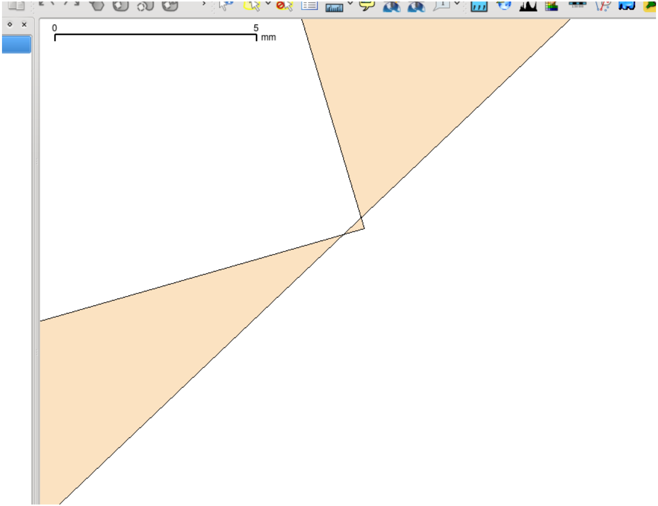
# Validation and automatic repair of two- and three-dimensional GIS datasets

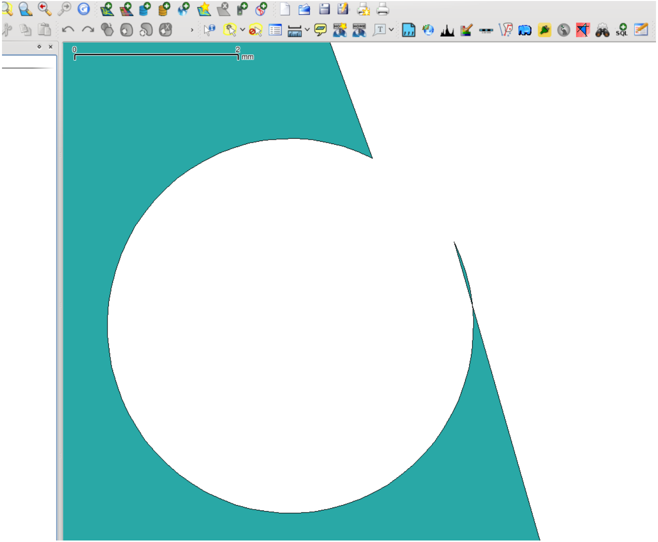
**M. Meijers** H. Ledoux K. Arroyo Ohori J. Zhao

OSGeo.nl dag 2013, Delft
2013–11–13

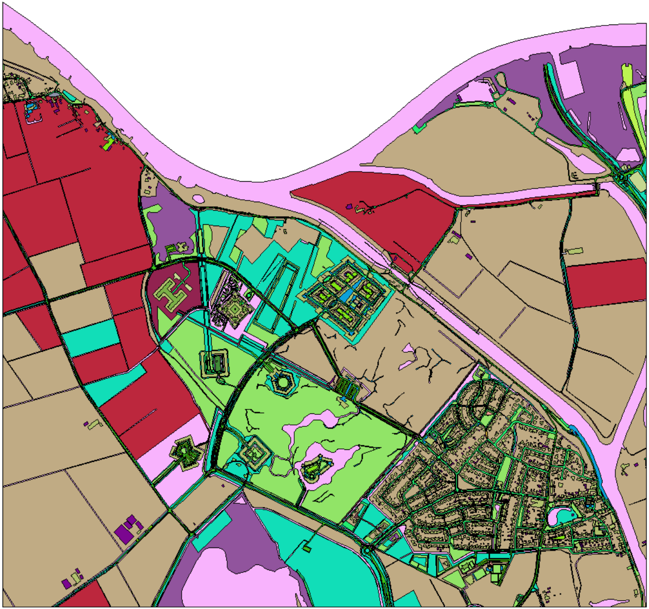
# Typical error: polygon is self-intersecting



