case in CityGML, only linear and planar primitives are allowed. No curves or parametric surfaces can be represented.

A Geometry object is a JSON object for which the type member's value is one of the following:

1. `"MultiPoint"`
2. `"MultiLineString"`
3. `"MultiSurface"`
4. `"CompositeSurface"`
5. `"Solid"`
6. `"MultiSolid"`
7. `"CompositeSolid"`
8. `"GeometryInstance"` (this is another type with different properties, see § 3.4 Geometry templates)

A Geometry object:

- **must** have one member with the name `"type"`. The value must be a string with one of the 8 allowed Geometry types, as defined above.
- **must** have one member with the name `"lod"`. The value must be a string with the LoD identifying the level-of-detail (LoD) of the geometry. This can be either a single digit (following the CityGML standards), or "X.Y"-formatted if the improved LoDs by TU Delft are used.
- **must** have one member with the name `"boundaries"`. The value is a hierarchy of arrays (the depth depends on the Geometry object) with integers. An integer refers to the index in the `"vertices"` array of the CityJSON object, and it is 0-based (ie the first element in the array has the index "0", the second one "1", etc.).
- **may** have one member `"semantics"`. The value is a JSON Object, as defined below.
- **may** have one member `"material"`. The value is a JSON Object, as defined below.
- **may** have one member `"texture"`. The value is a JSON Object, as defined below.

> There is *no* Geometry Object for MultiGeometry. Instead, for the `"geometry"` member of a CityObject, the different geometries may be enumerated in the array (all with the same value for the member `"lod"`).

§ 3.1. Coordinates of the vertices

A CityJSON must have one member named `"vertices"`. The value is an array of array of integers representing the coordinates of each vertex of the city model. The position of a vertex in this array (0-

based) is used to represent the "boundaries" of Geometry Objects.

- one vertex **must** be an array with exactly 3 integers, representing the *(x,y,z)* location of the vertex before it is transformed to its real-world coordinates (with §4 Transform Object).
- the array of vertices may be empty.
- vertices may be repeated.

```
"vertices": [
  [102, 103, 1],
  [11, 910, 43],
  [25, 744, 22],
  ...
  [23, 88, 5],
  [8523, 487, 22]
]
```

§ 3.2. Arrays to represent boundaries

The depth of the hierarchy of arrays depends on the Geometry object, and is as follows.

- A "MultiPoint" has an array with the indices of the vertices; this array can be empty.
- A "MultiLineString" has an array of arrays, each containing the indices of a LineString.
- A "MultiSurface", or a "CompositeSurface", has an array containing surfaces, each surface is modelled by an array of array, the first array being the exterior boundary of the surface, and the others the interior boundaries.
- A "Solid" has an array of shells, the first array being the exterior shell of the solid, and the others the interior shells. Each shell has an array of surfaces, modelled in the exact same way as a MultiSurface/CompositeSurface.
- A "MultiSolid", or a "CompositeSolid", has an array containing solids, each solid is modelled as above.

> JSON does not allow comments, the comments in the example below (C++ style: `//-- my comments`) are only to explain the cases, and should be removed.

```json
{
  "type": "MultiPoint",
  "lod": "1",
  "boundaries": [2, 44, 0, 7]
}
```

```json
{
  "type": "MultiLineString",
  "lod": "1",
  "boundaries": [
    [2, 3, 5], [77, 55, 212]
  ]
}
```

```json
{
  "type": "MultiSurface",
  "lod": "2",
  "boundaries": [
    [[0, 3, 2, 1]], [[4, 5, 6, 7]], [[0, 1, 5, 4]]
  ]
}
```

```json
{
  "type": "Solid",
  "lod": "2",
  "boundaries": [
    //-- exterior shell
    [ [[0, 3, 2, 1, 22]], [[4, 5, 6, 7]], [[0, 1, 5, 4]], [[1, 2, 6, 5]] ],
    //-- interior shell
    [ [[240, 243, 124]], [[244, 246, 724]], [[34, 414, 45]], [[111, 246, 5]
  ]
}
```

```json
{
  "type": "CompositeSolid",
  "lod": "3",
  "boundaries": [
    [ //-- 1st Solid
      [ [[0, 3, 2, 1, 22]], [[4, 5, 6, 7]], [[0, 1, 5, 4]], [[1, 2, 6, 5]]
```

```
      [ [[240, 243, 124]], [[244, 246, 724]], [[34, 414, 45]], [[111, 246,
    ],
    [ //-- 2nd Solid
      [ [[666, 667, 668]], [[74, 75, 76]], [[880, 881, 885]], [[111, 122, 2
    ]
  ]
}
```

See this tutorial for further explanation on the depth of arrays of Geometry objects.

§ 3.3. Semantics of geometric primitives

A Semantic Object is a JSON object representing the semantics of a primitive of a geometry (e.g. a surface of a building). A Semantic Object may also represent other attributes of the primitive (e.g. the slope of the roof, or the solar potential). For surface and volumetric geometries (e.g. `MultiSurface`, `Solid` and `MultiSolid`), a primitive is a surface. If a geometry is a `MultiPoint` or a `MultiLineString`, then the primitives are its respective sub-parts: points and linestrings.

