Semantic 3D city models | Lesson 4.1

A 3D city model is a digital representation, with three-dimensional geometries, of the common objects in an urban environment, with buildings usually being the most prominent objects.

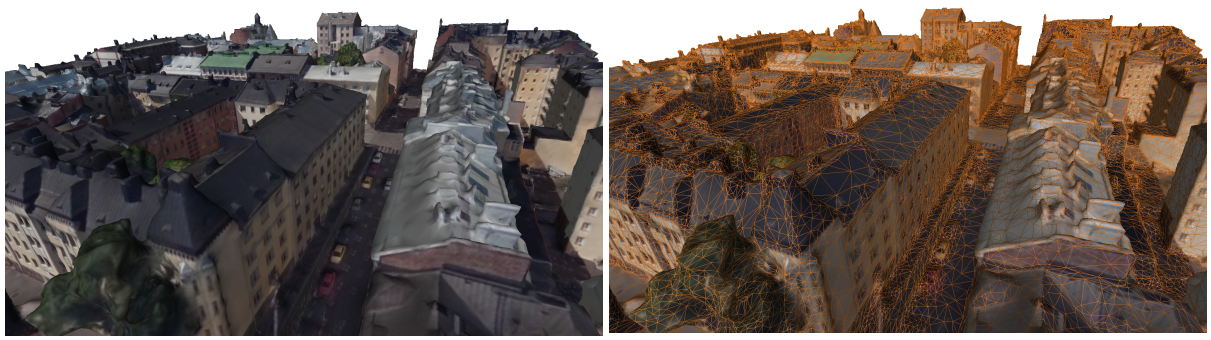
Because typical 3D city models are reconstructed/derived from various acquisition techniques, their structure, format, and characteristic will greatly vary. As an example, a 3D city model can be reconstructed with methods such as these: photogrammetry, laser scanning, extrusion from 2D footprints, conversion from architectural models and drawings, procedural modelling, volunteered geoinformation, etc.

This lesson discusses the main 3D city models formats, and focuses on *semantic* 3D city models, which are useful in a variety of applications.

1.1 Semantic 3D city models

textured mesh

Consider the 3D city model of Helsinki in Figure 1.1a (one part of it), which was reconstructed by dense matching of aerial images. The model is a textured mesh, formed by triangles to which a texture is attached (the triangles are visible in in Figure 1.1b). If you were asked to count the number of buildings (or cars, or dormers in a given building) you would surely just have to zoom in on the model, look at it, and then you could give the answer. However, for a computer, this 3D city model is simply represented as a series of triangles to which a texture is attached; the notion of 'building' (or 'car', or any other object) is thus not available. As a result, a computer cannot automatically answer these simple questions. It should be observed that there exist algorithms to segment and classify textured meshes into objects, but these are not fully automatic and are beyond the scope of this course. Other simple questions that a human could answer but a computer cannot:



(a) **(b)** the edges of the triangles are highlighted in orange.

Figure 1.1: Part of the 3D city model of Helsinki, Finland.

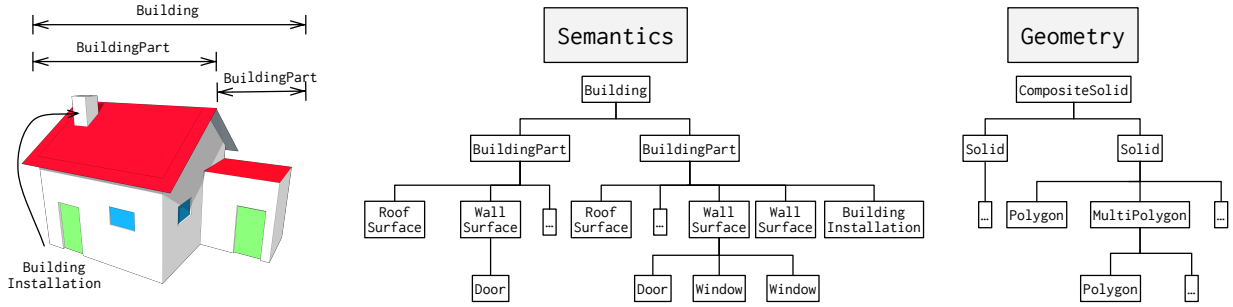


Figure 1.2: A building is semantically decomposed into different objects, and each objects is defined with geometry.