# Typical error: polygon is self-intersecting

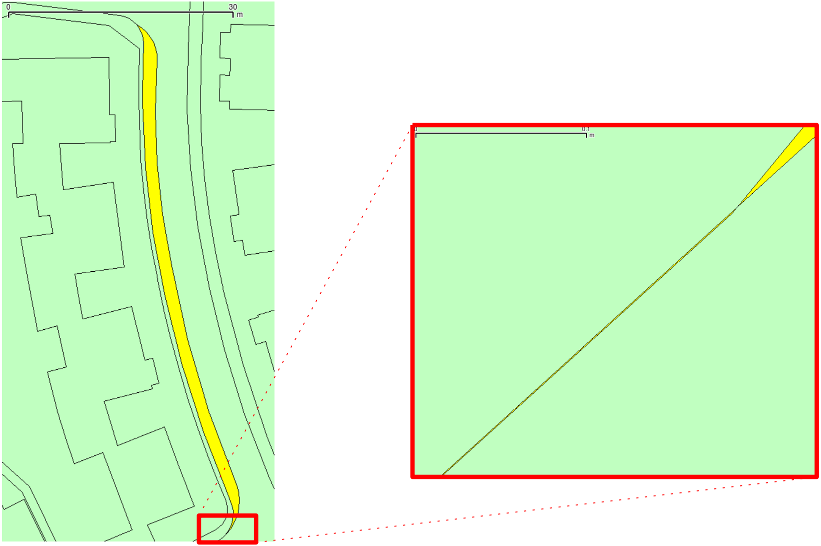


# Typical error: BGT has gaps and/or overlaps

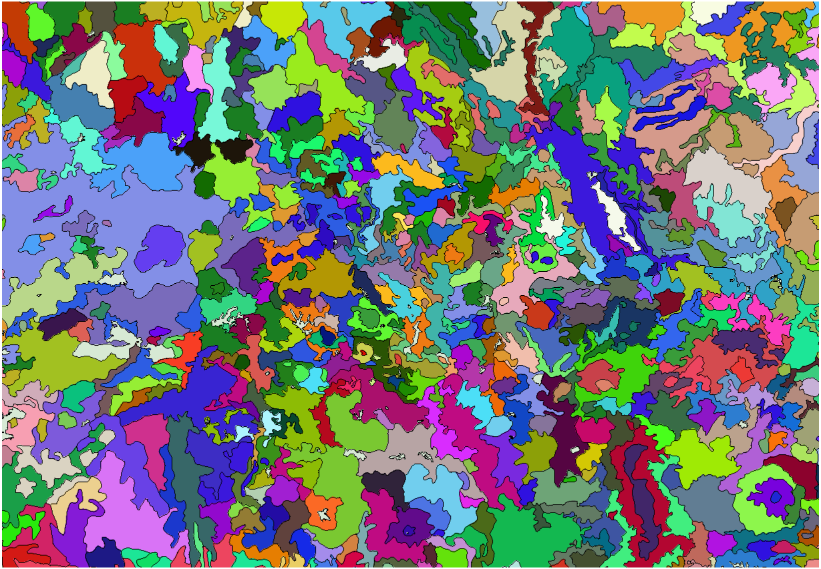


# Typical error: BGT has gaps and/or overlaps

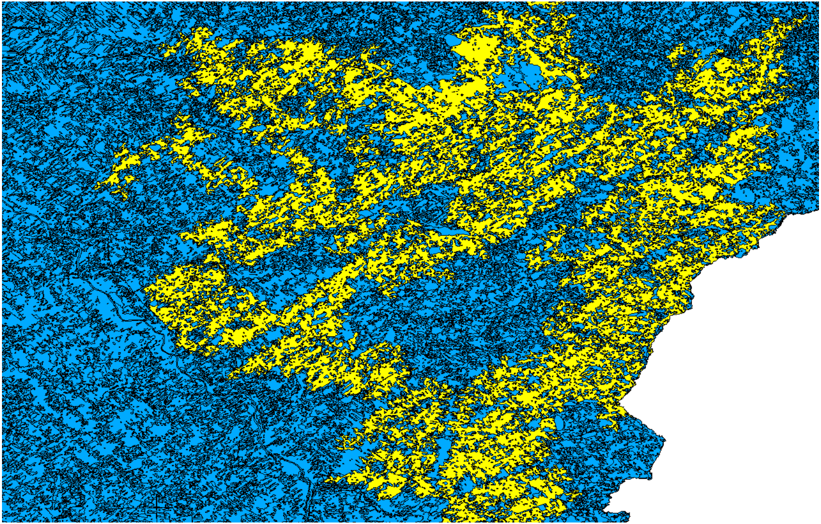
# Typical error: BGT has gaps and/or overlaps



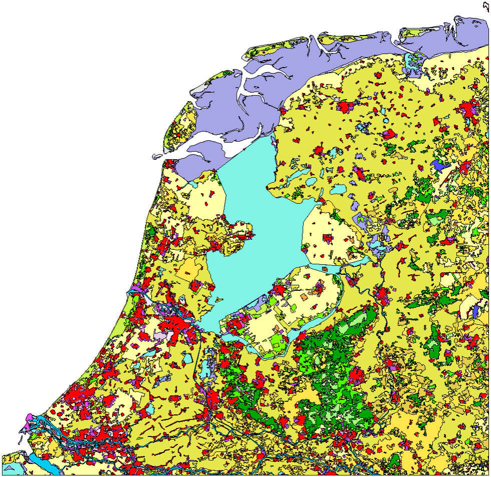
# Big and complex datasets: it quickly gets out-of-control



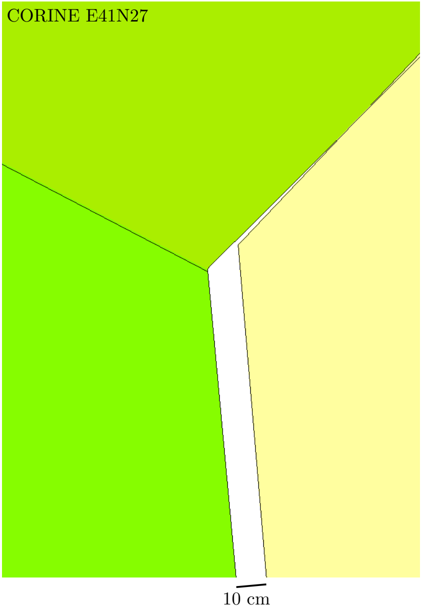
# Big and complex datasets: it quickly gets out-of-control



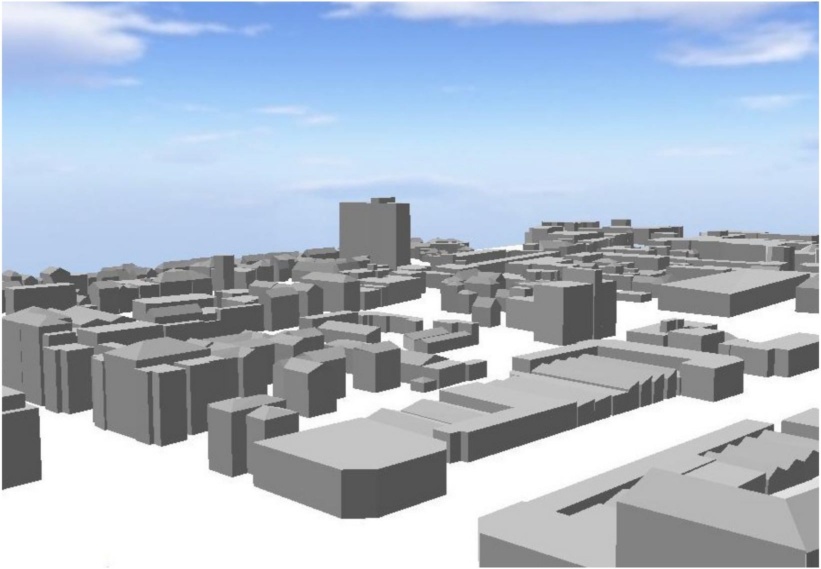
# Big and complex datasets: it quickly gets out-of-control



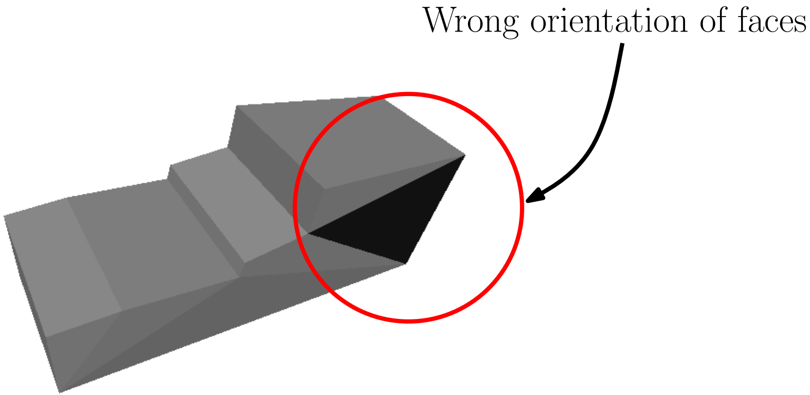
# Big and complex datasets: it quickly gets out-of-control