A Semantic Object:

- **must** have one member with the name `"type"`. The value is one of the allowed value. These depend on the City Object (see below).
- **may** have an attribute `"parent"`. The value is an integer pointing to another Semantic Object of the same geometry (index of it, 0-based). This is used to explicitly represent to which wall or roof a window or door belongs to; there can be only one parent.
- **may** have an attribute `"children"`. The value is an array of integers pointing to other Semantic Objects of the same geometry (index of it, 0-based). This is used to explicitly represent the openings (windows and doors) of walls and roofs.
- **may** have other attributes in the form of a JSON key-value pair, where the value must not be a JSON object (but a string/number/integer/boolean).

```
{
  "type": "RoofSurface",
  "slope": 16.4,
  "children": [2, 37],
  "solar-potential": 5
}
```

```
{
  "type": "Window",
  "parent": 2,
  "type-glass": "HR++"
}
```

"Building", "BuildingPart", "BuildingRoom", "BuildingStorey", "BuildingUnit", and "BuildingInstallation" can have the following semantics:

- "RoofSurface"
- "GroundSurface"
- "WallSurface"
- "ClosureSurface"
- "OuterCeilingSurface"
- "OuterFloorSurface"
- "Window"
- "Door"
- "InteriorWallSurface"
- "CeilingSurface"
- "FloorSurface"

For "WaterBody":

- "WaterSurface"
- "WaterGroundSurface"
- "WaterClosureSurface"

For Transportation ("Road", "Railway", "TransportSquare"):

- "TrafficArea"
- "AuxiliaryTrafficArea"
- "TransportationMarking"
- "TransportationHole"

It is possible to define and use other semantics, but these have to start with a `"+"`, inline with the rules defined in the § 8 Extensions.

```
{
  "type": "+SupportingWall"
}
```

Because in a given City Object (say a `"Building"`) several primitives can have the same semantics (think of a complex building that has been triangulated, there can be dozens of triangles used to model the same surface), a Semantic Object must be declared once, and each of the primitives that are represented by it points to it. This is achieved by first declaring all the Semantic Objects in an array, and then having an array where each primitive links to Semantic Objects (position in the array).

If a Geometry object has semantics, then the Geometry object:

- **must** have one member with the name `"semantics"`, whose values are two properties: `"surfaces"` and `"values"`. Both **must** be present.

Also:

- the value of `"surfaces"` is an array of Semantic Objects.
- the value of `"values"` is a hierarchy of arrays with integers. The depth depends on the Geometry object: for `MultiPoint` and `MultiLineString` this is a simple array of integers; for any other geometry type it is two less than the array `"boundaries"`. An integer refers to the index in the `"surfaces"` array of the same geometry, and it is 0-based. If one surface has no semantics, a value of `null` must be used.

> For legacy reasons, we use `"surfaces"` to name the array of Semantic Object. Nevertheless, this property is used for points and linestrings of `MultiPoints` and `MultiLineStrings`, as well.

```
{
  "type": "MultiSurface",
  "lod": "2",
  "boundaries": [
    [[0, 3, 2, 1]],
    [[4, 5, 6, 7]],
    [[0, 1, 5, 4]],
    [[0, 2, 3, 8]],
    [[10, 12, 23, 48]]
  ],
```

```
    "semantics": {
      "surfaces" : [
        {
          "type": "WallSurface",
          "slope": 33.4,
          "children": [2]
        },
        {
          "type": "RoofSurface",
          "slope": 66.6
        },
        {
          "type": "+PatioDoor",
          "parent": 0,
          "colour": "blue"
        }
      ],
      "values": [0, 0, null, 1, 2]
    }
  }
```

```
  {
     "type": "CompositeSolid",
     "lod": "2.2",
     "boundaries": [
       [ //-- 1st Solid
         [ [[0, 3, 2, 1, 22]], [[4, 5, 6, 7]], [[0, 1, 5, 4]], [[1, 2, 6, 5]]
       ],
       [ //-- 2nd Solid
         [ [[666, 667, 668]], [[74, 75, 76]], [[880, 881, 885]] ]
       ]
     ],
     "semantics": {
       "surfaces" : [
         {
           "type": "RoofSurface"
         },
         {
           "type": "WallSurface"
         }
       ],
       "values": [
         [ //-- 1st Solid
```

```
          [0, 1, 1, null]
        ],
        [ //-- 2nd Solid get all null values
          [null, null, null]
        ]
      ]
    }
  }
```

§ 3.4. Geometry templates

CityGML's "ImplicitGeometries", better known in computer graphics as *templates*, are one method to compress files since the geometries (such as benches, lamp posts, and trees), need only be defined once. In CityJSON, they are implemented differently from what is specified in CityGML: they are defined separately in the file, and each template can be reused. By contrast, in CityGML, the geometry used for a given City Object is reused by other City Objects, there is thus no central location where all templates are stored.

The Geometry Templates are defined as a JSON object that:

- **must** have one member with the name `"templates"`. The value is an array of Geometry Objects.
- **must** have one member with the name `"vertices-templates"`. The value is an array of coordinates of each vertex of the templates (0-based indexing). The reason the vertices index are not global is to ensure that operations on the vertices (eg for CRS transformation, for §4 Transform Object, or calculating the bounding box of a dataset) will not be affected by the templates (since they will often be defined locally, and translated/rotated/scaled to their final position).

```
"geometry-templates": {
  "templates": [
    {
      "type": "MultiSurface",
      "lod": "2.1",
      "boundaries": [
         [[0, 3, 2, 1]], [[4, 5, 6, 7]], [[0, 1, 5, 4]]
      ]
    },
    {
```

```
        "type": "MultiSurface",
        "lod": "1.3",
        "boundaries": [
          [[1, 2, 6, 5]], [[2, 3, 7, 6]], [[3, 0, 4, 7]]
        ]
      }
    ],
    "vertices-templates": [
      [0.0, 0.5, 0.0],
      ...
      [1.0, 1.0, 0.0],
      [0.0, 1.0, 0.0]
    ]
  }
```