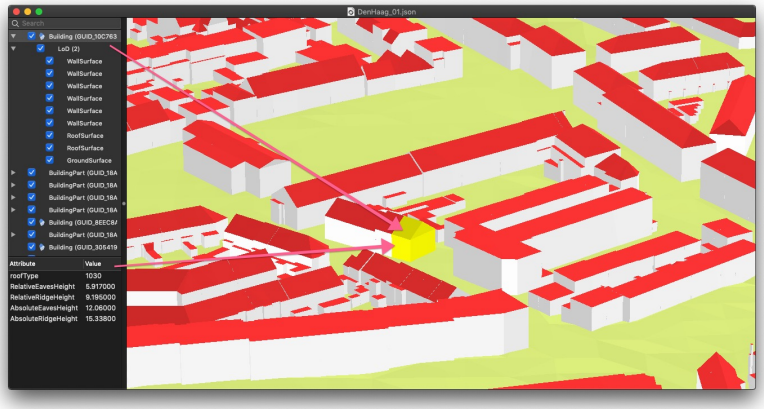


Figure 1.3: Part of the semantic 3D city model of The Hague, in the Netherlands. Notice that each building is decomposed into its semantic surfaces (wall, roof, and ground) and there are attributes for each. The model is not textured, but semantic models can have textures too.

1. how many windows does the main façade of a given building have?
2. how many floors does a given building have?
3. can the local park be seen from the second floor of a given building?

A semantic 3D city model is a data model where the *relevant* objects (and their sub-parts) are labelled with their meaning and have attributes attached to them. Conceptually, it means that a city is decomposed into classes that we deem relevant for certain applications, for instance the city is decomposed into the classes 'building', 'road', 'tree', 'lamppost', etc and each of the objects has its own 3D geometry and potentially (thematic) attributes (eg the owner of a building, the name of street, the city identifier for a lamppost, etc).

Observe also, as shown in Figure 1.2 for one building, that the objects can be further decomposed into semantically homogeneous parts, in 3D city modelling these are often the parts of a buildings (eg an extension to a house) and the type of surfaces (roof, façade, windows, doors).

The decomposition is thus hierarchical, and the relationships between the classes are stored (eg a building is composed of parts, which are formed of walls, which have windows). We say that a 3D city model is *spatially coherent* is the two decompositions are coherent.

spatially coherent

Figure 1.3 shows one semantic model being visualised in a viewer, notice that the user can identify the roof surfaces and that different attributes are available. Semantic 3D models can also be textured.

To avoid the fact that every city/country defines its own classes to decompose a city (eg a 'building' class can be a 'house' class in another

city), semantic models prescribe the classes and often even the thematic attributes that should be stored.

1.2 The CityGML data model

CityGML is an open data model to represent semantic 3D models of cities and landscapes, and it is standardised by the Open Geospatial Consortium; its official website is `https://www.opengeospatial.org/standards/citygml`. Its first version (v1.0.0) was released in 2008, and the current version (v2.0.0) in 2012; the upcoming version (v3.0) has been in draft for years, and will be released eventually.

The classes stored in CityGML are grouped into different modules. These are:

- **Appearance**: textures and materials for other types
- **Bridge**: bridge-related structures, possibly split into parts
- **Building**: the exterior and possibly the interior of buildings with individual surfaces that represent doors, windows, etc
- **CityFurniture**: benches, traffic lights, signs, etc
- **CityObjectGroup**: groups of objects (any types)
- **Generics**: other types that are not explicitly covered
- **LandUse**: areas that reflect different land uses, such as urban, agricultural, etc
- **Relief**: the shape of the terrain
- **Transportation**: roads, railways and squares
- **Tunnel**: tunnels, possibly split into parts
- **Vegetation**: areas with vegetation or individual trees
- **WaterBody**: lakes, rivers, canals, etc

Figure 1.4 shows one part of the CityGML UML models.

1.2.1 Levels-of-detail (LoDs)

One particularity of CityGML is that it prescribes the different standard levels of detail (LoDs) for 3D objects, which allows us to represent objects for different applications and purposes.

For each of the classes defined by CityGML, five LoDs can be defined. Figure 1.5 shows the ones for the buildings, and they are as follows:

LoD0 is a horizontal polygon representing the footprint (at the elevation of the terrain) and optionally a horizontal polygon representing the horizontal roof. Such models represent the transition from 2D to 3D GIS, and they do not contain volumetric geometries.

LoD1 is a block model, with an horizontal and planar roof that is usually derived by extruding a footprint to a given height. LoD1 models are easy to reconstruct: the footprint of a building, readily available in many countries, can be extruded to its height. The height can be the average (or median) of all the lidar points inside the footprint.