# Errors in CityGML datasets

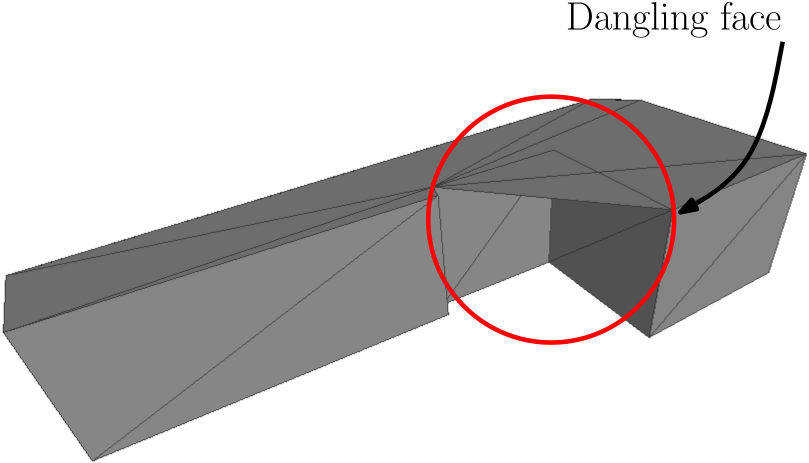


# Errors in CityGML datasets

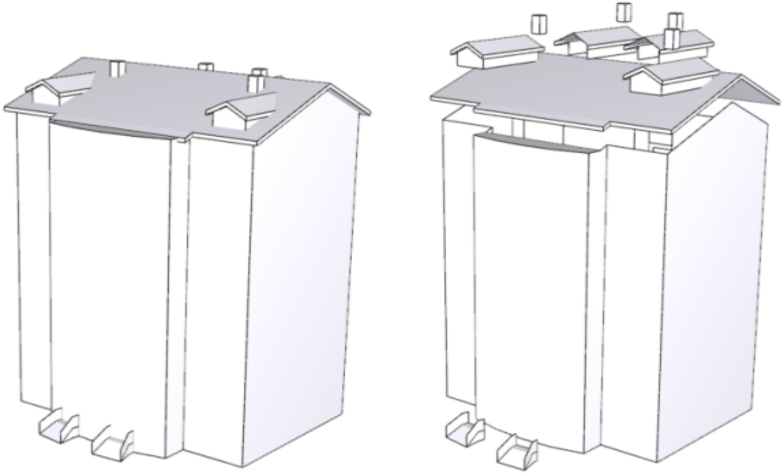


Wrong orientation of faces

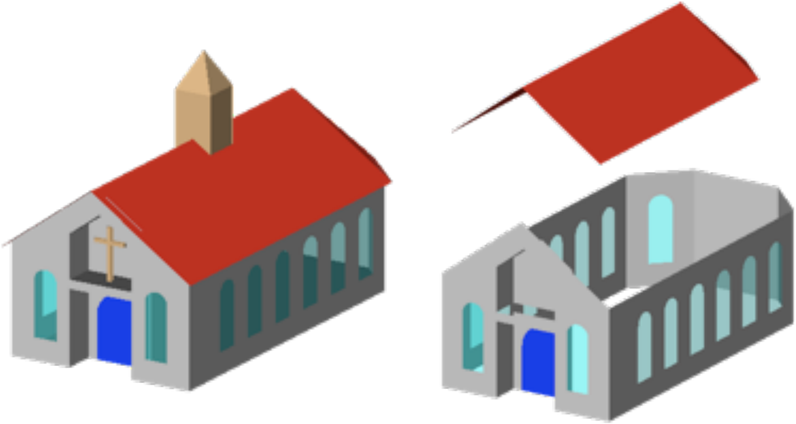
# Errors in CityGML datasets



# Errors in CityGML datasets



# Errors in CityGML datasets



# How do we deal with these problems?

We have solved our own problems by developing 3 prototypes:

- **prepair**: automatic repair of single polygons
- **pprepair**: automatic repair of planar partitions
- **val3dity**: validation and “simple repair” of 3D objects

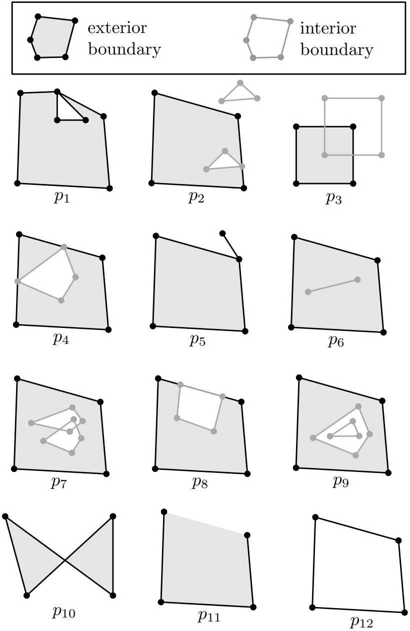
## prepair

repair of single polygons

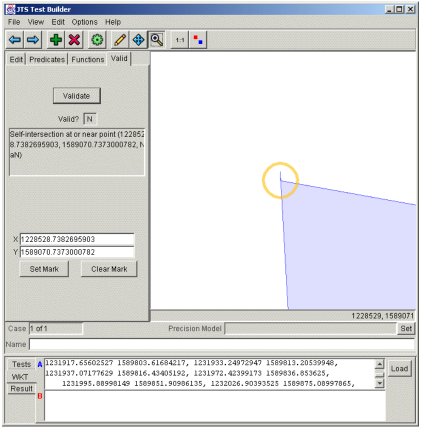
# Validation of a polygon = a solved problem

OGC Simple Features and ISO19107 rules:

1. no self-intersection
2. closed boundaries
3. rings can touch but not overlap
4. no duplicate points
5. no dangling edges
6. connected interior
7. etc



# If it's broken then repair it. But how?



Valid? N

Self-intersection at or near point (1228528.7382695903, 1589070.7373000782, NaN)

X 1228528.7382695903

Y 1589070.7373000782

1228529, 1589071

Case 1 of 1

Precision Model

Name

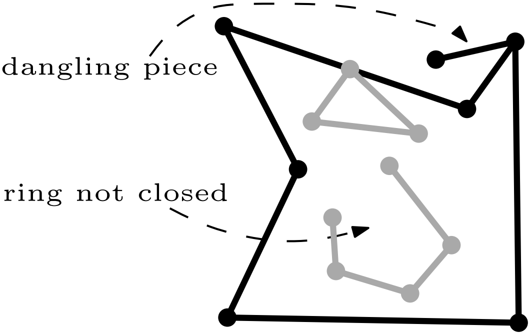
```
1231917.65602527 1589803.61684217, 1231933.24972947 1589813.20539948,
1231937.07177629 1589816.43405192, 1231972.42399173 1589836.853625,
    1231995.88998149 1589851.90986135, 1232026.90393525 1589875.08997865,
```

Errors are highlighted, but not repaired. One has to manually fix them.

# Our approach = constrained triangulation (CT)

Repairing = 3 simple steps:

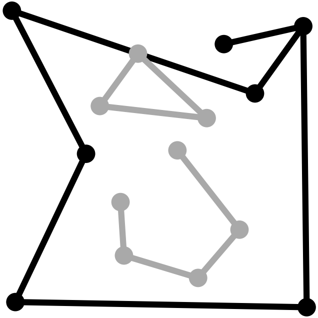
1. CT of input polygon
2. labelling of triangles (*outside* or *inside*)
3. reconstruction of the rings by depth-first search on the dual graph



# Our approach = constrained triangulation (CT)

Repairing = 3 simple steps:

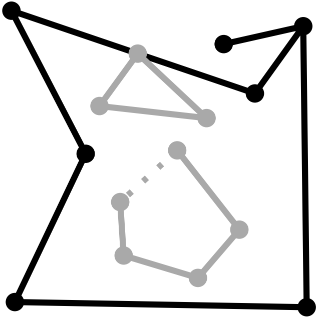
1. CT of input polygon
2. labelling of triangles (*outside* or *inside*)
3. reconstruction of the rings by depth-first search on the dual graph



# Our approach = constrained triangulation (CT)

Repairing = 3 simple steps:

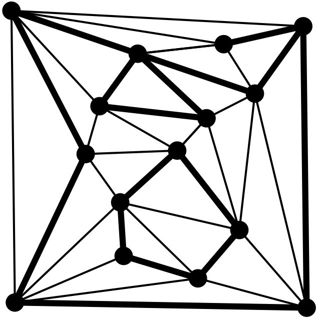
1. CT of input polygon
2. labelling of triangles (*outside* or *inside*)
3. reconstruction of the rings by depth-first search on the dual graph