A given template can be used as the geometry (or as one of the geometries) of a City Object. A new JSON object of type `"GeometryInstance"` is defined, and it:

- **must** have one member with the name `"template"`, whose value is the position of the template in the `"geometry-templates"` (0-indexing).
- **must** have one member with the name `"boundaries"`, whose value is an array containing only one vertex index, which refers to one vertex in the `"vertices"` property of a CityJSON file. (This is the reference point from which the transformations are applied, it is the "referencePoint" in CityGML.)
- **must** have one member with the name `"transformationMatrix"`, whose value is a 4x4 matrix (thus 16 values in an array) defining the rotation/translation/scaling of the template. Note that these 16 values are ordered row-by-row, as the example below shows.

```
  {
    "type": "SolitaryVegetationObject",
    "geometry": [
      {
        "type": "GeometryInstance",
        "template": 0,
        "boundaries": [372],
        "transformationMatrix": [
          2.0, 0.0, 0.0, 0.0,
          0.0, 2.0, 0.0, 0.0,
          0.0, 0.0, 2.0, 0.0,
          0.0, 0.0, 0.0, 1.0
        ]
```

```
    }
  ]
}
```

> The CityJSON website has a page to help developers with Geometry Templates, it contains simple examples, explains which transformations to apply to obtain world coordinates, and explains how matrices work (for instance, in the example above, a scaling of 2.0 is applied).

§ 4. Transform Object

To reduce the size of a CityJSON object (and thus the size of files) and to ensure that only a fixed number of digits is stored for the coordinates of the geometries, the coordinates of the vertices of the geometries are represented integer values. We therefore need to store the scale factor and the translation needed to obtain the original coordinates (stored with floats/doubles).

A CityJSON object must therefore have one member `"transform"`, whose values are 2 mandatory JSON objects (`"scale"` and `"translate"`), both arrays with 3 values.

The scheme of TopoJSON (called quantization) is reused, and here we simply add a third coordinate because our vertices are embedded in 3D space.

It should be noticed that only the `"vertices"` at the root of the CityJSON object are affected by the transformation, the vertices for the Geometric templates and textures are not.

To obtain the real position of a given vertex *v*, we must take the 3 values *vi* listed in the `"vertices"` member and:

```
v[0] = (vi[0] * ["transform"]["scale"][0]) + ["transform"]["translate"][0]
v[1] = (vi[1] * ["transform"]["scale"][1]) + ["transform"]["translate"][1]
v[2] = (vi[2] * ["transform"]["scale"][2]) + ["transform"]["translate"][2]
```

The following `"transform"` means that 2 important digits are kept (thus millimetre level if meters are the units of the CRS), and the `"translate"` usually matches with the minimum values of the axis-aligned bounding box (but does not need to).

```
"transform": {
    "scale": [0.001, 0.001, 0.001],
    "translate": [442464.879, 5482614.692, 310.19]
}
```

§ 5. Metadata

The core of CityJSON supports the following six properties, these are compliant with the international standard ISO19115.

```
"metadata": {
  "geographicalExtent": [ 84710.1, 446846.0, -5.3, 84757.1, 446944.0, 40.9
  "identifier": "eaeceeaa-3f66-429a-b81d-bbc6140b8c1c",
  "pointOfContact": {
    "contactName": "3D geoinformation group, Delft University of Technolog
    "contactType": "organization",
    "role": "owner",
    "phone": "+31-6666666666",
    "emailAddress": "3dgeoinfo-bk@tudelft.nl",
    "website": "https://3d.bk.tudelft.nl",
    "address": "Julianalaan 134, Delft 2628BL, the Netherlands"
  },
  "referenceDate": "1977-02-28",
  "referenceSystem": "https://www.opengis.net/def/crs/EPSG/0/2355",
  "title": "Buildings in LoD2.3 of Chibougamau, Québec"
}
```

The storage of additional ISO19115-compliant metadata attributes and/or of statistics related to 3D city models can be done with the MetadataExtended Extension. Examples of extra attributes/properties that can be stored: point of contact for the dataset, lineage, statistics about the present LoDs, the presence of textures/materials, etc.

§ 5.1. geographicalExtent (bbox)

While the geographical extent can be computed from the dataset itself, it is often useful to store it. It may be stored as an array with 6 values: `[minx, miny, minz, maxx, maxy, maxz]`. Notice that these are in the real coordinates of the dataset (based on § 5.5 referenceSystem (CRS)) and not after the coordinates have been compressed with the `"transform"` property (§ 4 Transform Object).

```
"metadata": {
  "geographicalExtent": [ 84710.1, 446846.0, -5.3, 84757.1, 446944.0, 40.9
}
```

5.2. identifier

A unique identifier for the dataset. It is recommend to use universally unique identifier, but this is not necessary.

```
"metadata": {
  "identifier": "44574905-d2d2-4f40-8e96-d39e1ae45f70"
}
```

