CityJSON Documentation

Release 2.4.0

3D geoinformation (TU Delft)

Jun 28, 2023

Contents

1 Installation **3**
- 1.1 Windows executable . . . 3
- 1.2 macOS & Linux: you need to compile . . . 3
- 1.3 Web-application . . . 3
- 1.4 Using val3dity as a library . . . 3
- 1.5 Python bindings . . . 3

2 Using val3dity **5**
- 2.1 How to run val3dity? . . . 5
- 2.2 Accepted input . . . 6
- 2.3 How are 3D primitives validated? . . . 6
- 2.4 Options for the validation . . . 7

3 Errors **11**
- 3.1 101 – TOO_FEW_POINTS . . . 14
- 3.2 102 – CONSECUTIVE_POINTS_SAME . . . 14
- 3.3 103 – RING_NOT_CLOSED . . . 14
- 3.4 104 – RING_SELF_INTERSECTION . . . 15
- 3.5 201 – INTERSECTION_RINGS . . . 15
- 3.6 202 – DUPLICATED_RINGS . . . 15
- 3.7 203 – NON_PLANAR_POLYGON_DISTANCE_PLANE . . . 16
- 3.8 204 – NON_PLANAR_POLYGON_NORMALS_DEVIATION . . . 16
- 3.9 205 – POLYGON_INTERIOR_DISCONNECTED . . . 16
- 3.10 206 – INNER_RING_OUTSIDE . . . 17
- 3.11 207 – INNER_RINGS_NESTED . . . 17
- 3.12 208 – ORIENTATION_RINGS_SAME . . . 17
- 3.13 300 – NOT_VALID_2_MANIFOLD . . . 17
- 3.14 301 – TOO_FEW_POLYGONS . . . 18
- 3.15 302 – SHELL_NOT_CLOSED . . . 18
- 3.16 303 – NON_MANIFOLD_CASE . . . 18
- 3.17 305 – MULTIPLE_CONNECTED_COMPONENTS . . . 19
- 3.18 306 – SHELL_SELF_INTERSECTION . . . 19
- 3.19 307 – POLYGON_WRONG_ORIENTATION . . . 19
- 3.20 401 – INTERSECTION_SHELLS . . . 19
- 3.21 402 – DUPLICATED_SHELLS . . . 19
- 3.22 403 – INNER_SHELL_OUTSIDE . . . 20
- 3.23 404 – SOLID_INTERIOR_DISCONNECTED . . . 20

3.24 405 – WRONG_ORIENTATION_SHELL . . . 20
3.25 501 – INTERSECTION_SOLIDS . . . 20
3.26 502 – DUPLICATED_SOLIDS . . . 21
3.27 503 – DISCONNECTED_SOLIDS . . . 21
3.28 601 – BUILDINGPARTS_OVERLAP . . . 21
3.29 701 – CELLS_OVERLAP . . . 21
3.30 702 – DUAL_VERTEX_OUTSIDE_CELL . . . 21
3.31 703 – PRIMAL_DUAL_XLINKS_ERROR . . . 22
3.32 704 – PRIMAL_DUAL_ADJACENCIES_INCONSISTENT . . . 22
3.33 901 – INVALID_INPUT_FILE . . . 22
3.34 902 – EMPTY_PRIMITIVE . . . 22
3.35 903 – WRONG_INPUT_PARAMETERS . . . 22
3.36 904 – FORMAT_NOT_SUPPORTED . . . 22
3.37 906 – PRIMITIVE_NO_GEOMETRY . . . 22
3.38 999 – UNKNOWN_ERROR . . . 22

4 Definitions **23**
4.1 ISO19107 primitives . . . 23
4.2 Aggregates & composites . . . 23
4.3 Overview of 3D primitives handled . . . 24
4.4 `Polygon` . . . 24
4.5 `MultiSurface` . . . 25
4.6 `CompositeSurface` . . . 25
4.7 `Solid` . . . 25
4.8 `MultiSolid` . . . 27
4.9 `CompositeSolid` . . . 27

5 FAQ **29**
5.1 Who validates the validator, huh? . . . 29
5.2 How to interpret the report? . . . 30
5.3 I get many errors 203 and 204, but my planes look planar to me. Why is that? . . . 31
5.4 I don't see all the errors in my solid . . . 31
5.5 I'm sure my 3D primitive is valid, but the validator says that something is wrong . . . 32
5.6 Do you validate the topological relationships between the solids? . . . 32
5.7 The IDs for the shells and surfaces in the report, are they 0-based or 1-based? . . . 32
5.8 Where can I get files containing `Solids` or `CompositeSolid`? . . . 32

6 Contact **33**

(version 2.4.0)

val3dity—pronounced 'val-three-dity'—allows us to validate 3D primitives according to the international standard ISO19107. Think of it as PostGIS ST_IsValid, but for 3D primitives (PostGIS is only for 2D ones).

In short, it verifies whether a 3D primitive respects the definition as given in ISO19107 and GML/CityGML. The validation of the following 3D primitives is fully supported:

- `MultiSurface`
- `CompositeSurface`
- `Solid`
- `MultiSolid`
- `CompositeSolid`

Unlike many other validation tools in 3D GIS, inner rings in polygons/surfaces are supported and so are cavities in solids (also called voids or inner shells). However, as is the case for many formats used in practice, only planar and linear primitives are allowed: no curves or spheres or other parametrically-modelled primitives are supported. There is no plan to support these geometries.

val3dity accepts as input:

- CityJSON
- CityJSON Lines (CityJSONL)
- tu3djson
- JSON-FG (OGC Features and Geometries JSON)
- OBJ
- OFF
- IndoorGML

For the CityJSON and IndoorGML formats, extra validations (specific to the format) are performed, eg the overlap between different parts of a building, or the validation of the navigation graph in IndoorGML.

Note: If you use val3dity in a scientific context, please cite these articles:

Ledoux, Hugo (2019). val3dity: validation of 3D GIS primitives according to the international standards. *Open Geospatial Data, Software and Standards*, 3(1), 2018, pp.1 [DOI]

Ledoux, Hugo (2013). On the validation of solids represented with the international standards for geographic information. *Computer-Aided Civil and Infrastructure Engineering*, 28(9):693-706. [PDF] [DOI]

Content

CHAPTER 1

Installation

1.1 Windows executable

For Windows, we offer an executable, and there's a Visual Studio project code in the folder `vs_build`, although installing the dependencies is slightly more complex than for macOS/Linux.

1.2 macOS & Linux: you need to compile

The details to compile and use val3dity are on GitHub.

1.3 Web-application

If you don't want to go through the troubles of compiling and/or installing val3dity, we suggest you use the web application. Simply upload your file to our server and get a validation report back. We delete the file as soon as it has been validated. However, a file is limited to 50MB.

1.4 Using val3dity as a library

val3dity can be compiled and used as a library, see the instructions.

Also, there is a simple example of how to use it in *./demo_lib* with instructions to compile it.

1.5 Python bindings

It is possible to use val3dity with Python, see val3ditypy.

CHAPTER 2

Using val3dity

Note: val3dity is a command-line program only, there is no graphical interface. Alternatively, you can use the web application.