# Our approach = constrained triangulation (CT)

Repairing = 3 simple steps:

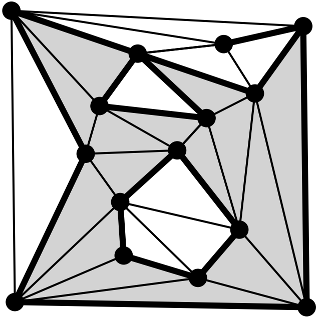
1. CT of input polygon
2. labelling of triangles (*outside* or *inside*)
3. reconstruction of the rings by depth-first search on the dual graph



# Our approach = constrained triangulation (CT)

Repairing = 3 simple steps:

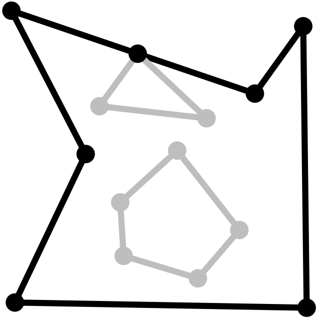
1. CT of input polygon
2. labelling of triangles (*outside* or *inside*)
3. reconstruction of the rings by depth-first search on the dual graph



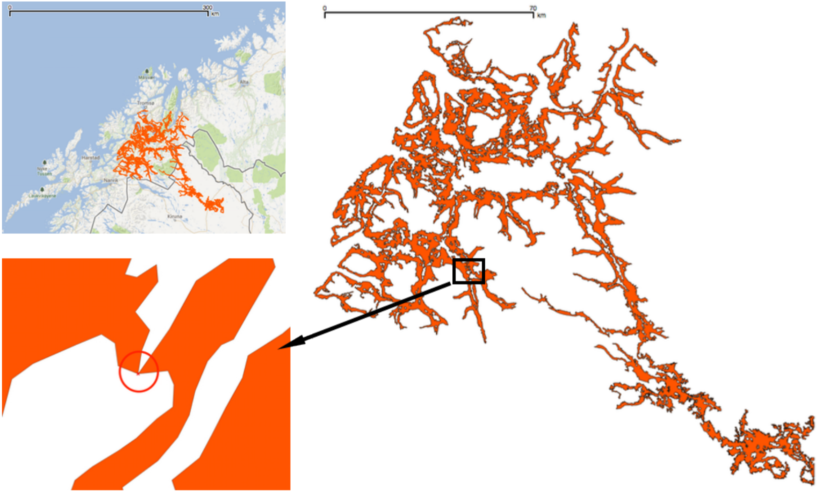
# Our approach = constrained triangulation (CT)

Repairing = 3 simple steps:

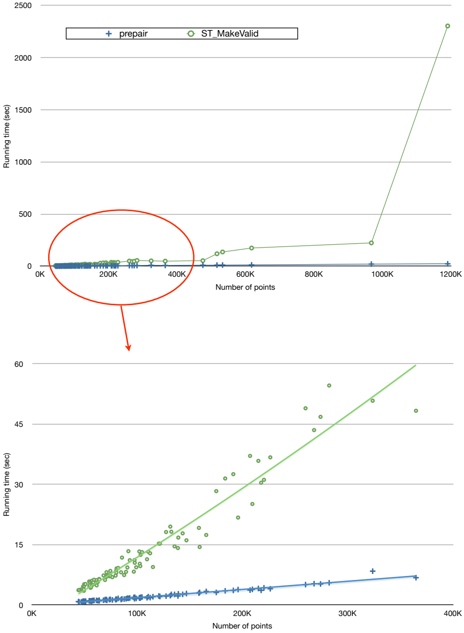
1. CT of input polygon
2. labelling of triangles (*outside* or *inside*)
3. reconstruction of the rings by depth-first search on the dual graph



# Faster than PostGIS by a factor 3–4 in practice



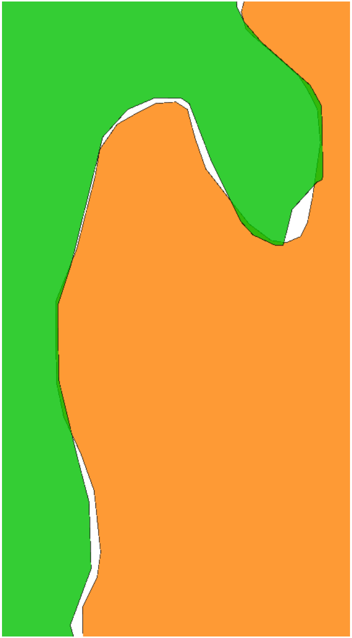
# Faster than PostGIS by a factor 3–4 in practice



# **pprepair**

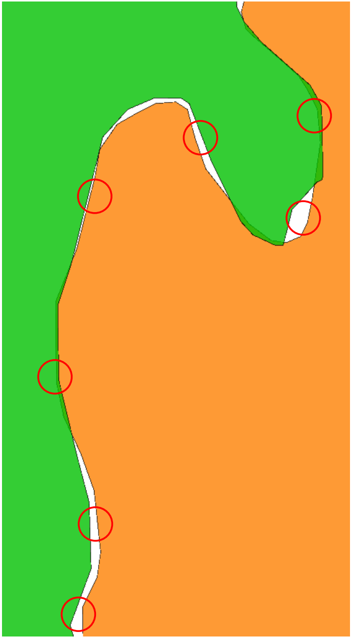
## repair of planar partitions

# One “common” repairing solution: snapping



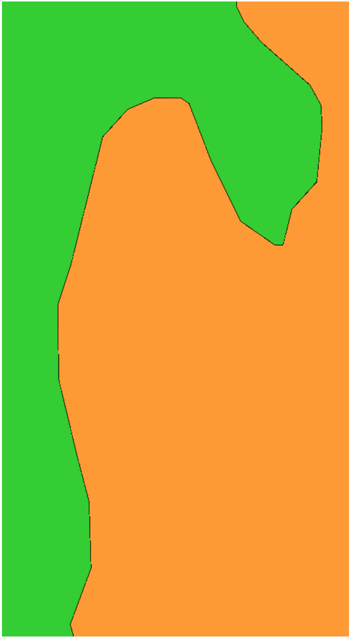
- Tolerance (*threshold*) is used for *snapping* vertices
- Tolerance based on scale of datasets
- Works fine for *simple* problems

# One “common” repairing solution: snapping



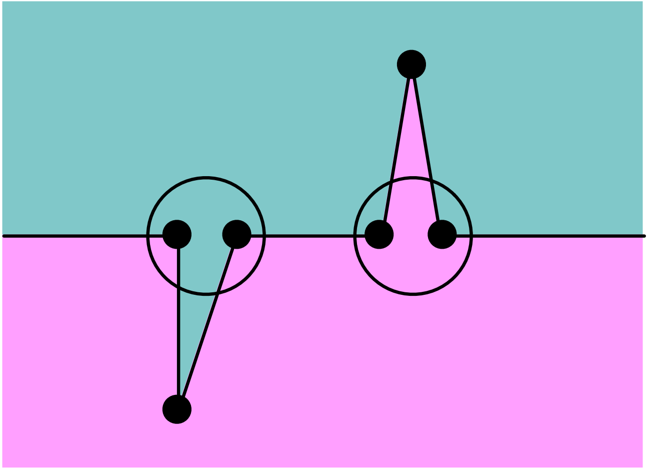
- Tolerance (*threshold*) is used for *snapping* vertices
- Tolerance based on scale of datasets
- Works fine for *simple* problems