5.3. pointOfContact

The point of contact for the dataset. This is a JSON object that

- **must** have one member with the name `"contactName"`. The value is the name of the contact.
- **must** have one member with the name `"emailAddress"`. The value is a string with the email.
- **may** have one member with the name `"role"`. The value describes the role that contact person/organisation has, it is one of the following: `"resourceProvider"`, `"custodian"`, `"owner"`, `"user"`, `"distributor"`, `"originator"`, `"pointOfContact"`, `"principalInvestigator"`, `"processor"`, `"publisher"`, `"author"`, `"sponsor"`, `"co-author"`, `"collaborator"`, `"editor"`, `"mediator"`, `"rightsHolder"`, `"contributor"`, `"funder"`, `"stakeholder"`.
- **may** have one member with the name `"website"`. The value is the URL of point of contact.
- **may** have one member with the name `"contactType"`. The value is a string which is either `"individual"` or `"organization"`. For an `"organization"`, the `"website"` can also be given.
- **may** have one member with the name `"address"`. The value is a string with the full address.
- **may** have one member with the name `"phone"`. The value is a string with the phone number.
- **may** have one member with the name `"organization"`. The value is the name of the organisation, to be used if the `"contactName"` is the name of a person.

```
"pointOfContact": {
  "contactName": "Justin Trudeau",
  "emailAddress": "justin.trudeau@parl.gc.ca",
  "phone": "+1-613-992-4211",
  "address": "24 Sussex Drive, Ottawa, Canada",
  "contactType": "individual",
  "role": "pointOfContact"
}
```

§ 5.4. referenceDate

The date when the dataset was compiled, without the time of the day, only a `"full-date"` as defined in RFC 3339, Section 5.6 should be used.

```
"metadata": {
  "referenceDate": "1977-02-28"
}
```

> JSON does not have a date type, and thus the representations defined by RFC 3339, Section 5.6 should be used. A simple date is `"full-date"` (thus `"1977-07-11"` as a string), and should be used for the metadata above.
>
> Other attributes in a CityJSON object can also have a date with a time, and such an attribute is specified as a `"full-time"`. For example `"1985-04-12T23:20:50.52Z"` (stored as a string).

§ 5.5. referenceSystem (CRS)

The coordinate reference system (CRS) is given as a URL formatted according to the OGC Name Type Specification:

`http://www.opengis.net/def/crs/{authority}/{version}/{code}`

where `{authority}` designates the authority responsible for the definition of this CRS (usually "EPSG" or "OGC"), and where `{version}` designates the specific version of the CRS ("0" (zero) is used if there is no version).

For instance, the Dutch national CRS in 3D:

```
"metadata": {
  "referenceSystem": "https://www.opengis.net/def/crs/EPSG/0/7415"
}
```

Be aware that the CRS should be a three-dimensional one, ie the elevation/height values should be with respect to a specific datum.

> Unlike in (City)GML where each object can have a different CRS (eg a wall of a building could theoretically have a different from the other walls used to represent the building), in CityJSON all the city objects need to be in the same CRS.

§ 5.6. title

A string describing the dataset.

```
"metadata": {
  "title": "3D city model of Chibougamau, Canada"
}
```

§ 6. Appearance Object

Both textures and materials are supported in CityJSON, and the same mechanisms used in CityGML are reused, so the conversion back-and-forth is easy. The material is represented with the X3D specifications, as is the case for CityGML. For the texture, the COLLADA standard is reused, as is the case for CityGML. However:

- the CityGML class `GeoreferencedTexture` is not supported.
- the CityGML class `TexCoordGen` is not supported, ie one must specify the UV coordinates in the texture files.
- the major difference is that in CityGML each Material/Texture object keeps a list of the primitives using it, while in CityJSON it is the opposite: if a primitive has a Material/Texture then it is stated with the primitive (with a link to it).

An Appearance Object is a JSON object that

- **may** have one member with the name `"materials"`, whose value is an array of Material Objects.
- **may** have one member with the name `"textures"`, whose value is an array of Texture Objects.
- **may** have both `"materials"` and `"textures"`.
- **may** have one member with the name `"vertices-texture"`, whose value is an array of coordinates of each so-called UV vertex of the city model.
- **may** have one member with the name `"default-theme-texture"`, whose value is the name of the default theme for the appearance (a string). This can be used if geometries have more than one textures, so that a viewer displays the default one.
- **may** have one member with the name `"default-theme-material"`, whose value is the name of the default theme for the material (a string). This can be used if geometries have more than one textures, so that a viewer displays the default one.

```
"appearance": {
  "materials": [],
  "textures":[],
  "vertices-texture": [],
  "default-theme-texture": "myDefaultTheme1",
  "default-theme-material": "myDefaultTheme2"
}
```

§ 6.1. Geometry Object having material(s)

Each surface in a Geometry Object can have one or more materials assigned to it. To store the material of a surface, a Geometry Object may have a member `"material"`, the value of this member is a collection of key-value pairs, where the key is the *theme* of the material, and the value is one JSON object that **must** contain either:

- one member `"values"`. The value is a hierarchy of arrays with integers. Each integer refers to the position (0-based) in the `"materials"` member of the `"appearance"` member of the CityJSON object. If a surface has no material, then `null` should be used in the array. The depth of the array depends on the Geometry object, and is equal to the depth of the `"boundary"` array minus 2, because each surface (`[[]]`) gets one material.
- one member `"value"`. The value is one integer referring to the position (0-based) in the `"materials"` member of the `"appearance"` member of the CityJSON object. This is used be-

cause often the materials are used to colour full objects, and repetition of materials is not necessary.