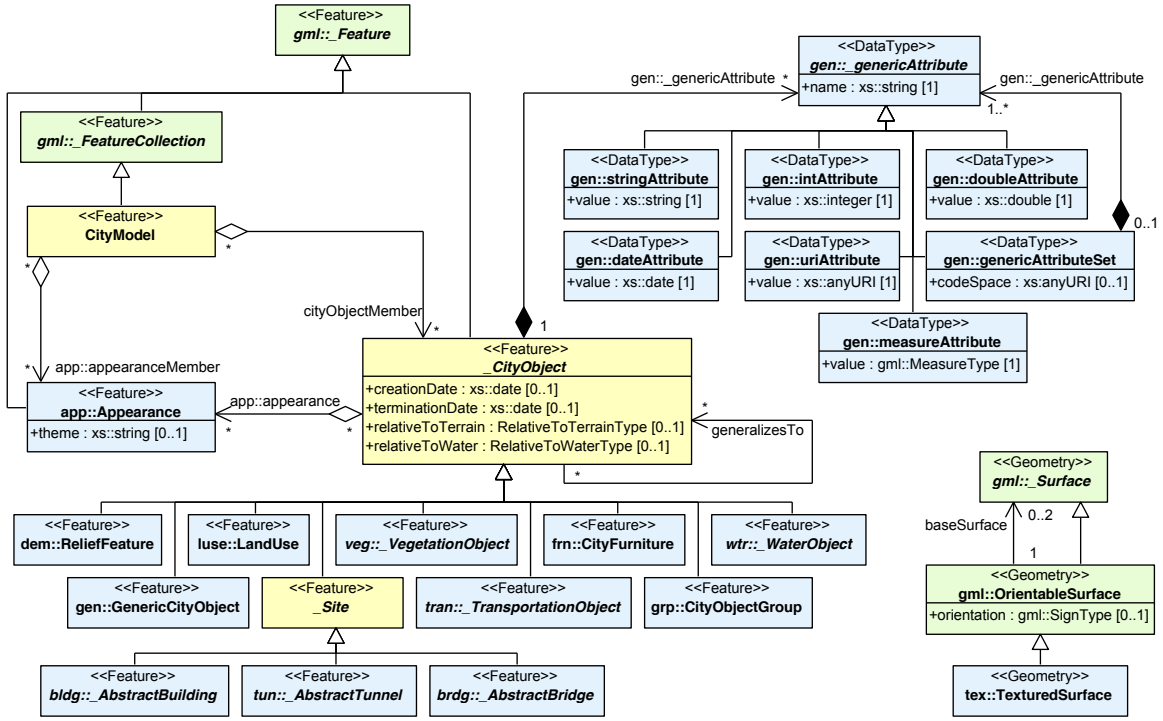


Figure 1.4: Overview of the UML model for the core of CityGML. (Figure © 2012 Open Geospatial Consortium, Inc.)

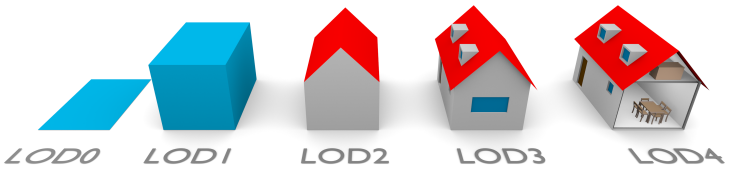


Figure 1.5: The five LODs of CityGML 2.0. The geometric details and the semantic complexity increase, ending with the LOD4 containing indoor features.

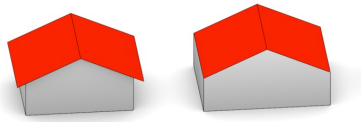


Figure 1.6: Two buildings represented in CityGML as LoD2 models. Both are valid LoD2 models.

LoD2 the generalised roof shape and larger roof superstructures are present. As such, LoD2 models are useful for rooftop solar potential estimations. They are usually obtained with photogrammetric techniques, and, in some cases, may be derived automatically (see Lesson 2.1).

LoD3 is a detailed architectural model containing openings (windows and doors), chimneys, and other façade details. Models at LoD3 are usually obtained with a conversion from BIM models or from terrestrial laser scanning. The presence of windows and other details makes them useful in applications such as energy simulations.

LoD4 is an LoD3 model containing indoor features such as rooms and furniture. LoD4 marks the boundary between GIS and BIM. They can only be constructed by converting BIM datasets (or CAD) to CityGML, and are in practice basically never used (since BIM software are better at processing indoor models than GIS software).