2.1 How to run val3dity?

To execute val3dity and see its options:

```
$ val3dity --help
```

To validate all the 3D primitives in a CityJSON file and see a summary output:

```
$ val3dity my3dcity.city.json
```

To validate each 3D primitive in `input.city.json`, and use a tolerance for testing the planarity of the surface of 20cm (0.2):

```
$ val3dity --planarity_d2p_tol 0.2 input.city.json
```

To validate an OBJ file and verify whether the 3D primitives from a Solid (this is the default):

```
$ val3dity input.obj
```

The same file could be validated as a MultiSurface, ie each of its surface are validated independently

```
$ val3dity -p MultiSurface input.obj
```

2.2 Accepted input

val3dity accepts as input:

- CityJSON
- CityJSON Lines (CityJSONL)
- tu3djson
- JSON-FG (OGC Features and Geometries JSON)
- OBJ
- OFF
- IndoorGML

For **CityJSON** files, all the City Objects (eg `Building` or `Bridge`) are processed and their 3D primitives are validated. The 3D primitives are bundled under their City Objects in the report. If your CityJSON contains `Buildings` with one or more `BuildingParts`, val3dity will perform an extra validation: it will ensure that the 3D primitives do not overlap (technically that the interior of each `BuildingPart` does not intersect with the interior of any other part of the `Building`). If there is one or more intersections, then *601 – BUILDINGPARTS_OVERLAP* will be reported.

For **IndoorGML** files, all the cells (in the primal subdivisions, the rooms) are validated individually, and then some extra validation tests are run on the dual navigation network. All errors 7xx are related specifically to IndoorGML.

For **JSON-FG** files, only `Polyhedron` and `MultiPolyhedron` are processed, the other possible types are ignored (`Prism`, `MultiPrism`, and all the 2D types inherited from GeoJSON). It should be noticed that the JSON-FG nomenclature is different: a `Polyhedron` is a `Solid`, and a `MultiPolyhedron` is a `MultiSolid` (an arbitrary aggregation of several solids and there is no assumption regarding their topological relationships).

For **OBJ** and **OFF** files, each primitive will be validated according to the ISO19107 rules. One must specify how the primitives should be validated (`MultiSurface`, `CompositeSurface`, or `Solid`). In an OBJ file, if there is more than one object (lines starting with "o", eg *o myobject*), each will be validated individually. Observe that OBJ files have no mechanism to define inner shells, and thus a solid will be formed by only its exterior shell. Validating one primitive in an OBJ as a MultiSurface (`-p MultiSurface` option) will individually validate each surface according to the ISO19107 rules, without ensuring that they form a 2-manifold. If your OBJ contains triangles only (often the case), then using the option *-p MultiSurface* is rather meaningless since most likely all your triangles are valid. Validation could however catch cases where triangles are collapsed to a line/point. Validating it as a solid verifies whether the primitive is a 2-manifold, ie whether it is closed/watertight and whether all normals are pointing outwards.

2.3 How are 3D primitives validated?

All primitives are validated hierarchically, for instance:

1. the lower-dimensionality primitives (the polygons) are validated by projecting them to a 2D plane (obtained with least-square adjustment) and using GEOS;
2. then these are assembled into shells/surfaces and their validity is analysed, as they must be watertight, no self-intersections, orientation of the normals must be consistent and pointing outwards, etc;
3. then the `Solids` are validated
4. finally, for `CompositeSolids` the interactions between the `Solids` are analysed.

This means that if one polygon of a Solid is not valid, the validator will report that error but will *not* continue the validation (to avoid "cascading" errors).

The formal definitions of the 3D primitives, along with explanations, are given in *Definitions*.

2.4 Options for the validation

2.4.1 `-h, --help`

Display usage information and exit.

2.4.2 `--ignore204`

Ignore the error *204 – NON_PLANAR_POLYGON_NORMALS_DEVIATION*.

2.4.3 `--overlap_tol`

Tolerance for testing the overlap between primitives in `CompositeSolids` and `BuildingParts`
default = -1 (disabled)

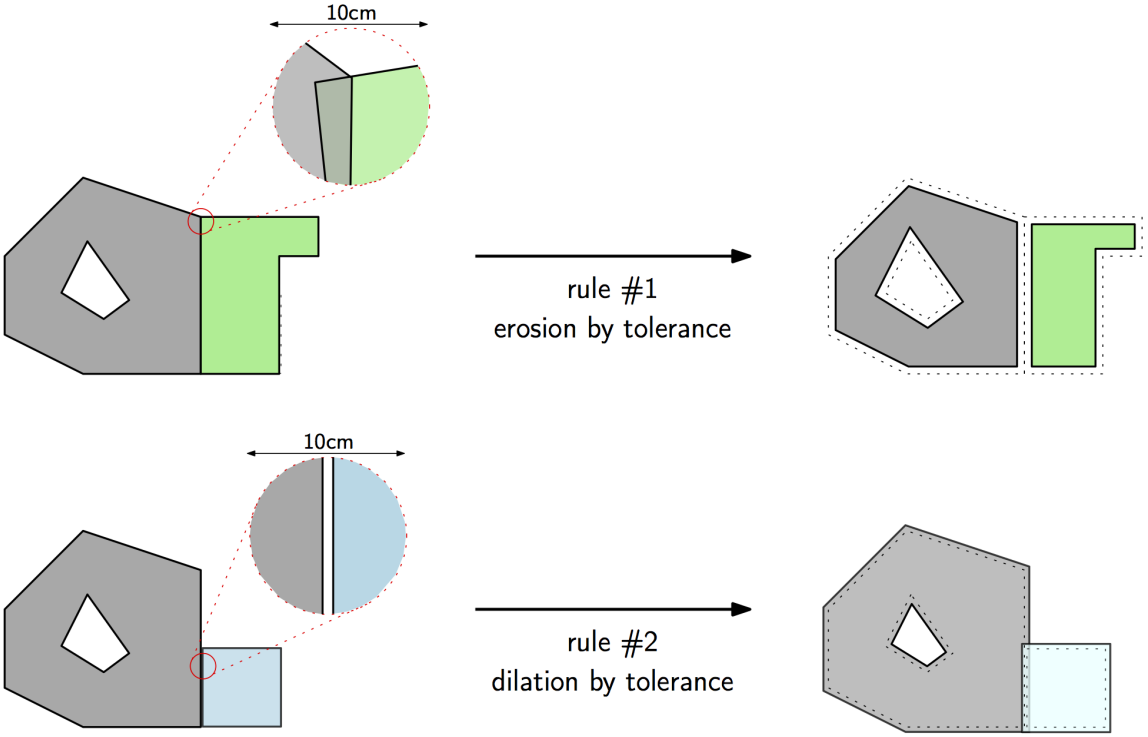
The maximum allowed distance for overlaps. Helps to validate the topological relationship between `Solids` forming a `CompositeSolid`, the `BuildingParts` of a building in CityJSON, or the cells in IndoorGML. The tolerance `--overlap_tol 0.05` means that each of the solids is given a 0.05unit *fuzzy* boundary (thus 5cm if meters are the unit of the input), and thus this is considered when validating. `0.0unit` means that the original boundaries are used. Using an overlap tolerance significantly reduces the speed of the validator, because rather complex geometric operations are performed.

A `CompositeSolid`, formed by the `Solids` A and B, should fulfil the following two properties:

rule #1: their interior should not overlap ($A^o \cap B^o = \emptyset$)

rule #2: their union should form one solid ($A \cup B =$ one Solid)

val3dity can validate these with a user-defined tolerance, eg 2cm. Each `Solid` is either dilated or eroded by tolerance/2, as shown below.



2.4.4 `--planarity_d2p_tol`

Tolerance for planarity based on a distance to a plane
default = 0.01

The distance between every point forming a surface and a plane must be less than `--planarity_d2p_tol` (eg 1cm, which is the default). This plane is fitted with least-square adjustment, and the distance between each of the point to the plane is calculated. If this distance is larger than the defined value, then *203 – NON_PLANAR_POLYGON_DISTANCE_PLANE* is reported. Read more at *203 – NON_PLANAR_POLYGON_DISTANCE_PLANE*.

Note: Planarity is defined with two tolerances: `--planarity_d2p_tol` and `--planarity_n_tol`.