In the following, the Solid has 4 surfaces, and there are 2 themes ("irradiation" and "irradiation-2"). These could represent, for instance, the different colours based on different scenarios of an solar irradiation analysis. Notice that the last surface gets no material (for both themes), thus `null` is used.

```
{
  "type": "Solid",
  "lod": "2.1",
  "boundaries": [
    [ [[0, 3, 2, 1]], [[4, 5, 6, 7]], [[0, 1, 5, 4]], [[1, 2, 6, 5]] ]
  ],
  "material": {
    "irradiation": {
      "values": [[0, 0, 1, null]]
    },
    "irradiation-2": {
      "values": [[2, 2, 1, null]]
    }
  }
}
```

§ 6.2. Geometry Object having texture(s)

To store the texture(s) of a surface, a Geometry Object may have a member with the value `"texture"`, its value is a collection of key-value pairs, where the key is the *theme* of the textures, and the value is one JSON object that must contain one member `"values"`, which is a hierarchy of arrays with integers. For each ring of each surface, the first value refers to the position (0-based) in the `"textures"` member of the `"appearance"` member of the CityJSON object. The other indices refer to the UV positions of the corresponding vertices (as listed in the `"boundaries"` member of the geometry). Each array representing a ring therefore has one more value than that to store its vertices.

The depth of the array depends on the Geometry object, and is equal to the depth of the `"boundary"` array.

In the following, the Solid has 4 surfaces, and there are 2 themes: "winter-textures" and "summer-textures" could for instance represent the textures during winter and summer.. Notice that the last 2 surfaces of the first theme gets no material, thus the value `null` is used.

```
{
  "type": "Solid",
  "lod": "2.2",
  "boundaries": [
    [ [[0, 3, 2, 1]], [[4, 5, 6, 7]], [[0, 1, 5, 4]], [[1, 2, 6, 5]] ]
  ],
  "texture": {
    "winter-textures": {
      "values": [
        [ [[0, 10, 23, 22, 21]], [[0, 1, 2, 6, 5]], [[null]], [[null]] ]
      ]
    },
    "summer-textures": {
      "values": [
        [
          [[1, 10, 23, 22, 21]],
          [[1, 1, 2, 6, 5]],
          [[1, 66, 12, 64, 5]],
          [[2, 99, 21, 16, 25]]
        ]

      ]
    }
  }
}
```

§ 6.3. Material Object

A Material Object:

- **must** have one member with the name `"name"`, whose value is a string identifying the material.
- **may** have the following members (their meaning is explained there):

    1. `"ambientIntensity"`. The value is a number between 0.0 and 1.0.
    2. `"diffuseColor"`. The value is an array with 3 numbers between 0.0 and 1.0 (RGB colour).
    3. `"emissiveColor"`. The value is an array with 3 numbers between 0.0 and 1.0 (RGB colour).

4. `"specularColor"`. The value is an array with 3 numbers between 0.0 and 1.0 (RGB colour).

5. `"shininess"`. The whose value is a number between 0.0 and 1.0.

6. `"transparency"`. The value is a number between 0.0 and 1.0 (1.0 being completely transparent).

7. `"isSmooth"`. The value is a Boolean value, is defined in CityGML as a hint for normal interpolation. If this boolean flag is set to true, vertex normals should be used for shading (Gouraud shading). Otherwise, normals should be constant for a surface patch (flat shading).

> If only `"name"` is defined for the Material Object, then it is up to the application that reads the CityJSON file to attach a material definition to the `"name"`. This might not always be possible. Therefore, it is advised to define as many from the optional members as needed for fully displaying the material.

```
"materials": [
  {
    "name": "roofandground",
    "ambientIntensity":  0.2000,
    "diffuseColor":  [0.9000, 0.1000, 0.7500],
    "emissiveColor": [0.9000, 0.1000, 0.7500],
    "specularColor": [0.9000, 0.1000, 0.7500],
    "shininess": 0.2,
    "transparency": 0.5,
    "isSmooth": false
  },
  {
    "name": "wall",
    "ambientIntensity":  0.4000,
    "diffuseColor":  [0.1000, 0.1000, 0.9000],
    "emissiveColor": [0.1000, 0.1000, 0.9000],
    "specularColor": [0.9000, 0.1000, 0.7500],
    "shininess": 0.0,
    "transparency": 0.5,
    "isSmooth": true
  }
]
```

§ 6.4. Texture Object

A Texture Object:

- **must** have one member with the name `"type"`. The value is a string with either "PNG" or "JPG" as value.
- **must** have one member with the name `"image"`. The value is a string with the name of the file. This file can be a URL (eg `"http://www.someurl.org/filename.jpg"`), a relative path (eg `"appearances/myroof.jpg"`), or an absolute path (eg `"/home/elvis/mycityjson/appearances/myroof.jpg"`).
- **may** have one member with the name `"wrapMode"`. The value can be any of the following: `"none"`, `"wrap"`, `"mirror"`, `"clamp"`, or `"border"`.
- **may** have one member with the name `"textureType"`. The value can be any of the following: `"unknown"`, `"specific"`, or `"typical"`.
- **may** have one member with the name `"borderColor"`. The value is an array with 4 numbers between 0.0 and 1.0 (RGBA colour).

```
"textures": [
  {
    "type": "PNG",
    "image": "http://www.someurl.org/filename.jpg"
  },
  {
    "type": "JPG",
    "image": "appearances/myroof.jpg",
    "wrapMode": "wrap",
    "textureType": "unknown",
    "borderColor": [0.0, 0.1, 0.2, 1.0]
  }
]
```

§ 6.5. Vertices-texture Object

An Appearance Object may have one member named `"vertices-texture"`. The value is an array of the *(u,v)* coordinates of the vertices used for texturing surfaces. Their position in this array (0-based) is used by the `"texture"` member of the Geometry Objects.