While the five LoDs are supposed to inform users about the representation of the data, in practice they are too generic (not precise enough) and can be ambiguous. For instance, as Figure 1.6 shows, a building with roof overhangs can be modelled as LoD2 with them, or without (and therefore the size of its footprint would be larger). Both are technically "valid" LoD2 models, but the acquisition methods required differ significantly. The model on the right can be acquired with aerial photogrammetry or aerial

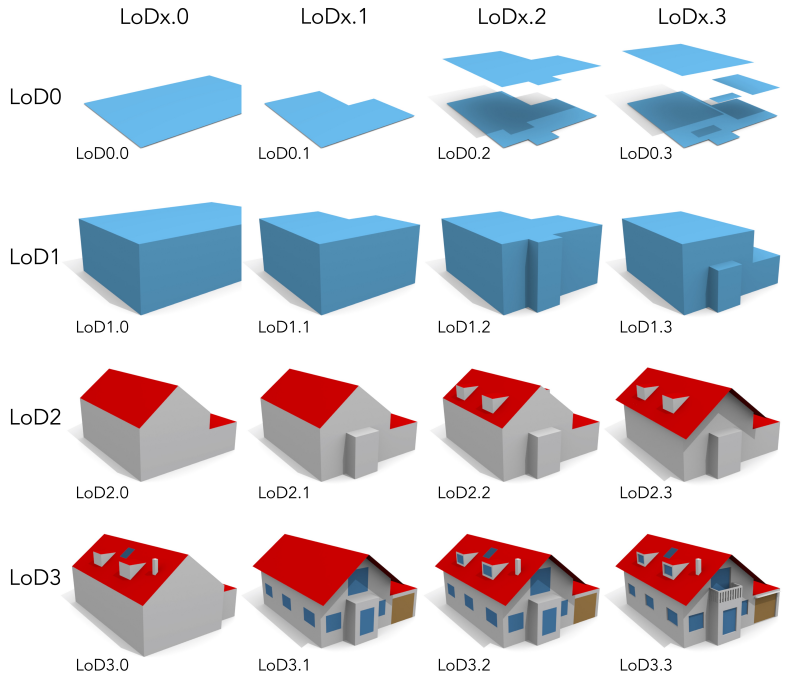


Figure 1.7: The improved LoDs for buildings; they are generally referred to as the *TUDelft LoDs*.

lidar (the walls are derived as projections from the roof outline), while the model on the left probably needs two acquisition techniques: the walls are at their actual location (ground survey was necessary) and the roof overhangs are explicitly present. To remedy to this situation, improved LoDs for buildings have been proposed at TU Delft, see Figure 1.7.

Notice that while each of the CityGML classes can be represented with 5 different LoDs, only those for buildings are prescribed and documented. For trees and roads, practitioners can decide that a given representation is 'LoD2', but that would purely indicate that the LoD is higher than a LoD1 one. There are efforts (scientific papers) to document these, but they have not been standardised yet.

1.2.2 Geometries

ISO 19107

CityGML uses the ISO 19107 geometric primitives for representing the geometry of its 3D objects. While the ISO 19107 primitives do not need to be linear or planar, ie curves defined by mathematical functions are allowed, CityGML uses a subset of ISO 19107, with the following two restrictions: (1) `GM_Curves` can only be *linear* (thus only `LineStrings` and `LinearRings` are used); (2) `GM_Surfaces` can only be *planar* (thus `Polygons` are used).

See Lesson 3.2 for the details of ISO 19107.

1.2.3 Textures and materials

The 3D geometries can be supplemented with textures and/or colours (called materials since different parameters like transparency can be defined) to give a better impression of their appearance.

CityGML reuses known and used standards in other fields for the appearances. The material is represented with the X3D specifications, and the texture with the COLLADA standard.

1.2.4 Extensions to the core data model with ADEs

The CityGML data model prescribes a certain number of classes, but sometimes practitioners may want to model additional objects. For this, CityGML has the concept of ADEs (application domain extensions). An ADE is defined as an extension/extra to the core data model, inheritance is used to refine the classes of CityGML (add attributes for instance) or to define entirely new classes.

CityGML has XML files and the schemas can be extended, see Section 1.3 for more details.

CityJSON has a similar mechanism, see below.

1.2.5 Encodings

Based on the CityGML data model, there exist three encodings:

1. XML/GML-based encoding, also called "CityGML"
2. CityJSON
3. a database schema called 3DCityDB, which can be implemented both for PostgreSQL and Oracle Spatial. This is not an official standard. Details at `https://www.3dcitydb.org/3dcitydb/`

We discuss in the following the first two.

1.3 The XML encoding of CityGML

So far, the only "approved" encoding of the CityGML data model is the XML-based encoding described in the official standard specification. Notice that "CityGML" refers to both the data model and the XML encoding, which can admittedly be very confusing.

GML specifications: `https://www.opengeospatial.org/standards/gml`

CityGML is actually an application schema of GML, which is the *Geography Markup Language*, also standardised by the OGC. This can be observed in the UML diagram in Figure 1.4: a city object (class `_CityObject`) inherits from the generic `gml::_Feature` in GML, which is the parent class for all geographic features.