2.4.5 --planarity_n_tol

Tolerance for planarity based on normals deviation
default = 20 (degree)

Helps to detect small folds in a surface. `--planarity_n_tol` refers to the normal of each triangle after the surface has been triangulated. If the triangle normals deviate from each other more than the given tolerance, then error *204 – NON_PLANAR_POLYGON_NORMALS_DEVIATION* is reported. Read more at *204 – NON_PLANAR_POLYGON_NORMALS_DEVIATION*.

Note: Planarity is defined with two tolerances: `--planarity_d2p_tol` and `--planarity_n_tol`.

2.4.6 -p, --primitive

Which geometric primitive to validate. Only relevant for OBJ/OFF, because for CityJSON all primitives are validated. Read more geometric primitives at *Definitions*.
One of `Solid`, `CompositeSurface`, `MultiSurface`.

2.4.7 -r, --report

Outputs the validation report to the file given. The report is in JSON file format, and can be used to produce nice reports automatically or to extract statistics. Use val3dity report browser with your report.

2.4.8 --listerrors

Outputs a list of the val3dity errors.

2.4.9 --snap_tol

Tolerance for snapping vertices that are close to each others
default = 0.001

Geometries modelled in GML store amazingly very little topological relationships. A cube is for instance represented with 6 surfaces, all stored independently. This means that the coordinates xyz of a single vertex (where 3 surfaces "meet") is stored 3 times. It is possible that these 3 vertices are not exactly at the same location (eg (0.01, 0.5, 1.0), (0.011, 0.49999, 1.00004) and (0.01002, 0.5002, 1.0007)), and that would create problems when validating since there would be holes in the cube for example. The snap tolerance basically gives a threshold that says: "if 2 points are closer then *X*, then we assume that they are the same". It's setup by default to be 1mm.

2.4.10 --verbose

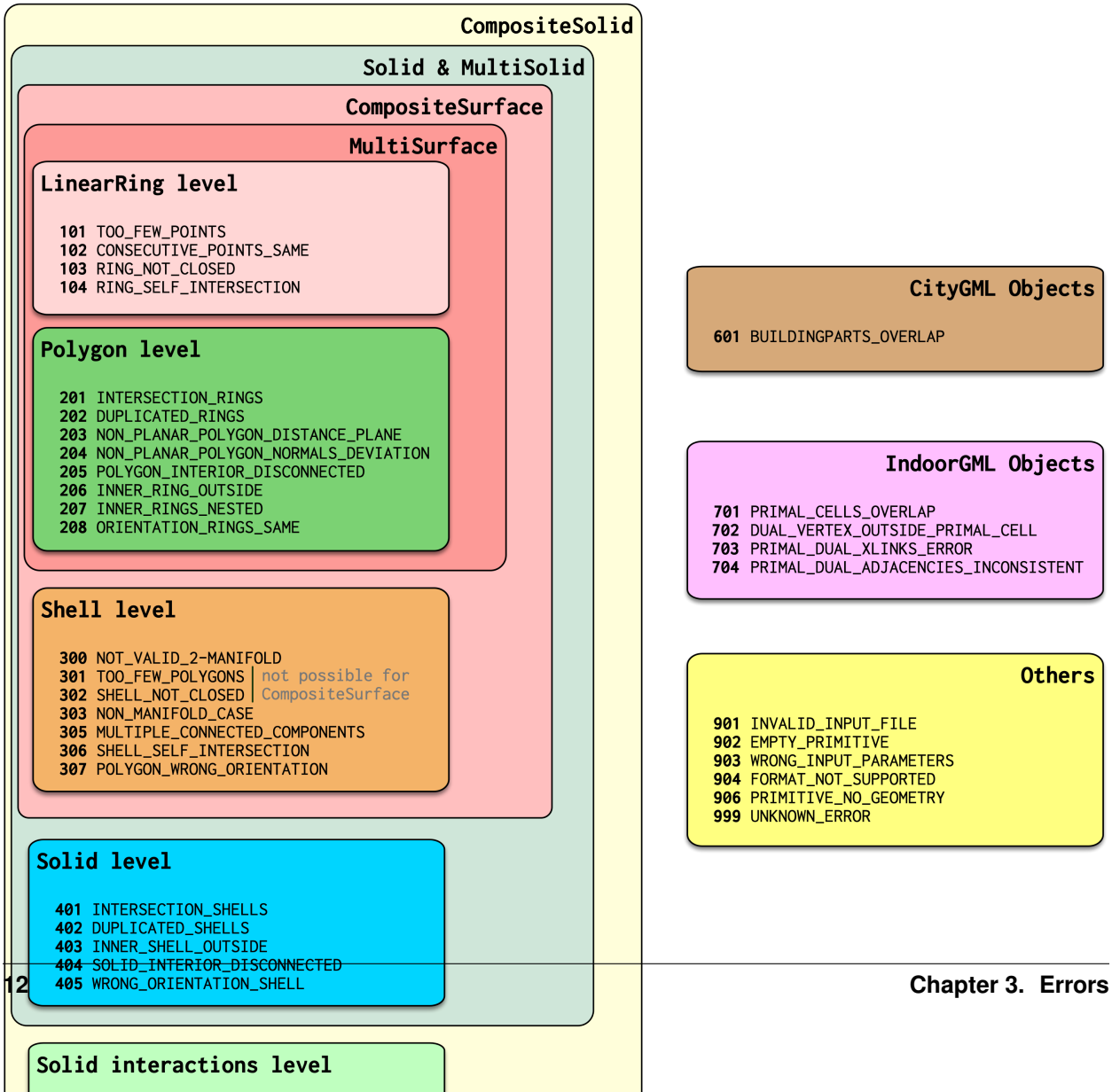
The validation outputs to the console the status of each step of the validation. If this option is not set, then this goes to a file *val3dity.log* in the same folder as the executable.

2.4.11 --version

Display version information and exit.

CHAPTER 3

Errors



- *101 – TOO_FEW_POINTS*
- *102 – CONSECUTIVE_POINTS_SAME*
- *103 – RING_NOT_CLOSED*
- *104 – RING_SELF_INTERSECTION*
- *201 – INTERSECTION_RINGS*
- *202 – DUPLICATED_RINGS*
- *203 – NON_PLANAR_POLYGON_DISTANCE_PLANE*
- *204 – NON_PLANAR_POLYGON_NORMALS_DEVIATION*
- *205 – POLYGON_INTERIOR_DISCONNECTED*
- *206 – INNER_RING_OUTSIDE*
- *207 – INNER_RINGS_NESTED*
- *208 – ORIENTATION_RINGS_SAME*
- *300 – NOT_VALID_2_MANIFOLD*
- *301 – TOO_FEW_POLYGONS*
- *302 – SHELL_NOT_CLOSED*
- *303 – NON_MANIFOLD_CASE*
- *305 – MULTIPLE_CONNECTED_COMPONENTS*
- *306 – SHELL_SELF_INTERSECTION*
- *307 – POLYGON_WRONG_ORIENTATION*
- *401 – INTERSECTION_SHELLS*
- *402 – DUPLICATED_SHELLS*
- *403 – INNER_SHELL_OUTSIDE*
- *404 – SOLID_INTERIOR_DISCONNECTED*
- *405 – WRONG_ORIENTATION_SHELL*
- *501 – INTERSECTION_SOLIDS*
- *502 – DUPLICATED_SOLIDS*
- *503 – DISCONNECTED_SOLIDS*
- *601 – BUILDINGPARTS_OVERLAP*
- *701 – CELLS_OVERLAP*
- *702 – DUAL_VERTEX_OUTSIDE_CELL*
- *703 – PRIMAL_DUAL_XLINKS_ERROR*
- *704 – PRIMAL_DUAL_ADJACENCIES_INCONSISTENT*
- *901 – INVALID_INPUT_FILE*
- *902 – EMPTY_PRIMITIVE*
- *903 – WRONG_INPUT_PARAMETERS*

- *904 – FORMAT_NOT_SUPPORTED*
- *906 – PRIMITIVE_NO_GEOMETRY*
- *999 – UNKNOWN_ERROR*

3.1 101 – TOO_FEW_POINTS

A ring should have at least 3 points. For GML rings, this error ignores the fact that the first and the last point of a ring are the same (see *103 – RING_NOT_CLOSED*), ie a GML ring should have at least 4 points.