# One “common” repairing solution: snapping



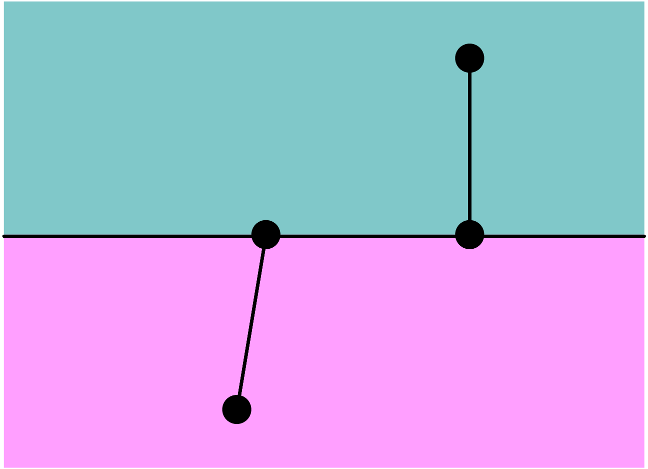
- Tolerance (*threshold*) is used for *snapping* vertices
- Tolerance based on scale of datasets
- Works fine for *simple* problems

# Snapping is error-prone and “dangerous”



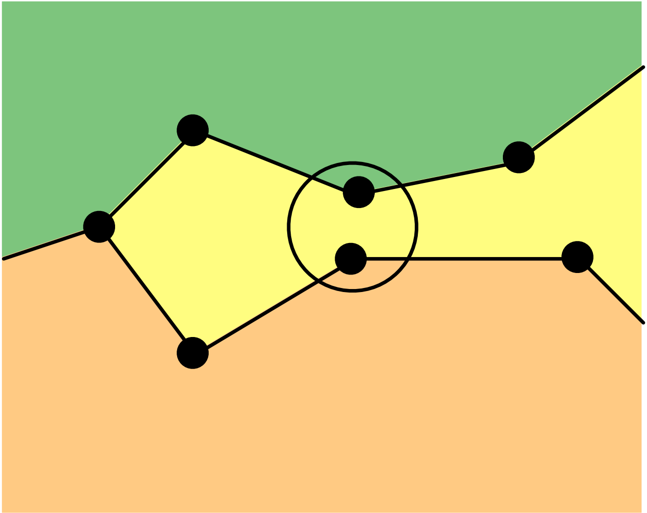
Spikes and punctures can create invalid polygons

# Snapping is error-prone and “dangerous”



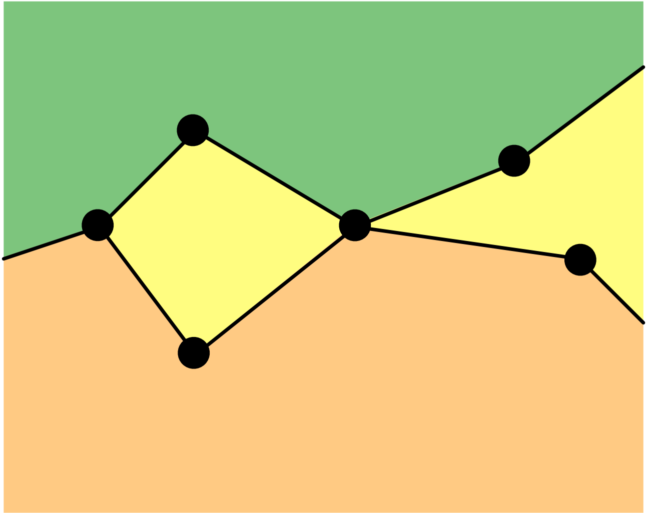
Spikes and punctures can create invalid polygons

# Snapping is error-prone and “dangerous”



Splitting of polygons into several polygons

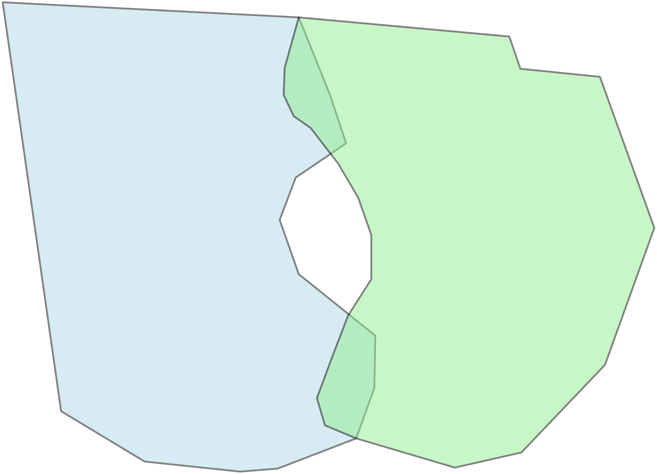
# Snapping is error-prone and "dangerous"



Splitting of polygons into several polygons

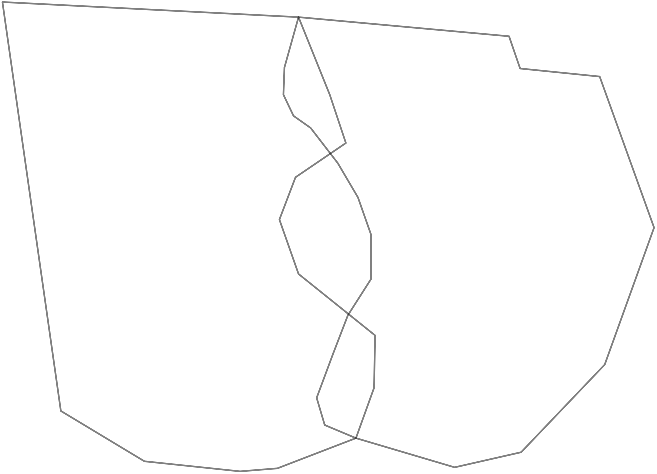
# Our solution = constrained triangulation (CT)

1. Construct CT of input polygons
2. Label each triangle with label of its polygon
3. Problems = triangles with no label or > 1 labels
4. Repair gaps/overlaps *locally* by changing labels



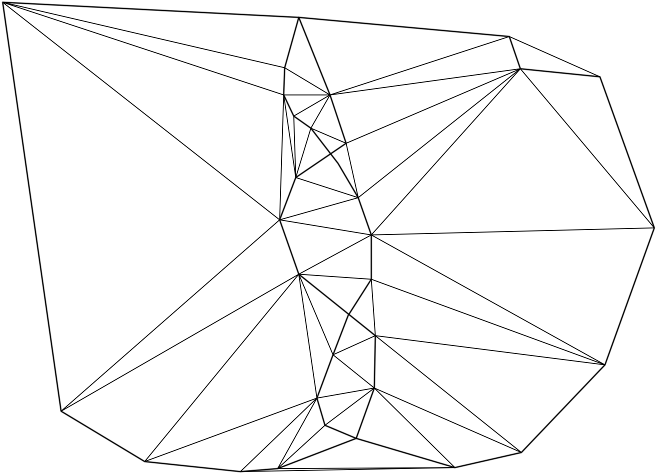
# Our solution = constrained triangulation (CT)