As shown in Figure 1.8, CityGML datasets consist of a set of plain text files (XML files) and possibly some accompanying image files that are used as textures. Each text file can represent a part of the dataset, such as a specific region, objects of a specific type (such as a set of roads), or a predefined LoD. The structure of a CityGML file is a hierarchy that ultimately reaches down to individual objects and their attributes.

Because CityGML files are XML files, they can be parsed by any XML-parser (there are many available), and also can be modified with a text editor.

```
<?xml version="1.0" encoding="UTF-8"?>
<CityModel xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:gml="http://www.opengis.net/gml"
  xmlns:gen="http://www.opengis.net/citygml/generics/1.0"
  xmlns="http://www.opengis.net/citygml/1.0"
  xsi:schemaLocation="http://www.opengis.net/citygml/1.0">
  <cityObjectMember>
  <bldg:Building gml:id="9a06451677c7">
    <bldg:function>1070</bldg:function>
    <bldg:lod1Solid>
      <gml:Solid>
        <gml:exterior>
          <gml:CompositeSurface>
            <gml:surfaceMember>
              <gml:Polygon>
                <gml:exterior>
                  <gml:LinearRing>
                    <gml:pos>0.0 0.0 0.0</gml:pos>
                    <gml:pos>0.0 1.0 0.0</gml:pos>
                    <gml:pos>1.0 1.0 0.0</gml:pos>
                    <gml:pos>1.0 0.0 0.0</gml:pos>
                    <gml:pos>0.0 0.0 0.0</gml:pos>
                  </gml:LinearRing>
                </gml:exterior>
              </gml:Polygon>
            </gml:surfaceMember>
    ...
    </bldg:Building>
    <bldg:Building gml:id="jdhd76sa">
    ...
    </bldg:Building>
  </cityObjectMember>
</CityModel>
```

Figure 1.8: Part of a CityGML file containing 2 buildings.

The schema of CityGML is encoded in XML files called "XSD" (XML Schema Definition). This way, software can validate whether the syntax of a file corresponds to that of the data model, for instance it can defined that a Building must have a geometry, and that a set of attributes are mandatory.

CityGML ADEs. When distributing files containing ADEs, usually the extensions to the data model must be made available; with XML-based CityGML files those are XML Schema files (`.xsd` files). City data contained in a CityGML file can be objects from the core model (eg buildings) and new objects defined in an ADE (eg sheds could be defined).

1.3.1 The drawback of the XML encoding

The vast majority of the efforts concerning CityGML have been spent on developing the concepts and the data model, and it appears that very little attention has been paid to deriving a *usable* exchange format. Indeed, the XML encoding is verbose, hierarchical, complex, and not adapted for the web. These drawbacks hinder the use of CityGML in practice, which can be observed by: (1) the low number of software packages supporting full read/write/edit capabilities for CityGML files; and (2) the relatively low number of datasets stored in CityGML files.

CityGML files are notoriously known to be very difficult to parse and to extract information from. This has to do with the fact that XML itself requires special libraries to handle the data, that GML has several different

ways to store the same geometry*, and that CityGML files have deep hierarchies (which are problematic for DBMS implementation, which tend to be 'flat') and several XLinks.

1.4 CityJSON

JSON: JavaScript Object Notation: http://json.org

CityJSON (currently at version 1.0), is a JSON encoding of the CityGML 2.0.0 data model. JSON is, like GML, a text-based data exchange format that can be read both by humans and machines.

It has a number of advantages, over GML, for several reasons. First, and foremost, JSON dominates the web: nowadays if two applications need to exchange data they will most likely use JSON (over XML). Of the ten most popular APIs on the web, only one exposes its data in XML, the others all use JSON†. Second, JSON is predominantly favoured by developers (on *Stack Overflow* it is by far the most discussed exchange format) which means that more libraries and software will support it, and these will most likely be maintained. Finally, JSON is based on two data structures that are available in virtually every programming language (more details below), and we can thus structure a file in a way that developers would build and index in memory the objects (developers then do not need to use external libraries, all features and geometries are already indexed, and ready to use).

A CityJSON file represents a given geographical area; the file contains one JSON object of type `"CityJSON"` and would typically contain the following JSON properties:

```
{
  "type": "CityJSON",
  "version": "1.0",
  "CityObjects": {},
  "vertices": [],
  "appearance": {}
}
```

1.4.1 City objects are "flattened out"

The property `"CityObjects"` contains a JSON dictionary where the properties are the identifiers of the city objects (*IDs*). The schema of CityGML has been flattened out and all hierarchies removed. Figure 1.9 shows the city objects that are supported in CityJSON, both 1st- and 2nd-level city objects are stored in the dictionary `"CityObjects"`.