This ring is for instance invalid:

```
<gml:LinearRing>
  <gml:pos>0.0 0.0 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>
```

3.2 102 – CONSECUTIVE_POINTS_SAME

Points in a ring should not be repeated (except first-last in case of GML, see *103 – RING_NOT_CLOSED*). This error is for the common error where 2 *consecutive* points are at the same location. Error 104 is for points in a ring that are repeated, but not consecutive.

This ring is for instance invalid:

```
<gml:LinearRing>
  <gml:pos>0.0 0.0 0.0</gml:pos>
  <gml:pos>1.0 0.0 0.0</gml:pos>
  <gml:pos>1.0 0.0 0.0</gml:pos>
  <gml:pos>1.0 1.0 0.0</gml:pos>
  <gml:pos>0.0 1.0 0.0</gml:pos>
  <gml:pos>0.0 0.0 0.0</gml:pos>
</gml:LinearRing>
```

3.3 103 – RING_NOT_CLOSED

This applies only to GML rings and to JSON-FG, in CityJSON/OBJ/OFF it is ignored. The first and last points have to be identical (at the same location). This is verified after the points have been merged with the *–snap_tol* option.

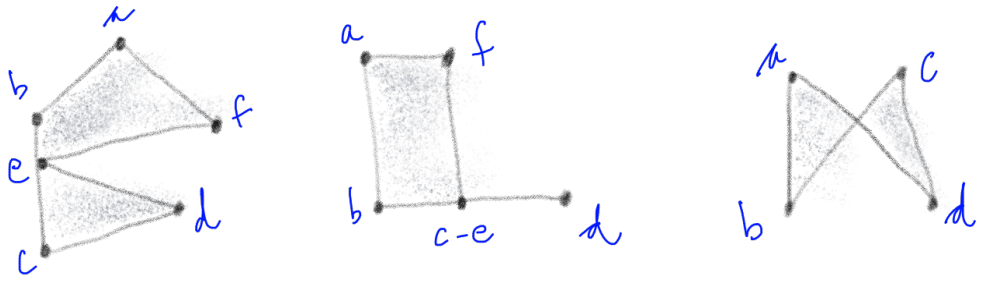
This ring is for instance invalid:

```
<gml:LinearRing>
  <gml:pos>0.0 0.0 0.0</gml:pos>
  <gml:pos>1.0 0.0 0.0</gml:pos>
  <gml:pos>1.0 1.0 0.0</gml:pos>
  <gml:pos>0.0 1.0 0.0</gml:pos>
</gml:LinearRing>
```

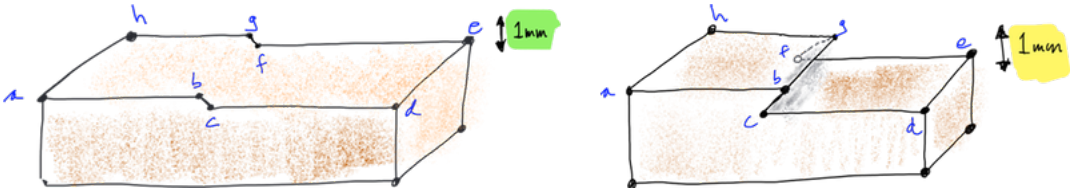
3.4 104 – RING_SELF_INTERSECTION

A ring should be *simple*, ie it should not self-intersect. The self-intersection can be at the location of an explicit point, or not. This case includes rings that are (partly) collapsed to a line for instance.

Observe that self-intersection in 3D and 2D is different, ie a bowtie (the first polygon below) has a self-intersection "in the middle" in 2D, but in 3D if the 4 vertices are not on a plane then there is no intersection.

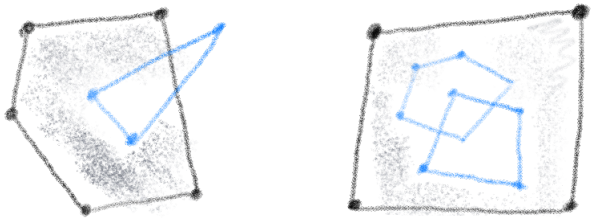


A ring is self-intersecting if its projection to the best-fitted plane (done with least-square) through the vertices of the polygon containing the ring has a self-intersection. This rule is there because if it is not possible to project the rings/polygons to a plane, then it is not possible to triangulate it (which is necessary, at least by val3dity, to validate 3D primitives). In the figure below, the left example shows one polygon (the top one) where a projection (let say to the xy-plane) would not cause any self-intersection. However, the right example does cause a self-intersection. It is the same is the vertices *b* and *c* are projected to the same location: a self-intersection is also returned.



3.5 201 – INTERSECTION_RINGS

Two or more rings intersect, these can be either the exterior ring with an interior ring or only interior rings.



3.6 202 – DUPLICATED_RINGS

Two or more rings are identical.

3.7 203 – NON_PLANAR_POLYGON_DISTANCE_PLANE

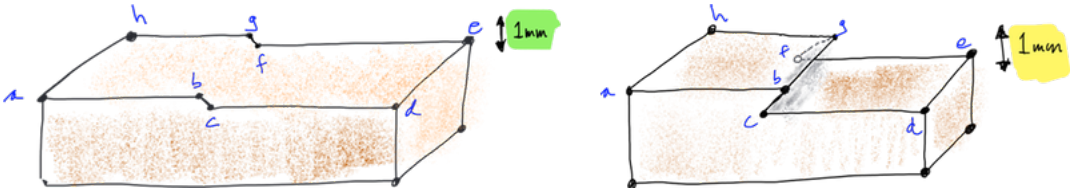
A polygon must be planar, ie all its points (used for both the exterior and interior rings) must lie on a plane. To verify this, we must ensure that the the distance between every point and a plane is less than a given *tolerance* (eg 1cm). In the validator, this plane is fitted with least-square adjustment, and the distance between each of the point to the plane is calculated. If the distance is larger than the given threshold then an error is reported. The distance to the plane, if larger than the threshold, is also reported in the report.

3.8 204 – NON_PLANAR_POLYGON_NORMALS_DEVIATION

To ensure that small folds on a surface are detected. Consider the Solid below, the top surface containing 8 vertices (*abcdefgh*) is clearly non-planar since there is a vertical "fold" in the middle. The normal of the sub-surface *abgh* points upwards, while that of *bcfg* points in a different angle. But this surface would not be detected by the *203 – NON_PLANAR_POLYGON_DISTANCE_PLANE* test (with a tolerance of 1cm for instance) since all the vertices are within that threshold. Thus, another requirement is necessary: the distance between every point forming a polygon and *all* the planes defined by all possible combinations of 3 non-collinear points is less than a given tolerance. In practice it can be implemented with a triangulation of the polygon (any triangulation): the orientation of the normal of each triangle must not deviate more than than a certain user-defined tolerance.

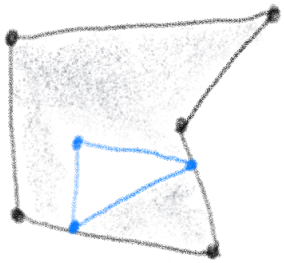
A surface is first checked for *203 – NON_PLANAR_POLYGON_DISTANCE_PLANE*, if it's valid then *204 – NON_PLANAR_POLYGON_NORMALS_DEVIATION* is checked. However, it is only checked if there are no *104 – RING_SELF_INTERSECTION* in the polygon, since otherwise it's not possible to triangulate the polygon. In the figure below, the Solid on the left could be tested for 204, while the right one couldn't (but an *104 – RING_SELF_INTERSECTION* would be returned).