1. Construct CT of input polygons
2. Label each triangle with label of its polygon
3. Problems = triangles with no label or > 1 labels
4. Repair gaps/overlaps *locally* by changing labels



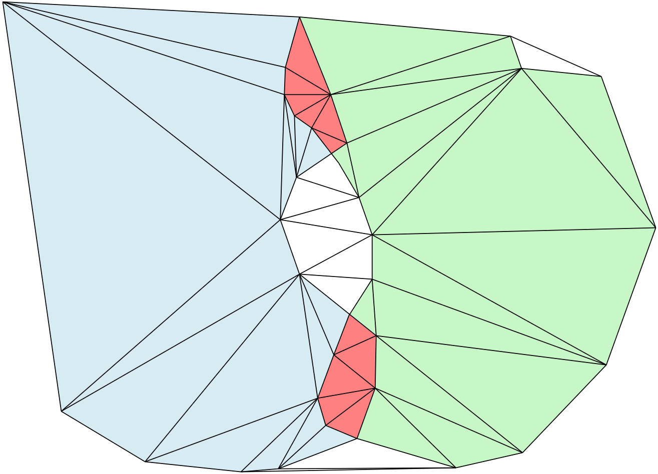
# Our solution = constrained triangulation (CT)

1. Construct CT of input polygons
2. Label each triangle with label of its polygon
3. Problems = triangles with no label or > 1 labels
4. Repair gaps/overlaps *locally* by changing labels



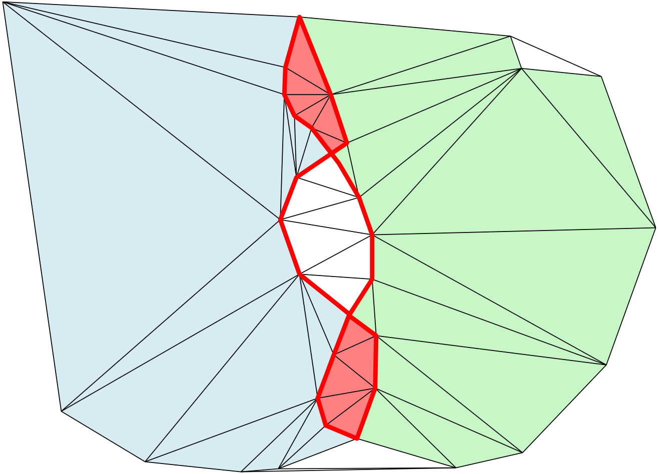
# Our solution = constrained triangulation (CT)

1. Construct CT of input polygons
2. Label each triangle with label of its polygon
3. Problems = triangles with no label or > 1 labels
4. Repair gaps/overlaps *locally* by changing labels



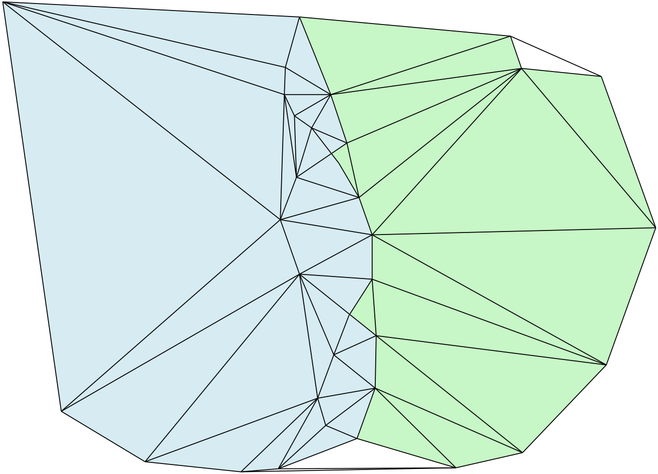
# Our solution = constrained triangulation (CT)

1. Construct CT of input polygons
2. Label each triangle with label of its polygon
3. Problems = triangles with no label or > 1 labels
4. Repair gaps/overlaps *locally* by changing labels



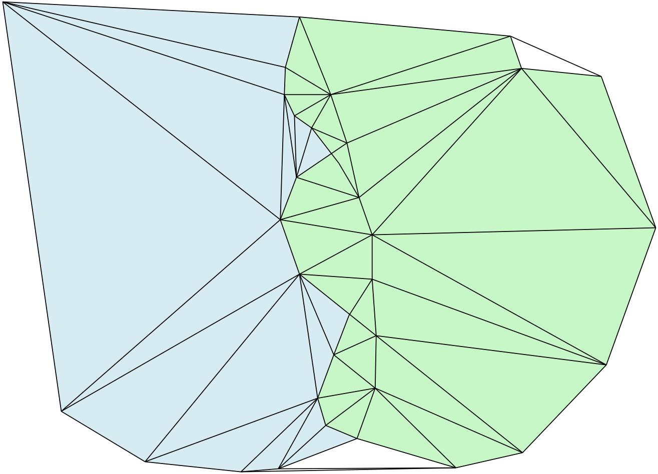
# Our solution = constrained triangulation (CT)

1. Construct CT of input polygons
2. Label each triangle with label of its polygon
3. Problems = triangles with no label or > 1 labels
4. Repair gaps/overlaps *locally* by changing labels



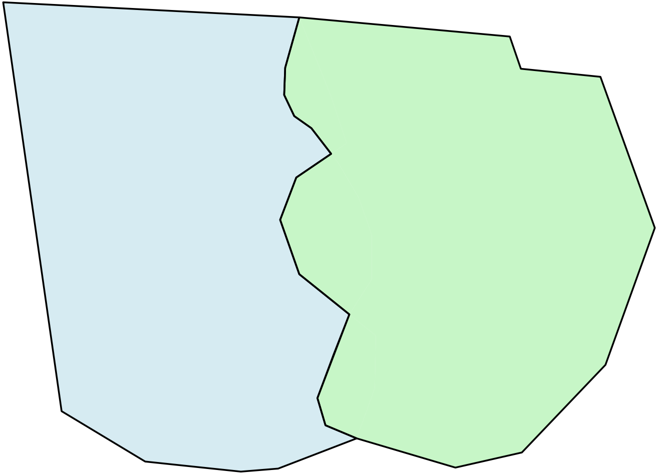
# Our solution = constrained triangulation (CT)

1. Construct CT of input polygons
2. Label each triangle with label of its polygon
3. Problems = triangles with no label or > 1 labels
4. Repair gaps/overlaps *locally* by changing labels