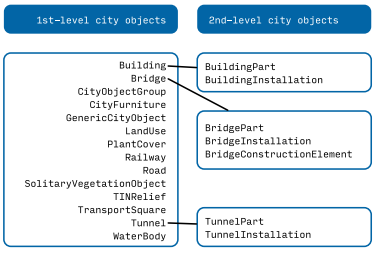


Figure 1.9: The implemented CityJSON classes (same name as CityGML classes) are divided into 1st and 2nd levels.

As an example, for a `Building` containing 2 `BuildingParts`, the 3 objects will be represented at the same level and linked by their *IDs*.

```
"CityObjects": {
  "id-1": {
    "type": "Building",
    "attributes": {...},
    "children": ["id-2", "id-3"],
```

* See the *GML Madness* blog post where 25 different ways to store a simple square in GML are described, a developer implementing a parser for CityGML would have to support them all, and more for the primitives in higher dimensions! (https://erouault.blogspot.com/2014/04/gml-madness.html)

† https://twobithistory.org/2017/09/21/the-rise-and-rise-of-json.html

```
      "geometry": [{...}]
    },
    "id-2": {
      "type": "BuildingPart",
      "parents": ["id-1"],
      "geometry": [{...}]
      ...
    },
    "id-3": {
      "type": "BuildingPart",
      "parents": ["id-1"],
      "geometry": [{...}]
      ...
    }
  }
```

Each city object can have a `"parents"` and/or a `"children"` property, and this is how in the snippet the building `"id-1"` is linked to its 2 parts. The fact that a dictionary is used means that developers have direct access to the city objects through their IDs (and also in constant time if a hash map is used to implement the dictionary).

A city object can be of any of the types defined in Figure 1.9, and each of them must have the same structure, and at a minimum contain a `"geometry"` property. If attributes are to be stored, they have to be in the `"attributes"` property. This simplifies the work of the developer because there is a single point of entry for all geometries and attributes, unlike with XML-encoded CityGML.

```
  {
    "type": "PlantCover",
    "attributes": {
      "averageHeight": 11.05,
      "colour": "green"
    },
    "geometry": [{...}]
  }
```

1.4.2 Geometry

CityJSON defines the same 3D geometric primitives used in CityGML, with the same restrictions for linearity/planarity. However, since they are rarely used in a 3D context, *Point* and *LineString* only have their Multi* counterparts; a single *Point* is a *MultiPoint* with only one object. When a geometry is defined, it must contain a value for the LoD. In order to avoid ambiguities, we encourage the use of the TUDelft LoDs (see above), over the five standard CityGML ones. City Object can have several LoDs, and thus CityJSON, as is the case for CityGML, allows us to store concurrently several LoDs for the same object.

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

It should be noticed that CityJSON uses a different approach from (City)GML to store the (x, y, z) coordinates of geometric primitives. A geometric primitive does not list all the coordinates of its vertices,

rather the coordinates of the vertices are stored in a separate array (the `"vertices"` property of the CityJSON object), and geometric primitives refer to the position of a vertex in that array.

```
"vertices": [
  [8623.234, 487111.009, 13.92],
  [8829.456, 488115.134, 10.07],
  [8554.508, 487229.995, 19.61],
  ...
  [8523.134, 487625.134, 2.03]
]
```

OBJ specifications: https://en.wikipedia.org/wiki/Wavefront_.obj_file

The indexing mechanism of the format *Wavefront OBJ* is reused, because it has been used for many years, with success, in the computer graphics community. There are several advantages to this approach. First, the files can be compressed: 3D vertices are often shared by several surfaces, and repeating them can be costly (especially if they are very precise, often sub-millimetre is used). Second, this increases the topological relationships that are explicitly stored in the file, and several operations can be sped up and made more robust (eg are two buildings adjacent?). Third, it is very easy to convert to a representation listing all coordinates; the inverse is not true.

The geometry is based on an enumeration of the vertices forming each ring of a surface, as follows. A `"MultiSurface"` has an array containing surfaces, where each surface is modelled by an array of arrays, 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 being the interior shells; each shell has an array of surfaces, modelled in the exact same way as a `"MultiSurface"`. Concrete examples of each geometric type are given at https://www.cityjson.org/help/dev/geom-arrays/. Notice that unlike with (City)GML, there is only one variation per geometry type, which (greatly) simplifies the life of developers.

```
{
  "type": "Solid",
  "lod": 2.2,
  "boundaries": [
    [ [[0, 3, 2, 1, 22]], [[4, 12, 123, 5, 6, 7]], [[0, 1, 5, 4]], [[1, 2, 6, 5]] ],
    [ [[240, 243, 124]], [[244, 246, 724]], [[34, 414, 45]], [[111, 246, 5]] ]
  ]
}
```

1.4.3 Appearance

Both textures and materials are supported, and the same mechanisms as CityGML are used for these. The material is represented with the X3D specifications, as is the case for CityGML. For the texture, the COLLADA specifications are reused, as is the case for CityGML.