- the array of vertices may be empty.
- one vertex must be an array with exactly 2 values, representing the *(u,v)* coordinates.

- The value of u and v must be between 0.0 and 1.0.
- vertices may be repeated

```
"vertices-texture": [
  [0.0, 0.5],
  [1.0, 0.0],
  [1.0, 1.0],
  [0.0, 1.0]
]
```

§ 7. Handling large files

Because CityJSON aims at being easy-to-use and developers friendly, it is advised to keep the size of CityJSON files small. Files of several hundreds of megabytes are bad practice, and should be avoided since users will have great difficulties visualising and manipulating them.

§ 7.1. Decomposing an area into parts/tiles

One solution to handle a large dataset is to subdivide it into tiles or regions, and ensure that each part has a reasonable size. Each part becomes a CityJSON file.

§ 7.2. Text sequences and streaming with CityJSONFeature

Another solution is to decompose a CityJSON object into its *features* (the City Objects), create several JSON objects, and store them in a JSON Lines text (also called Newline Delimited JSON). This is a format to store several JSON objects in a single file, and allows the processing of each object one at a time.

A CityJSON Feature Object allows storage of one feature, for instance a `"Building"` with eventually its children `"BuildingPart"` and/or `"BuildingInstallation"`. Unlike a CityJSON Object, all the vertices and appearances of the object are *local*.

A CityJSON Feature Object:

- is a JSON object.
- **must** have one member with the name `"type"`. The value must be `"CityJSONFeature"`.

- **must** have one member with the name `"id"`. The value must be a string representing the identifier of the City Object Feature, this is used to clearly identify which of the CityObjects is the parent.
- **must** have one member with the name `"CityObjects"`. The value of this member is a collection of key-value pairs, where the key is the ID of the object, and the value is one City Object. The ID of a City Object should be unique (within one `"CityJSONFeature"`), and all the children of the `"CityJSONFeature"` must be included (and the children of the children (recursively), if there are any).
- **must** have one member with the name `"vertices"`. The value is an array of coordinates of each vertex of the current City Object Feature (stored with integers). Their position in this array (0-based) is used as an index to be referenced by the Geometry Objects for the JSON object (warning: the vertices are local to the JSON object).
- **may** have one member with the name `"appearance"`. The value may contain JSON objects representing the textures and/or materials of surfaces. See § 6 Appearance Object for details.
- **must not** have other members.

```
{
  "type": "CityJSONFeature",
  "id": "myid",
  "CityObjects": {},
  "vertices": [],
  "appearance": {}
}
```

```
{
  "type": "CityJSONFeature",
  "id": "id-1",
  "CityObjects": {
    "id-1": {
      "type": "Building",
      "attributes": {
        "roofType": "gabled roof"
      },
      "children": ["mypart"],
      "geometry": [...]
    },
    "mypart": {
      "type": "BuildingPart",
      "parents": ["id-1"],
```

```
      "children": ["mybalcony"],
      "geometry": [...]
    },
    "mybalcony": {
      "type": "BuildingInstallation",
      "parents": ["mypart"],
      "geometry": [...]
    }
  },
  "vertices": [...]
}
```

The following root property of a CityJSON Object are not allowed in a CityJSONFeature Object:

- `"transform"`
- `"version"`
- `"metadata"`
- `"geometry-templates"`: these should be resolved/dereferenced
- `"extensions"`: these should be in the metadata or collection

Note that a CityJSON Feature Object does not contain all the information that is required for parsing the feature. Most commonly, the transformation properties (the Transform Object) and CRS need to be known by the client in order to correctly georeference the City Objects. These properties may be known by the client upfront, or they may be accessible in a CityJSON Object, which is sent as the first object in a JSON Lines text stream, or in other ways not described here.

In case the properties are stored in a CityJSON Object, this object needs to be a valid CityJSON Object. This implies that the CityJSON object must contain all the required properties, including `"CityObjects"` and `"vertices"`, even though they are empty, because this information is stored in the subsequent CityJSON Features.

Below is an example of a CityJSONFeature stream (or a JSON Lines text file), with a CityJSON Object storing the metadata and transformation properties:

```
{"type":"CityJSON","version":"1.1","transform":{...},"CityObjects":{},"meta
{"type":"CityJSONFeature","id":"a","CityObjects":{...},"vertices":[...]}
{"type":"CityJSONFeature","id":"b","CityObjects":{...},"vertices":[...]}
{"type":"CityJSONFeature","id":"c","CityObjects":{...},"vertices":[...]}
```

> **Suggested convention:** `"CityJSON"` and `"CityJSONFeature"` objects may be stored in a file with the extension `'.city.jsonl'`

> Observe that CityJSON does not prescribe the format or standard that should be used to store several JSON objects in a given file, it only defines how `"CityJSON"` and `"CityJSONFeature"` objects should be defined.

§ 8. Extensions

CityJSON uses JSON Schemas to document and validate its data model, including its Extensions. Schemas offer a way to validate the syntax of a JSON document, and thus the possibility to require certain JSON members. Therefore, for writing more complex Extensions, a basic familiarity with JSON Schemas is advised.

A CityJSON *Extension* is a JSON file that documents how the core data model of CityJSON is extended, and is also used for validating the CityJSON files. This is conceptually akin to the Application Domain Extensions (ADEs) in CityGML.