By definition, if *204 – NON_PLANAR_POLYGON_NORMALS_DEVIATION* is reported then all the vertices are within 1cm (or the tolerance you gave as input), thus you wouldn't be able to visualise them. Also, 204 usually means that the vertices in the polygon are *very* close to each other (say 0.1mm), and thus it's easy to get a large deviation (say 80 degree; the report contains the actual deviation).



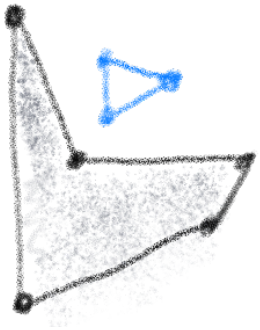
3.9 205 – POLYGON_INTERIOR_DISCONNECTED

The interior of a polygon must be connected. The combination of different valid rings can create such an error, for example:



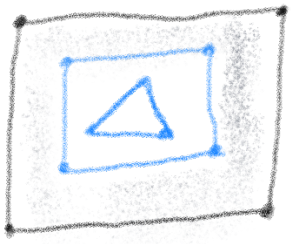
3.10 206 – INNER_RING_OUTSIDE

One or more interior rings are located *completely* outside the exterior ring. If the interior ring intersects the exterior ring (even at only one point), then error *201 – INTERSECTION_RINGS* should be returned.



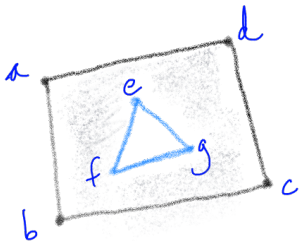
3.11 207 – INNER_RINGS_NESTED

One or more interior ring(s) is(are) located completely inside another interior ring.



3.12 208 – ORIENTATION_RINGS_SAME

The interior rings must have the opposite direction (clockwise vs counterclockwise) when viewed from a given point-of-view. When the polygon is used as a bounding surface of a shell, then the rings have to have a specified orientation (see 307/308).



3.13 300 – NOT_VALID_2_MANIFOLD

The shell is not valid, but the exact error (errors 3xx) is not known. This error happens when the construction of the shell failed for unknown reason. Hopefully you don't get that error.

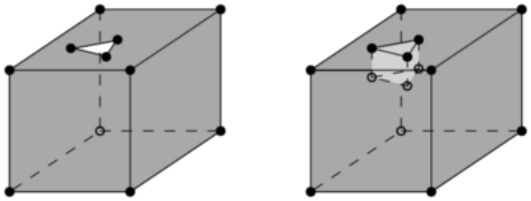
3.14 301 – TOO_FEW_POLYGONS

A shell should have at least 4 polygons—the simplest volumetric shape in 3D is a tetrahedron.

3.15 302 – SHELL_NOT_CLOSED

The shell must not have 'holes', ie it must be 'watertight'. This refers only to the topology of the shell, not to its geometry (see *306 – SHELL_SELF_INTERSECTION*).

The left solid is invalid, while the right one is valid (since the hole is filled with other polygons):

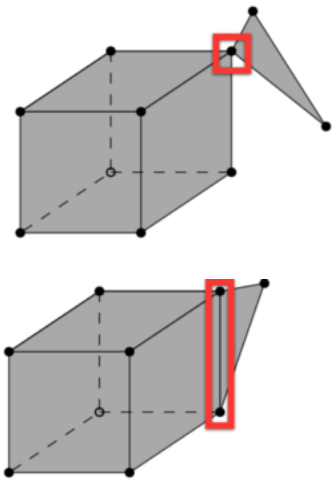


3.16 303 – NON_MANIFOLD_CASE

Each shell must be simple, ie it must be a 2-manifold. Two cases are possible:

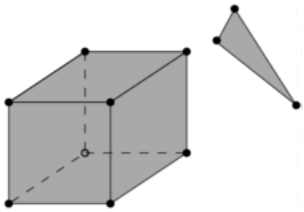
1. An edge of a shell is non-manifold when there are more than 2 incident polygons to it.
2. A vertex is non-manifold when its incident polygons do not form one 'umbrella'

Notice that this error might be returned for shells having their surfaces that are not consistently oriented (while they are 2-manifold). Imagine you have a cube with 6 surfaces, if some surfaces have their normal pointing outwards, and some inwards, then this error might be returned (or *307 – POLYGON_WRONG_ORIENTATION*, depending on the configuration).



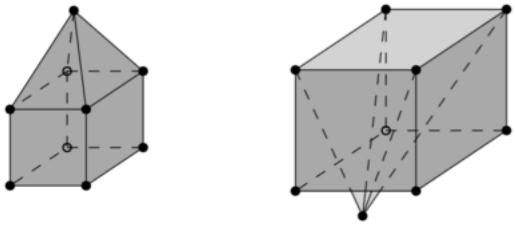
3.17 305 – MULTIPLE_CONNECTED_COMPONENTS

Polygons that are not connected to the shell should be reported as an error.



3.18 306 – SHELL_SELF_INTERSECTION

If topology of the shell is correct and the shell is closed (thus no error 301/302/303/304/305), it is possible that the geometry introduces errors, eg intersections. For instance, the topology of both these shells is identical, but the geometry differs. The left shell is valid while the right one is invalid.



3.19 307 – POLYGON_WRONG_ORIENTATION

If one polygon is used to construct a shell, its exterior ring must be oriented in such as way that when viewed from outside the shell the points are ordered counterclockwise.

3.20 401 – INTERSECTION_SHELLS

The interior of two or more shells intersect, these can be either the exterior shells with an interior shells or two or more interior shells. Two shells sharing (part of) a face is also not allowed.

Conceptually the same as *201 – INTERSECTION_RINGS*.

3.21 402 – DUPLICATED_SHELLS

Two or more shells are identical in *one* Solid. Note that for example a MultiSolid is a collection of Solids, but the topological relationships between them are not prescribed at all, they can be duplicated.

Conceptually the same as *202 – DUPLICATED_RINGS*.

3.22 403 – INNER_SHELL_OUTSIDE

One or more interior shells are located *completely* outside the exterior shell. If the interior shell intersects the exterior shell (even at only one point), then error *401 – INTERSECTION_SHELLS* should be returned.

Conceptually the same as *206 – INNER_RING_OUTSIDE*.

3.23 404 – SOLID_INTERIOR_DISCONNECTED

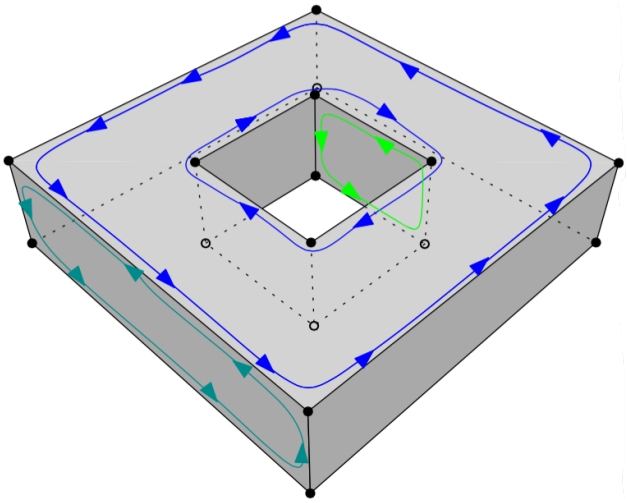
Conceptually the same as *205 – POLYGON_INTERIOR_DISCONNECTED* the configuration of the interior shells makes the interior of the solid disconnected.

3.24 405 – WRONG_ORIENTATION_SHELL

The polygon/surfaces forming an outer shell should have their normals pointing outwards, and for an interior shell inwards.