X3D specifications: https://en.wikipedia.org/wiki/X3D

COLLADA specifications: https://www.khronos.org/collada/

1.4.4 Extension to the core model

CityJSON also supports extensions to the core data model of CityGML for specific applications and use-cases. They are simply called *Extensions*

and are defined as simple JSON files, and support the addition of new feature types, as well as the addition of new attributes for features and for datasets. See https://www.cityjson.org/specs/#extensions for more details.

1.4.5 CityGML support

It should be observed that, at this moment, CityJSON is not an official OGC standard. CityJSON was started, and is maintained, by the 3D geoinformation group at TU Delft. Others have since joined its development. It was developed to simplify the tasks of developers and to foster the use of the official data model in practice, but with a usable and simple-to-use encoding.

CityJSON implements most of the data model, and all the CityGML modules have been mapped to CityJSON objects. However, for the sake of simplicity and efficiency, some modules and features have been omitted and/or simplified. If a module is supported, it does not mean that there is a 1-to-1 mapping between the classes and features in CityGML and CityJSON, but rather that it is possible to represent the same information, but in a different manner. CityJSON thus conforms to a subset of CityGML, although technically only XML-encoded CityGML files can be conformant to the specifications of CityGML.

The main features that are not supported are:

- The LoD4 of CityGML, which was mostly designed to represent the interior of buildings (including details and furniture), is not implemented. The main reason is that currently there are virtually no datasets having LoD4 buildings. If there is a need in the future, the concepts and the implementation would follow the same rules described above.
- Several CRSs in the same datasets. In CityJSON, all geometries in a given CityJSON object must use the same CRS. In CityGML, 3 adjacent buildings can all have different CRSs, and some of the geometries to represent the walls can be in yet another CRS (although admittedly it is seldom used!).
- Arbitrary coordinate reference systems (CRSs). Only an EPSG code can be used.
- Identifiers for low-level geometries. In CityGML most objects can have an ID (usually `gml:id`). That is, not only can one building have an ID, but also each of the 3D primitives forming its geometry can have an ID. In CityJSON, only city objects and semantic surfaces can have IDs.
- Raster files for the relief. Only TINs are at this moment supported.

EPSG codes: https://epsg.io

1.5 Other formats for 3D city modelling

We describe briefly in this section a few formats and standards that are related to 3D city modelling and that are sometimes used in practice. Those generally focus mostly on *geometries*, but lack support for semantics and attributes (to a varying degree). They are thus usually less suitable

and less agile than the family of CityGML formats, that is they can be useful for a few use-cases.

1.5.1 Standard computer graphics formats: OBJ, PLY, OFF, etc

There exist several similar formats in computer graphics for storing and representing meshes (which are usually triangular meshes, but polygons can also be represented):

OBJ specifications: `http://paulbourke.net/dataformats/obj/`

OBJ (Wavefront Object) is one of the most popular text-based formats in the 3D graphics community. It has a simple structure where first the vertices are listed, and then each polygon is listed, as a list of references to the vertex ID (its position in the list of vertices). The OBJ format can also encode colours and texture information, which are stored in a separate file (a `.mtl` file, Material Template Library). Attributes for specific polygons or groups of polygons is only possible by using the comments and grouping possibilities (as a hack), there are no standardised and documented ways to do so.

OFF specifications: `https://en.wikipedia.org/wiki/OFF_(file_format)`

OFF is a simpler format: only polygons can be represented, optionally with their colours.

PLY is based on the same ideas for the geometries, and attributes can also be attached to vertices and polygons. (See the *Computational modelling of terrains* book Section 12.1).

Notice that neither of these formats allow us to store an ISO 19107 solid having inner shells and attributes/semantics for different parts/elements.

1.5.2 glTF (GL Transmission Format)

glTF specifications: `https://www.khronos.org/gltf/`

glTF is a JSON-based open 3D format by Khronos Group for the exchange of 3D models. It also has a binary encoding for storing mesh geometry and animation data. It provides compact representation of geometries, and small file sizes.

CesiumJS: `https://cesium.com/cesiumjs/`
three.js: `https://threejs.org/`

It used for instance in CesiumJS (which supports semantic 3D city models to some extents), and in other libraries like *three.js*.

1.5.3 LandInfra & InfraGML

LandInfra is a relatively new OGC open standard for land and infrastructure features, integrating concepts from IFC/BIM (see Lesson 6.2) and CityGML.