A CityJSON Extension can extend the core data model in four ways:

1. Defining new properties at the root of a document
2. Defining attributes on existing City Objects
3. Defining a new Semantic Object
4. Defining a new City Object, or "extending" one of the existing City Objects

> While Extensions are less flexible than CityGML ADEs (inheritance and namespaces are for instance not supported, and less customisation is possible), it should be noted that the flexibility of ADEs comes at a price: the software processing an extended CityGML file will not necessarily know what structure to expect.
> There is ongoing work to use the ADE schemas to automatically do this, but this currently is not supported by most software. Viewers might not be affected by ADEs because the geometries are usually not changed by an ADE (although they can!). However, software parsing the XML to extract attributes and features might not work directly (and thus specific code would need to be written).
>
> CityJSON Extensions are designed such that they can be read and processed by standard CityJSON software, often no changes in the parsing code is required. This is achieved by enforcing a set of 6 simple rules (see § 8.7 Rules to follow to define new City Objects) when adding new City Objects. If these are followed, then a CityJSON file containing Extensions will be seen as a "standard" CityJSON file.

One of the philosophies of JSON is "schema-less", which means that one is allowed to define new properties for the JSON objects without documenting them in a JSON schema (watch out: this does *not* mean that JSON does not have schemas!). While this is in contrast to CityGML (and GML as a whole) where the schemas are central, the schemas of CityJSON are (partly) following that philosophy.

If one wants to document the parcel area in square-meters for a `"Building"` (`"area-parcel": {"value": 437, "uom": "m2"}`), the easiest way is just to add a new property to the City Object attributes:

```
{
  "type": "Building",
  "attributes": {
    "storeysAboveGround": 2,
    "area-parcel": {
      "value": 437,
      "uom": "m2"
    }
  },
  "geometry": [...]
}
```

However, a regular attribute (without the `"+"` prefix) cannot be made mandatory in the core CityJSON schema. Only with an Extension can an attribute be made mandatory (see § 8.4 Case 2: Defining attrib-

utes on existing City Objects).

Therefore, an *Extension* is used for enforcing certain properties, attributes, or City Object types in CityJSON objects. An *Extension* makes sense if it is expected that different data producers and consumers in the target domain need to exchange data, or if an additional City Object or Semantic type is required for accurately modelling the data.

§ 8.1. Using an Extension in a CityJSON file

An Extension should be given a name (eg "Noise") and the URL of the Extension file should be defined, including the version of the Extension that is used for this file. It is expected that the Extension is publicly available at the URL, and can be downloaded.

Several Extensions can be used in a single CityJSON Object, each one is indexed by its name in the `"extensions"` JSON object. In the example below we have two Extensions: one named "Noise" and one named "Solar_Potential".

```
{
  "type": "CityJSON",
  "version": "1.1",
  "extensions": {
    "Noise": {
      "url" : "https://someurl.org/noise.json",
      "version": "2.0"
    },
    "Solar_Potential": {
      "url" : "https://someurl.org/solar.json",
      "version": "0.8"
    }
  },
  "CityObjects": {},
  "vertices": []
}
```

§ 8.2. The Extension file

A CityJSON Extension is a JSON object, and it **must** have the following 8 members:

1. one member with the name `"type"`. The value must be `"CityJSONExtension"`.

2. one member with the name `"name"`. The value must be a string identifying the extension.
3. one member with the name `"url"`. The value must be a string with the HTTP URL of the location of the schema where the JSON object is located.
4. one member with the name `"version"`. The value must be a string identifying the version of the Extension.
5. one member with the name `"versionCityJSON"`. The value must be a string (X.Y) identifying the version of CityJSON that uses the Extension.
6. one member with the name `"extraRootProperties"`. The value must be a JSON object. Its content is part of a JSON schema (explained below), or an empty object.
7. one member with the name `"extraAttributes"`. The value must be a JSON object. Its content is part of a JSON schema (explained below), or an empty object.
8. one member with the name `"extraCityObjects"`. The value must be a JSON object. Its content is part of a JSON schema (explained below), or an empty object.

```
{
  "type": "CityJSONExtension",
  "name": "Noise",
  "description": "Extension to model the noise",
  "url": "https://someurl.org/noise.ext.json",
  "version": "0.5",
  "versionCityJSON": "1.1",
  "extraRootProperties": {},
  "extraAttributes": {},
  "extraCityObjects": {}
}
```

If an element of the Extension reuses, or references, structures and/or objects defined in the schemas of CityJSON, then assume that the Extension is in the same folder as the schemas. An example would be to reuse the Solid type:

```
"items": {
  "oneOf": [
    {"$ref": "geomprimitives.json#/Solid"}
  ]
}
```

§ 8.3. Case 1: Adding new properties at the root of a document

It is allowed to add a new property at the root of a CityJSON file, but if one wants to document it in a schema, then this property must start with a `"+"`. Imagine we wanted to store some census data for a given neighbourhood for which we have a CityJSON file, then we could define the extra root property `"+census"` as follows:

```
"extraRootProperties": {
  "+census": {
    "type": "object",
    "properties": {
      "percent_men": {
        "type": "number",
        "minimum": 0.0,
        "maximum": 100.0
      },
      "percent_women": {
        "type": "number",
        "minimum": 0.0,
        "maximum": 100.0
      }
    }
  }
}
```

And a CityJSON file would look like this:

```
{
  "type": "CityJSON",
  "version": "1.1",
  "extensions": {
    "Census": {
      "url": "https://someurl.org/census.ext.json",
      "version": "0.7"
    }
  },
  "CityObjects": {...},
  "vertices": [...],
  "+census": {
    "percent_men": 49.5,
```

```
    "percent_women": 51.5
  }
}
```