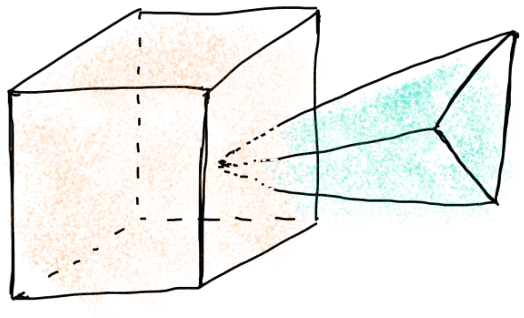
'Outwards' is as follows: if a right-hand system is used, ie when the ordering of the points on the surface follows the direction of rotation of the curled fingers of the right hand, then the thumb points towards the outside. The torus below shows the correct orientation for some rings of some faces.

Conceptually the same as *208 – ORIENTATION_RINGS_SAME*.



3.25 501 – INTERSECTION_SOLIDS

The interior of 2 `Solids` part of a `CompositeSolid` intersects.

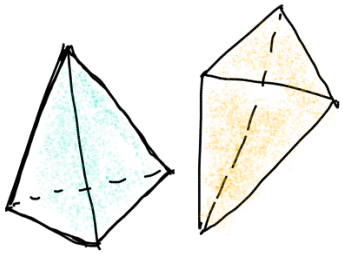


3.26 502 – DUPLICATED_SOLIDS

Two `Solids` in a `CompositeSolid` are identical.

3.27 503 – DISCONNECTED_SOLIDS

Two `Solids` in a `CompositeSolid` are disconnected.



3.28 601 – BUILDINGPARTS_OVERLAP

Some primitives in a Building and/or BuildingPart have their interior overlapping.

3.29 701 – CELLS_OVERLAP

Two IndoorGML cells overlap with each other. Similar to *601 – BUILDINGPARTS_OVERLAP* but for IndoorGML. The overlap allowed can be controlled with the validation option *–overlap_tol*.

3.30 702 – DUAL_VERTEX_OUTSIDE_CELL

The dual vertex of an IndoorGML cell is located outside the cell.

3.31 703 – PRIMAL_DUAL_XLINKS_ERROR

The XLinks in an IndoorGML file are wrong, the primal and the dual are not correctly linked. The schema validation (.xsd) of IndoorGML does not validate this.

3.32 704 – PRIMAL_DUAL_ADJACENCIES_INCONSISTENT

The adjacency of IndoorGML cells in the primal and the dual are no consistent. Basically, if two 3-cells are adjacent in the primal, are they also in the dual (do they have a dual edge)? The *–overlap_tol* influences this.

3.33 901 – INVALID_INPUT_FILE

Input file is not valid or corrupted. If a CityGML file, you can check it against the schema.

3.34 902 – EMPTY_PRIMITIVE

The input file contains empty primitives, eg an OBJ where to surfaces are defined, or a `CompositeSolid` in a CityJSON file containing one empty Solid.

3.35 903 – WRONG_INPUT_PARAMETERS

The parameters used for the validation are not valid.

3.36 904 – FORMAT_NOT_SUPPORTED

It can be that certain versions of a supported format are not supported, eg v3.0 of CityGML is not.

3.37 906 – PRIMITIVE_NO_GEOMETRY

The primitive has no geometry that val3dity recognises. It could be that the primitive has a geometry, but that it is a 2D geometry or a format that is not processed by val3dity. In CityJSON, it is possible to have for instance a `Building` without geometry, so this error would be reported. In JSON-FG, the 2D geometries in `"geometry"` are not processed, and neither as the 3D geometries of type `"Prism"`.

3.38 999 – UNKNOWN_ERROR

If none of the above is suitable, which means something went (really) bad. If you see this error, please report it.

CHAPTER 4

Definitions

The international standard ISO19107 provides definitions for the 3D primitives as used in GIS applications and the "geo-world". Because the aim is to represent *all* the possible real-world features, the 3D primitives are more complex than that in other fields (where often-times a volumetric object is restricted to be a 2-manifold, and where inner rings in surfaces are disallowed), or that of 3D objects in several commercial 3D GIS packages (where the 3D primitives are simply not defined and where inner boundaries in surfaces and solids are disallowed).

4.1 ISO19107 primitives

The 3D primitives as defined in ISO19107 are a generalisation to 3D of the 2D ones and are as follows:

A 0D primitive is a `GM_Point`, a 1D a `GM_Curve`, a 2D a `GM_Surface`, and a 3D a `GM_Solid`. A d-dimensional primitive is built with a set of (d-1)-dimensional primitives, eg a `GM_Solid` is formed by several `GM_Surfaces`, which are formed of several `GM_Curves`, which are themselves formed of `GM_Point`.

Note: While the ISO19107 primitives do not need to be linear or planar (eg curves defined by mathematical functions are allowed), val3dity uses the following restrictions (which are the same as the international standard CityGML and most (all perhaps?) 3D GIS):

1. `GM_Curves` can only be *linear*;
2. `GM_Surfaces` can only be *planar*.

4.2 Aggregates & composites

Primitives can be combined into either *aggregates* or *composites*.

An aggregate is an arbitrary collection of primitives of the same dimensionality that is simply used to bundle together geometries. GML (and CityGML) has classes for each dimensionality (`Multi*`). An aggregate does not prescribe

any topological relationships between the primitives, it is simply a list of primitives (ie they can overlap or be disconnected).

A composite of dimension *d* is a collection of *d*-dimensional primitives that form a *d*-manifold, which is a topological space that is locally like a *d*-dimensional Euclidean space ($\mathbb{R}^d$). The most relevant example in a GIS context is a `CompositeSurface`: it is a 2-manifold, or, in other words, a surface embedded in $\mathbb{R}^d$. An obvious example is the surface of the Earth, for which near to every point the surrounding area is topologically equivalent to a plane.

4.3 Overview of 3D primitives handled

Observe that for a primitive to be valid, all its lower-dimensionality primitives have to be valid. For instance, a valid Solid cannot have as one of its surfaces a Polygon having a self-intersection (which would make it invalid).

4.4 `Polygon`

A Polygon in the context of val3dity is always embedded in $\mathbb{R}^d$, ie its vertices have (*x, y, z*) coordinates. To be valid, it needs to fulfil the 6 assertions below, which are given on pages 27-28 of the Simple Features document. These rules are verified by first projecting each Polygon to a plane, this plane is obtained in val3dity by least-square adjustment of all the points of a Polygon. A Polygon must also be *planar* to be valid: its points (used for both the exterior and interior rings) have to lie on a plane (see *203 – NON_PLANAR_POLYGON_DISTANCE_PLANE* and *204 – NON_PLANAR_POLYGON_NORMALS_DEVIATION* more information about this).

1. Polygons are topologically closed;
2. The boundary of a Polygon consists of a set of LinearRings that make up its exterior and interior boundaries;
3. No two Rings in the boundary cross and the Rings in the boundary of a Polygon may intersect at a Point but only as a tangent, eg

$$\forall P \in Polygon, \forall c1, c2 \in P.Boundary(), c1 \neq c2,$$

$$\forall p, q \in Point, p, q \in c1, p \neq q, [p \in c2 \Rightarrow q \notin c2];$$

4. A Polygon may not have cut lines, spikes or punctures eg:

$$\forall P \in Polygon, P = P.Interior.Closure;$$

5. The interior of every Polygon is a connected point set;
6. The exterior of a Polygon with 1 or more holes is not connected. Each hole defines a connected component of the exterior.

Some concrete examples of invalid polygons are shown below, and here are a few explanations:

- Each ring should be closed (p_{11}): its first and its last points should be the same.
- Each ring defining the exterior and interior boundaries should be simple, ie non-self-intersecting (p_1 and p_{10}). Notice that this prevents the existence of rings with zero-area (p_6), and of rings having two consecutive points at the same location. It should be observed that the polygon p_1 is not allowed (in a valid representation of the polygon, the triangle should be represented as an interior boundary touching the exterior boundary).