It actually partially overlaps with CityGML: it contains the thematic classes 'Building', 'Road' and 'Railway' (`Transportation` in CityGML), and 'LandSurface' (`ReliefFeature` in CityGML). However, it has a more detailed representation for land and infrastructure features, eg administrative units, ownership rights, spatial units for land use (land parcels and the legal spaces of buildings), surveying and representation, alignment for roads and railways, subsurface models for terrain, etc

InfraGML is the GML-based encoding of LandInfra, and the only one standardised.

LandInfra is a relatively young standard and at present it is difficult to identify any concrete examples of its usage in practice; the majority of citations about LandInfra describe the need to consider LandInfra in future work.

1.6 Notes and comments

The official specifications of CityGML (in PDF format) are available at `https://www.opengeospatial.org/standards/citygml`. A summary is available in Gröger and Plümer (2012).

(Stadler and Kolbe, 2007) first proposed and described the semantic and spatial decompositions of a city, and how keeping the two decomposition aligned has several advantages in practice.

Biljecki et al. (2015) describe and list 30 use-cases and 100 applications that make use of semantic 3D city models.

See Biljecki et al. (2018) for an overview of the existing ADEs.

CityJSON specifications, examples datasets, tutorials, and software are available at `https://cityjson.org`. Ledoux et al. (2019) discuss in details the encoding and give concrete examples why they believe it is a superior encoding to XML for the CityGML data model; parts of this lesson was taken and adapted from that paper.

Airaksinen et al. (2019) describe the efforts and workflows used by the city Helsinki to built both a textures mesh and a semantic 3D city models of their city. Details about how the model is used in practice are also given.

Kumar et al. (2019) describe the role and position of LandInfra with respect to CityGML and BIM/IFC.

For the description of LoD of other classes then buildings, see Kumar et al. (2019) for terrains, Labetski et al. (2018) for roads, and Ortega-Córdova (2018) for trees.

1.7 Exercises

1. It is stated that a given CityJSON file will be on average 6X compacter than an equivalent CityGML file. Explain why CityJSON files are compacter.
2. Build manually a CityJSON file of a unit cube that represent a LoD2 building, and assign to its surfaces the correct semantics (roof, ground, façade). Add a few random attributes to the building. Make sure your file is valid by following that tutorial: `https://www.cityjson.org/tutorials/validation/`
3. What would be the "best" format to store the textured mesh of Helsinki (in Figure 1.1)?

Bibliography

Airaksinen, E., M. Bergström, H. Heinonen, K. Kaisla, K. Lahti, and J. Suomisto (2019). *The Kalasatama digital twins project—The final report of the KIRA-digi pilot project*. Tech. rep. City of Helsinki. URL: https://www.hel.fi/static/liitteet-2019/Kaupunginkanslia/Helsinki3D_Kalasatama_Digital_Twins.pdf.

Biljecki, F., K. Kumar, and C. Nagel (2018). CityGML Application Domain Extension (ADE): overview of developments. *Open Geospatial Data, Software and Standards* 3.1.

Biljecki, F., J. Stoter, H. Ledoux, S. Zlatanova, and A. Çöltekin (2015). Applications of 3D City Models: State of the Art Review. *ISPRS International Journal of Geo-Information* 4.4, pp. 2842–2889.

Gröger, G. and L. Plümer (2012). CityGML—Interoperable semantic 3D city models. *ISPRS Journal of Photogrammetry and Remote Sensing* 71, pp. 12–33.

Kumar, K., A. Labetski, K. Arroyo Ohori, H. Ledoux, and J. Stoter (2019). The LandInfra standard and its role in solving the BIM-GIS quagmire. *Open Geospatial Data, Software and Standards* 4.5.

Labetski, A., S. van Gerwen, G. Tamminga, H. Ledoux, and J. Stoter (2018). A proposal for an improved transportation model in CityGML. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*. Vol. XLII-4/W10, pp. 89–96.

Ledoux, H., K. A. Ohori, K. Kumar, B. Dukai, A. Labetski, and S. Vitalis (2019). CityJSON: a compact and easy-to-use encoding of the CityGML data model. *Open Geospatial Data, Software and Standards* 4.4.

Ortega-Córdova, L. (2018). Urban vegetation modeling 3D levels of detail. MA thesis. MSc thesis in Geomatics, Delft University of Technology.

Stadler, A. and T. H. Kolbe (2007). Spatio-semantic coherence in the integration of 3D city models. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences. Proceedings of the WG II/7 5th International Symposium Spatial Data Quality 2007 with the theme: Modelling qualities in space and time*. Ed. by A. Stein. Enschede, the Netherlands, p. 8.