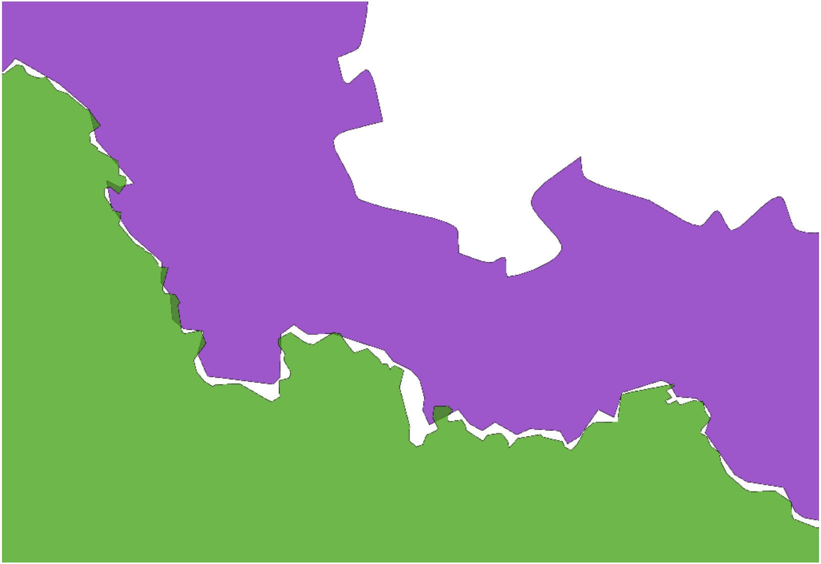


# Our solution = constrained triangulation (CT)

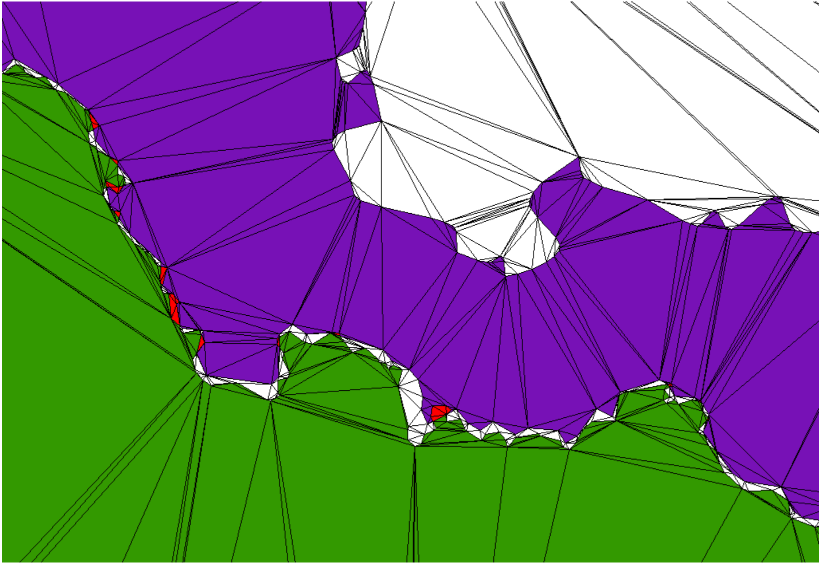
1. Construct CT of input polygons
2. Label each triangle with label of its polygon
3. Problems = triangles with no label or > 1 labels
4. Repair gaps/overlaps *locally* by changing labels