- The rings of a polygon should not cross (p_3, p_7, p_8 and p_{12}) but may intersect at one tangent point (the interior ring of p_2 is a valid case, although p_2 as a whole is not since the other interior ring is located outside the interior one). More than one tangent point is allowed, as long as the interior of the polygon stays connected (see below).
- A polygon may not have cut lines, spikes or punctures (p_5 or p_6); removing these is known as the regularisation of a polygon (a standard point-set topology operation).
- The interior of every polygon is a connected point set (p_4).
- Each interior ring creates a new area that is disconnected from the exterior. Thus, an interior ring cannot be located outside the exterior ring (p_2) or inside other interior rings (p_9).

4.5 MultiSurface

It is an arbitrary collection of *Polygon*. Validating a `MultiSurface` simply means that each Polygon is validated individually; a `MultiSurface` is valid if all its `Polygons` are valid.

4.6 CompositeSurface

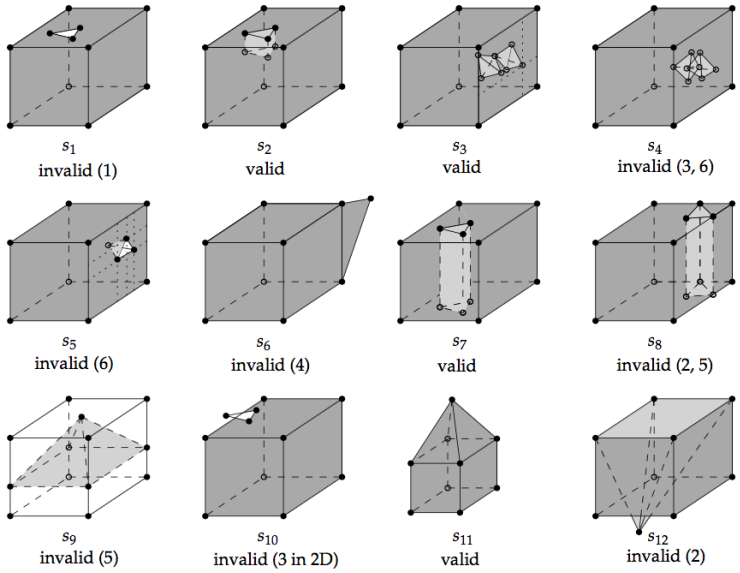
Besides that each Polygon must be individually valid, the Polygons forming a `CompositeSurface` are not allowed to overlap and/or to be disjoint. Furthermore, if we store a `CompositeSurface` in a data structure, each edge is guaranteed to have a maximum of two incident surfaces, and around each vertex the incident faces form one 'umbrella'.

4.7 Solid

A `CompositeSurface` that is closed (ie it contains no 'holes', it is 'watertight') and orientable is referred to as a "Shell". Shells are used to define the boundaries of a `Solid`. In the figure above, the `Solid` has two boundaries: an exterior one (the cube in grey) and one interior one (the cube in orange), which defines a *void' in the solid. A ''Solid'* can have an infinity of interior boundaries, or none. Observe that a cavity is not the same as a hole in a torus (a donut) such as that in the figure below: it can be represented with one exterior boundary having a genus of 1 and no interior shell. Interior boundaries in surfaces are possible, simple LOD1 buildings having for instance an inner yard require them.

According to the ISO19107 specifications, the different boundaries of a solid are allowed to interact with each other, but only under certain circumstances. Since there is no implementation specifications for 3D primitives, we have to generalise the 2D assertions for the validity of a 2D polygon (see *MultiSurface*). Observe that all of them, except the 3rd, generalise directly to 3D since a point-set topology nomenclature is used. The only modifications needed are that, in 3D, polygons become solids, rings become shells, and holes become cavities.

To further explain what the assertions are in 3D, the figure below shows 12 solids, some of them valid, some not; all the statements below refer to these solids.



The first assertion means that a solid must be closed, or 'watertight' (even if it contains interior shells). The solid s_1 is thus not valid but s_2 is since the hole in the top surface is 'filled' with other surfaces.

The second assertion implies that each shell must be *simple* (ie a 2-manifold).

The third assertion means that the boundaries of the shells can intersect each others, but the intersection between the shells can only contain primitives of dimensionality 0 (vertices) and 1 (edges). If a surface or a volume is contained, then the solid is not valid. The solid s_3 is an example of a valid solid: it has two interior shells whose boundaries intersect at one point (at the apexes of the tetrahedra), and the apex of one of the tetrahedra is coplanar with the 4 points forming one surface of the exterior shell. If the interior of the two interior shells intersects (as in s_4) the solid is not valid; this is also related to the sixth assertion stating that each cavity must define one connected component: if the interior of two cavities are intersecting they define the same connected component. Notice also that s_5 is not valid since one surface of its cavity intersects with one surface of the exterior shell (they "share a surface"); s_5 should be represented with one single exterior shell (having a 'dent'), and no interior shell.

The fourth assertion states that a shell is a 2-manifold and that no dangling pieces can exist (such as that of s_6); it is equivalent to the *regularisation* of a point-set in 3D.

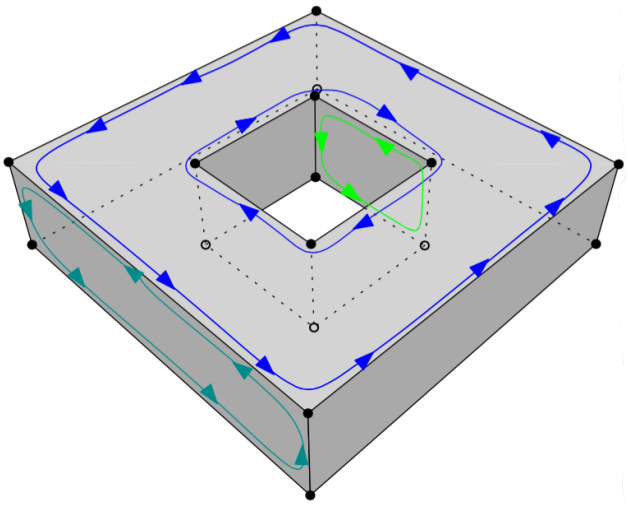
The fifth assertion states that the interior of a solid must form a connected point-set (in 3D). Consider the solid s_7, it is valid since its interior is connected and it fulfils the other assertions; notice that it is a 2-manifold but that unlike other solids in the figure (except s_8) its genus) is 1. If we move the location of the triangular prism so that it touches the boundary of the exterior shell (as in s_8), then the solid becomes invalid since its interior is not connected anymore, and also since its exterior shell is not simple anymore (2 edges have 4 incident planar faces, which is not 2-manifold). It is also possible that the interior shell of a solid separates the solid into two parts: the interior shell of s_9 (exterior shell is not coloured for clarity) is a pyramid having four of its edges intersecting with the exterior shell, but no two surfaces are shared, thus these interactions are allowed. However, the presence of the pyramid separates the interior of the solid into two unconnected volumes (violating assertion 5); for both s_8 and s_9, the only possible valid representation is with two different solids.

Notice also that, as for other primitives, for a solid to be valid all its lower-dimensionality primitives must be valid. That is, each surface of the shells has to be valid. An example of an invalid surface would be one having a hole (an inner ring) overlapping the exterior ring (see s_{10}).

Furthermore, it should also be noticed that for a solid to be valid both its topology and its geometry should be valid. A solid such as s_{11} is valid, but if the location of only one of its vertices is modified (for instance if the apex of the pyramid of s_{11} is moved downwards to form s_{12}) then it becomes invalid. Both s_{11} and s_{12} can be represented with a graph having exactly the same topology (which is valid for both), but if we consider the geometry then the latter solid

is not valid since its exterior shell is not simple.