§ 8.4. Case 2: Defining attributes on existing City Objects

It is also possible to add, and document in a schema, specific attributes, for example if we wanted to have the colour of the buildings as a RGBA value (red-green-blue-alpha):

```
{
  "type": "Building",
  "attributes": {
    "storeysAboveGround": 2,
    "+colour": {
      "rgba": [255, 255, 255, 1]
    }
  },
  "geometry": [...]
}
```

Another example would be to store the area of the parcel of a building, and also to document the unit of measurement (UoM):

```
{
  "type": "Building",
  "attributes": {
    "storeysAboveGround": 2,
    "+area-parcel": {
      "value": 437,
      "uom": "m2"
    }
  },
  "geometry": [...]
}
```

For these two cases, the CityJSON Extension object would look like the snippet below. Notice that `"extraAttributes"` may have several properties (the types of the City Objects are the possibilities) and then each of these has as properties the new attributes (there can be several).

An extra attribute must start with a "+"; it is good practice to prepend the attribute with the name of the Extension, to avoid that 2 attributes from 2 different Extensions have the same name.

The value of the property is a JSON schema, this schema can reference and reuse JSON objects already defined in the CityJSON schemas. Thus, the keywords of the property values are defined by the JSON Schema specification. For instance, "additionalProperties" is a JSON-schema keyword stating that one is not allowed to add properties to this JSON object, beyond the ones defined in the schema (eg "value", "uom").

```
"extraAttributes": {
  "Building": {
    "+colour": {
      "type": "object",
      "properties": {
        "rgba": {
          "type": "array",
          "items": {"type": "number"},
          "minItems": 4,
          "maxItems": 4
        }
      },
      "required": ["rgba"],
      "additionalProperties": false
    },
    "+area-parcel": {
      "type": "object",
      "properties": {
        "value": { "type": "number" },
        "uom": { "type": "string", "enum": ["m2", "feet2"] }
      },
      "required": ["value", "uom"],
      "additionalProperties": false
    }
  }
}
```

§ 8.5. Case 3: Defining a new Semantic Object

To define a new semantic surface (besides the ones prescribed, see §3.3 Semantics of geometric primitives), a "+" must be prepended to its name, eg "+ThermalSurface".

```
{
  "type": "+ThermalSurface"
}
```

§ 8.6. Case 4: Creating and/or extending new City Objects

The creation of a new City Object is done by defining it in the CityJSON Extension object in the `"extraCityObjects"` property:

```
"extraCityObjects": {
  "+NoiseBuilding": {
    "allOf": [
      { "$ref": "cityobjects.json#/_AbstractBuilding" },
      {
        "properties": {
          "type": { "enum": ["+NoiseBuilding"] },
          "attributes": {
            "properties": {
              "buildingLDenMin": {"type": "number"}
            }
          }
        },
        "required": ["type"]
      }
    ]
  }
}
```

```
"extraCityObjects": {
  "+NoiseBuildingPart": {
    "allOf": [
      { "$ref": "cityobjects.json#/_AbstractBuilding" },
      {
        "properties": {
          "type": { "enum": ["+NoiseBuildingPart"] },
          "attributes": {
            "properties": {
              "buildingLDenMin": {"type": "number"}
            }
```

```
            }
          },
          "required": ["type", "parents"]
        }
      ]
    }
  }
```

Since all City Objects are documented in the schemas of CityJSON (in `cityobjects.schema.json`), it is basically a matter of copying the parts needed in a new file and modifying its content.

A new name for the City Object must be given and it must begin with a `"+"`.

Because City Objects can be of different levels (1st-level ones can exist by themselves; 2nd-level ones need to have a parent), we need to explicitly define that `"parents"` is mandatory for 2nd-level objects.

Please note that since JSON schemas do not allow inheritance, the only way to extend a City Object is to define an entirely new one (with a new name, eg `"+NoiseBuilding"`). This is done by copying the schema of the parent City Object and extending it.

§ 8.7. Rules to follow to define new City Objects

The challenge when creating Extensions to the core model is that we do not want to break the software packages (viewers, spatial analysis, etc) that already read and process CityJSON files. While one could define a new City Object and document it, if this new object does not follow the rules below then it will mean that new specific software needs to be built for it (and this would go against the fundamental ideas behind CityJSON).

1. The name of a new City Object must begin with a `"+"`, eg `"+NoiseBuilding"`.
2. A new City Object must conform to the rules of CityJSON, ie it must contain a property `"type"`.
3. Existing City Objects cannot be extended and have new types as children, eg it is not allowed to add a new City Object `"+Balcony"` to a `"Building"`. Instead, a new type, eg `"+FancyBuilding"`, should be created and it can have a `"+Balcony"` as a potential children.
4. All the geometries must be in the property `"geometry"`, and cannot be located somewhere else deep in a hierarchy of a new property.

5. The Geometry object's boundary must be one of the eight types described in § 3 Geometry Objects. Similarly, the geometry appearances and templates must follow the core specification. This ensures that all the code written to process, manipulate, and view CityJSON files will be working without modifications.
6. The reuse of types defined in CityJSON, eg `"Solid"` or semantic surfaces, is allowed.

§ 9. CityJSON Schemas

The JSON schemas of the specifications are publicly available at https://cityjson.org/schemas/.

§ 10. CityGML v3.0 conformance

CityJSON v1.1 is conformant with the CityGML v3.0 data model, although not all extension modules have been implemented.

The details of which modules are supported (and where the so-called *null mapping* are applied, see CityGML Modularization), are available at https://www.cityjson.org/conformance/v30/