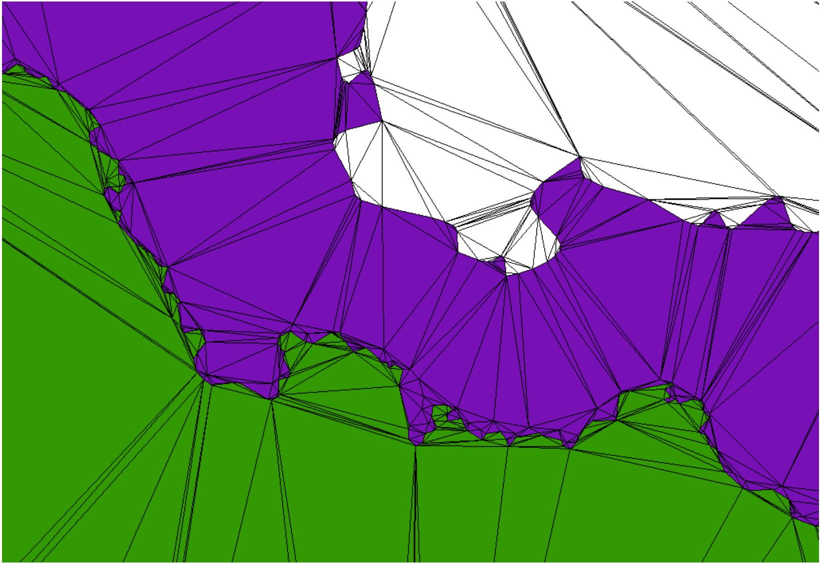
# Local control with simple rules



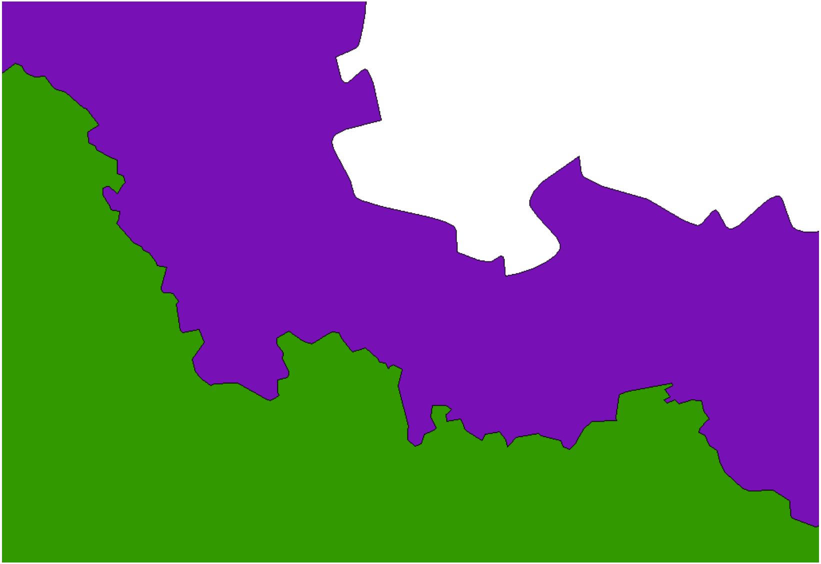
# Local control with simple rules



# Local control with simple rules

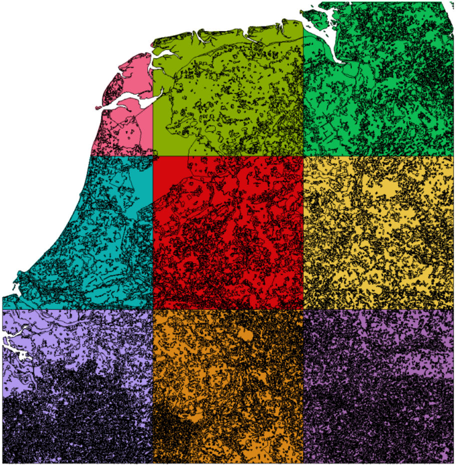


# Local control with simple rules



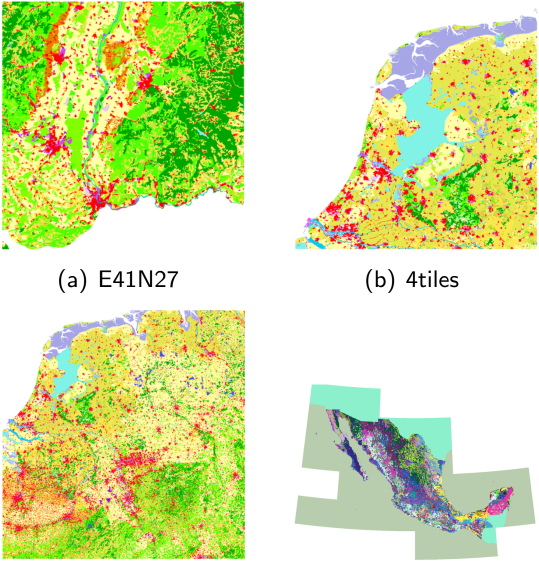
# Local control with simple rules

# Experiments with large real-world datasets



CORINE dataset: land-use of Europe
tiles of 100km X 100km

# Experiments with large real-world dataset



(a) E41N27

(b) 4tiles

(c) 16tiles

(d) Mexico

# Experiments with large real-world datasets

| | # polygons | # pts | # pts<br>largest polygon | avg # pts<br>per polygon |
|---|---:|---:|---:|---:|
| **E41N27** | 14 969 | 496 303 | 26 740 | 34 |
| **4tiles** | 4 984 | 365 702 | 16 961 | 75 |
| **16tiles** | 63 868 | 6 622 133 | 95 112 | 104 |
| **Mexico** | 26 866 | 4 181 354 | 117 736 | 156 |

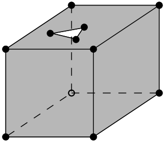
# Comparison with other GIS packages

| | pprepair | | |
|---|---|---|---|
| | # pts | memory | time |
| **E41N27** | 496 303 | 145 MB | 19s |
| **4tiles** | 365 702 | 116 MB | 17s |
| **16tiles** | 6 622 133 | 1.45 GB | 4m47s |
| **Mexico** | 4 181 354 | 1.01 GB | 3m31s |

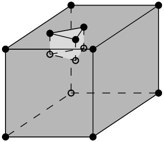
# val3dity

validation of 3D solids

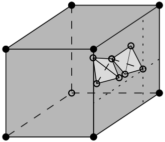
# ISO 19107 rules also in 3D



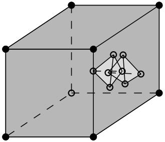
$s_1$
invalid



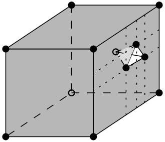
$s_2$
valid



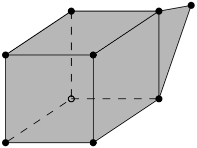
$s_3$
valid



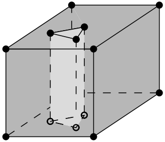
$s_4$
invalid



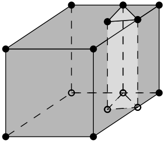
$s_5$
invalid



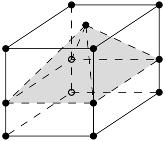
$s_6$
invalid



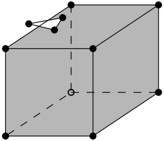
$s_7$
valid



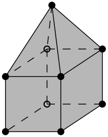
$s_8$
invalid



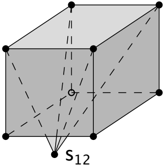
$s_9$
invalid



$s_{10}$
invalid



$s_{11}$
valid



$s_{12}$
invalid

# ISO 19107 rules also in 3D

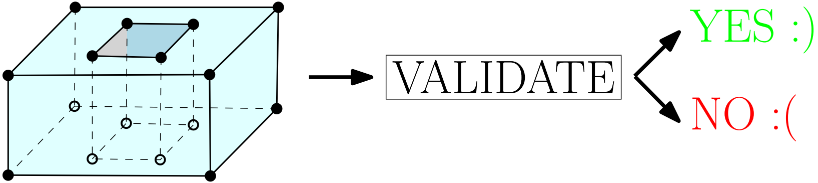
1. distinct vertex
2. closedness of the rings of every surface
3. orientation of points within a surface (with inner rings)
4. planarity of surfaces
5. non-self intersection of surfaces
6. non-overlapping inner rings on a surface
7. orientation of normal vectors
8. "watertightness" of every shell
9. "connectedness" of the interior
10. how inner/outer shells interact with each others
11. ...

# ISO 19107 rules also in 3D

None of the (commercial) tools are ISO compliant

# The implementation

- As compliant to ISO 19107 as possible
- Use of CGAL: robust and fast
- C++
- Be kind to the user
- Try to automatically repair invalid solids (work in progress)

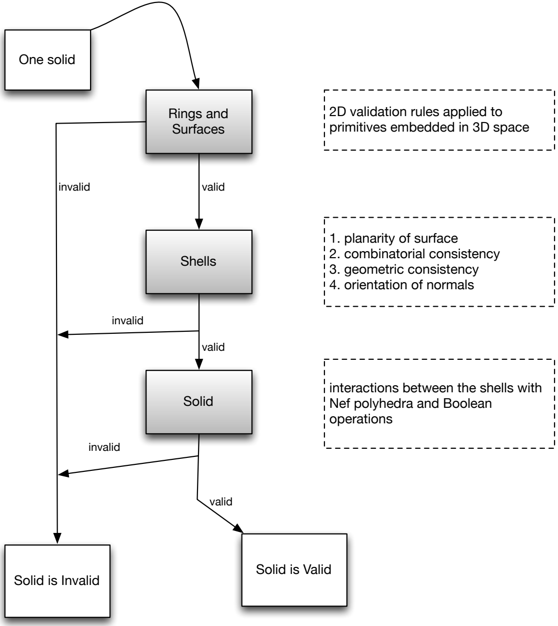


VALIDATE

YES :)

NO :(

# Validation is performed hierarchically



One solid

Rings and Surfaces

Shells

Solid

Solid is Invalid

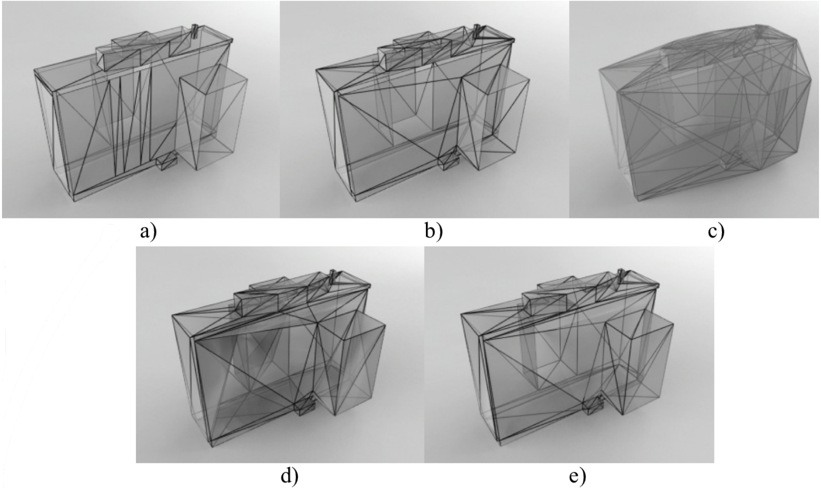
Solid is Valid

2D validation rules applied to primitives embedded in 3D space

1. planarity of surface
2. combinatorial consistency
3. geometric consistency
4. orientation of normals

interactions between the shells with Nef polyhedra and Boolean operations

# Automatic repair = our current work



# Thanks for your attention

www.github.com/tudelft-gist/ prepair
pprepair
val3dity