Lastly, the orientation of the polygons must be considered. In 2D, the only requirement for a polygon is that its exterior ring must have the opposite orientation of that of its interior ring(s) (eg clockwise versus counterclockwise). In 3D, if one polygon is used to construct a shell, its exterior ring must be oriented in such as way that, when viewed from the outside of the shell, the points are ordered counterclockwise. See for instance this solid and the orientation of three of its polygons (different colours).



In other words, the normal of the surface must point outwards if a right-hand system is used, ie when the ordering of points follows the direction of rotation of the curled fingers of the right hand, then the thumb points towards the outside. If the polygon has interior rings, then these have to be ordered clockwise.

4.8 `MultiSolid`

It is an arbitrary collection of `Solids`. Validating a `MultiSolid` simply means that each `Solid` is validated individually; a `MultiSolid` is valid if all its `Solids` are valid.

4.9 `CompositeSolid`

Besides that each `Solid` must be individually valid, the `Solids` are not allowed to overlap and/or to be disjoint.

A `CompositeSolid`, formed by the `Solids` A and B, should fulfil the following two properties:

1. their interior should not overlap ($A^o \cap B^o = \emptyset$)
2. their union should form one `Solid` ($A \cup B =$ one Solid)

val3dity can validate these with a user-defined tolerance (see the option *–overlap_tol*), to ignore small overlaps/gaps that often arise in practice.

CHAPTER 5

FAQ

- *Who validates the validator, huh?*
- *How to interpret the report?*
- *I get many errors 203 and 204, but my planes look planar to me. Why is that?*
- *I don't see all the errors in my solid*
- *I'm sure my 3D primitive is valid, but the validator says that something is wrong*
- *Do you validate the topological relationships between the solids?*
- *The IDs for the shells and surfaces in the report, are they 0-based or 1-based?*
- *Where can I get files containing `Solids` or `CompositeSolid`?*

5.1 Who validates the validator, huh?

We wrote a comprehensive suite of tests using the *pytest* python framework and a custom setup. This tests, among many others, the following:

- Empty files and geometries, invalid input file formats. However, val3dity *does not* validate the schema of the input.
- All the error cases listed in *Errors*.
- Various valid geometries and boundary conditions.
- Command-line user input.

You can read more about the details, or check the complete list of tested geometries

5.2 How to interpret the report?

With the option `--report` a JSON report is output.

The report lists errors at 3 levels:

1. errors with the input files (errors 9xx)
2. errors with the features, eg Buildings in CityJSON (errors 6xx and 7xx)
3. errors with the geometries (errors 1xx – 5xx)

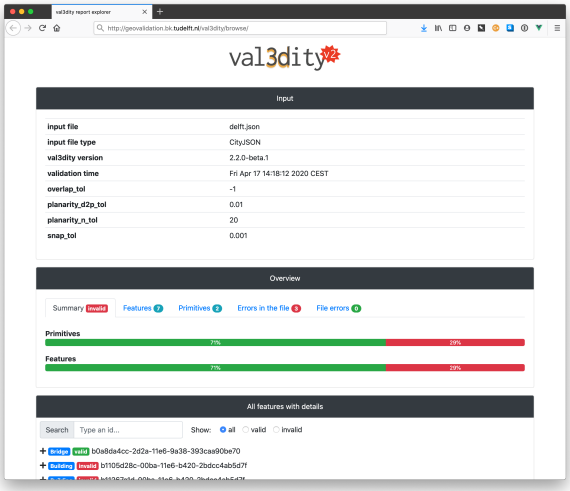
If you use CityJSON, the sub-primitives cannot have IDs, so a 0-based system is used for reporting. For example, if you have a Solid with 2 Shells, then the first one is "0" and the second is "1", and the same applies for each of the surface of a Shell.

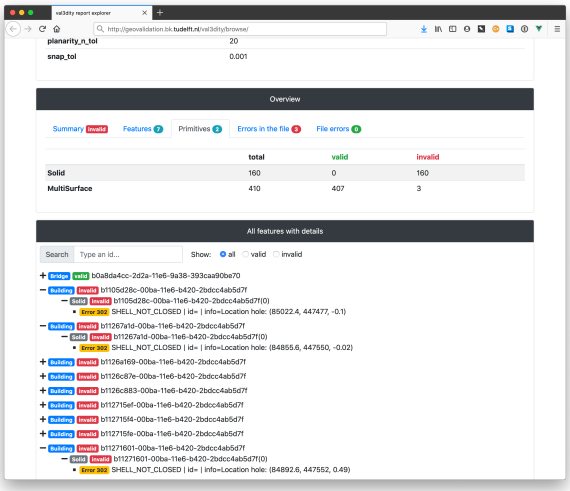
In the report, if one error of a Solid is like this one:

```
{
  "code": 102,
  "description": "CONSECUTIVE_POINTS_SAME",
  "id": "0 | 2",
  "info": "",
  "type": "Error"
}
```

this means that the Shell with ID=0 (the first one, thus the outer Shell) and the 3rd surface in it (because of ID=2) has the error 102. The same principle applies to all primitives, the "|" separate the primitive IDs.

You can navigate this report with a JSON browser (eg drag it in Firefox) or by loading it to the val3dity report browser:





There you get an overview of the statistics per features and primitives, and each feature has its primitives and errors as children.

5.3 I get many errors 203 and 204, but my planes look planar to me. Why is that?

This is a very common error, actually *203 – NON_PLANAR_POLYGON_DISTANCE_PLANE* is the most common error for all the files so far uploaded to our web application.

Read carefully the explanations of the errors *203 – NON_PLANAR_POLYGON_DISTANCE_PLANE* and *204 – NON_PLANAR_POLYGON_NORMALS_DEVIATION*.

A surface is first check for error 203, if valid then error 204 is checked. By definition, if an error 204 is reported then all the vertices are within 1cm (tolerance you used), thus you wouldn't be able to visualise them. That usually means that you have vertices that are very close (say 0.1mm) and thus it's easy to get a large deviation (say 80degree; the report contains the deviation).

5.4 I don't see all the errors in my solid

It's normal: as shown in the figure below, a solid is validated *hierarchically*, ie first every surface (a polygon embedded in 3D) is validated in 2D (by projecting it to a plane), then every shell is validated, and finally the interactions between the shells are analysed to verify whether the solid is valid. If at one stage there are errors, then the validation stops to avoid "cascading errors". So if you get the error *203 – NON_PLANAR_POLYGON_DISTANCE_PLANE*, then fix it and re-run the validator again. That does mean that you might have to upload your file and get it validated several times—if that becomes too tedious we strongly suggest you to download the code, compile it and run it locally (it's open-source and free to use).

5.5 I'm sure my 3D primitive is valid, but the validator says that something is wrong

It's possible that there are bugs in val3dity. Please report the issue and provide the following:

1. the JSON report (use option `--report_json`)
2. (a link to) the input file you used
3. which platform you use, and whether you compiled it yourself or used the web-application

5.6 Do you validate the topological relationships between the solids?

If these solids are part of a `CompositeSolid` then yes, otherwise no. We do verify whether two `BuildingParts` forming a `Building` overlap though.

We however plan to offer in the future this for all primitives/buildings in a file, so that one can verify whether two different buildings overlap for instance.

5.7 The IDs for the shells and surfaces in the report, are they 0-based or 1-based?

0-based.

5.8 Where can I get files containing `Solids` or `CompositeSolid`?

- www.cityjson.org has many files
- in the folder `/data/` of the GitHub repository of val3dity there are many examples of files containing different primitives, and in different formats.
- www.indoorgml.net has a few files
- overview of cities with 3D city models

CHAPTER 6

Contact

val3dity is maintained by Hugo Ledoux and the 3D geoinformation group at TU Delft.

For any questions/issues/errors, please open an issue on GitHub