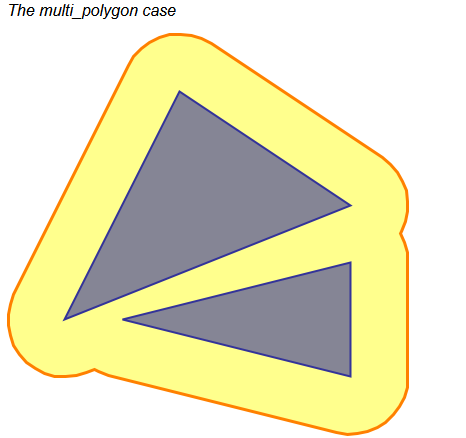
**joinAll**

Possible replacement idea, use buffer which can combine several polygons into one by offsetting them and then shrinking them by the same amount.



Otherwise I have not been able to reproduce am issue with joinAll so far, even with what was supposed to be the data associated with the bug report, though in the case of the bug report the data was in lat/long coordinates so maybe that’s the reason.

the logic for joinAll is as follows:

1. Compile a list of areas of each polygon
2. Construct an adjacency matrix n x n where n is the number of polygons
   1. Call join on each polygon against the next
   2. If that returns a value, i.e. the polygons can be added together to make a single polygon, than put a 1 in the cell at [I,j] and [j,i]
      1. join converts an OpenStudio polygon (list of points) to a BoostRing which is a local typedef using the boost type ring.
      2. The two rings are unioned together, the result is checked for spikes
      3. The result is then checked to make sure there is one polygon returned and that it has no inner paths, that the polygon has an area and the area is greater than a minimum size and that the polygon is not self intersecting. If any of these conditions are not met then the join returns null
3. Convert the adjacency matrix into one or more lists of connected components (findConnectedComponents)
   1. For some reason a copy of the matrix is made but the value is tested for being <= tolerance. I suspect this is left over from a previous iteration where the matrices stored the distance between each polygon
   2. Iterate over the number of polygons (i)
   3. If the polygon index has already been added to the list then skip it
   4. Create a new components list and add the polygon index to it.
   5. Iterate over remaining polygons (j starting at i+1)
   6. If the cell [I,j] ot [j,i] is non zero then add j to the component list
   7. Add the component list to the oputput
   8. Repeat from (b) until

1. Sort each list of connected components by descending area
2. Call boolean join on the connected components. At this point the components list is not in the order they need to be joined in to ensure that every union call returns one polygon, so instead the join is performed inside an outer loop 1 to number of components to ensure that everything is checked.

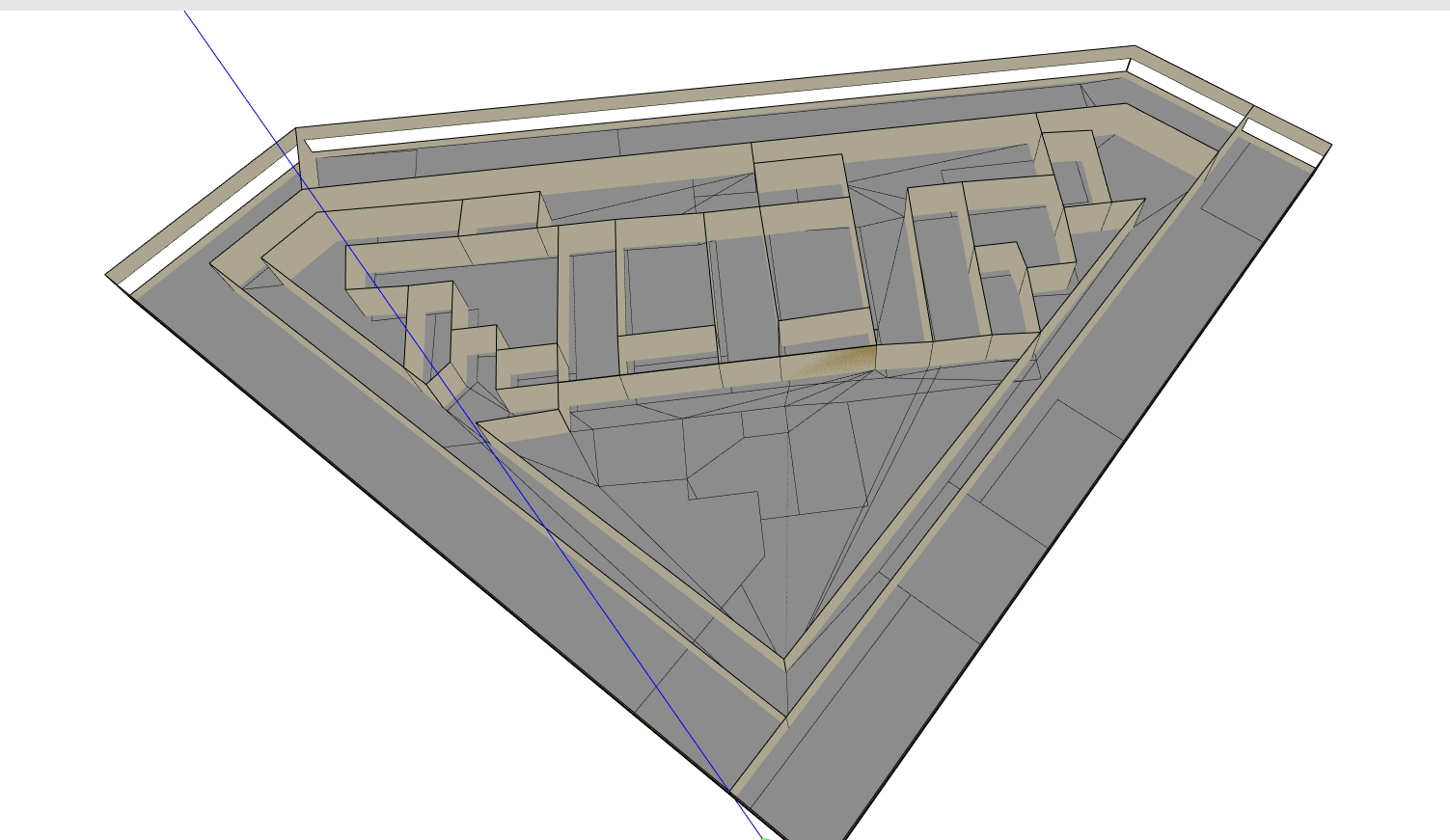
There are reported issues with joinAll not producing the correct results and in fact sometimes not being deterministic, so how can joinAll fail?

1. The adjacency test. This uses the boost library join method to determine whether two polygons are adjacent by testing to see if calling join with those two polygons returns a single polygon as a result. There is no tolerance with join so the most likely cause of failure here is numerical accuracy.  
     
   A solution to this would be to inflate the polygons by the tolerance amount, and then deflate them by the same amount after the operation has completed. Another solution would be to build a canonical set of points, compare the points in the two polygons, if the points are within the tolerance of each other then adjust one of the points.
2. Creating the components list. Apart from what looks like some legacy code still being used there is nothing in this method that can cause a problem, the logic is sound and the process itself is not particularly complicated

**'Surface Shattering' (3424)**

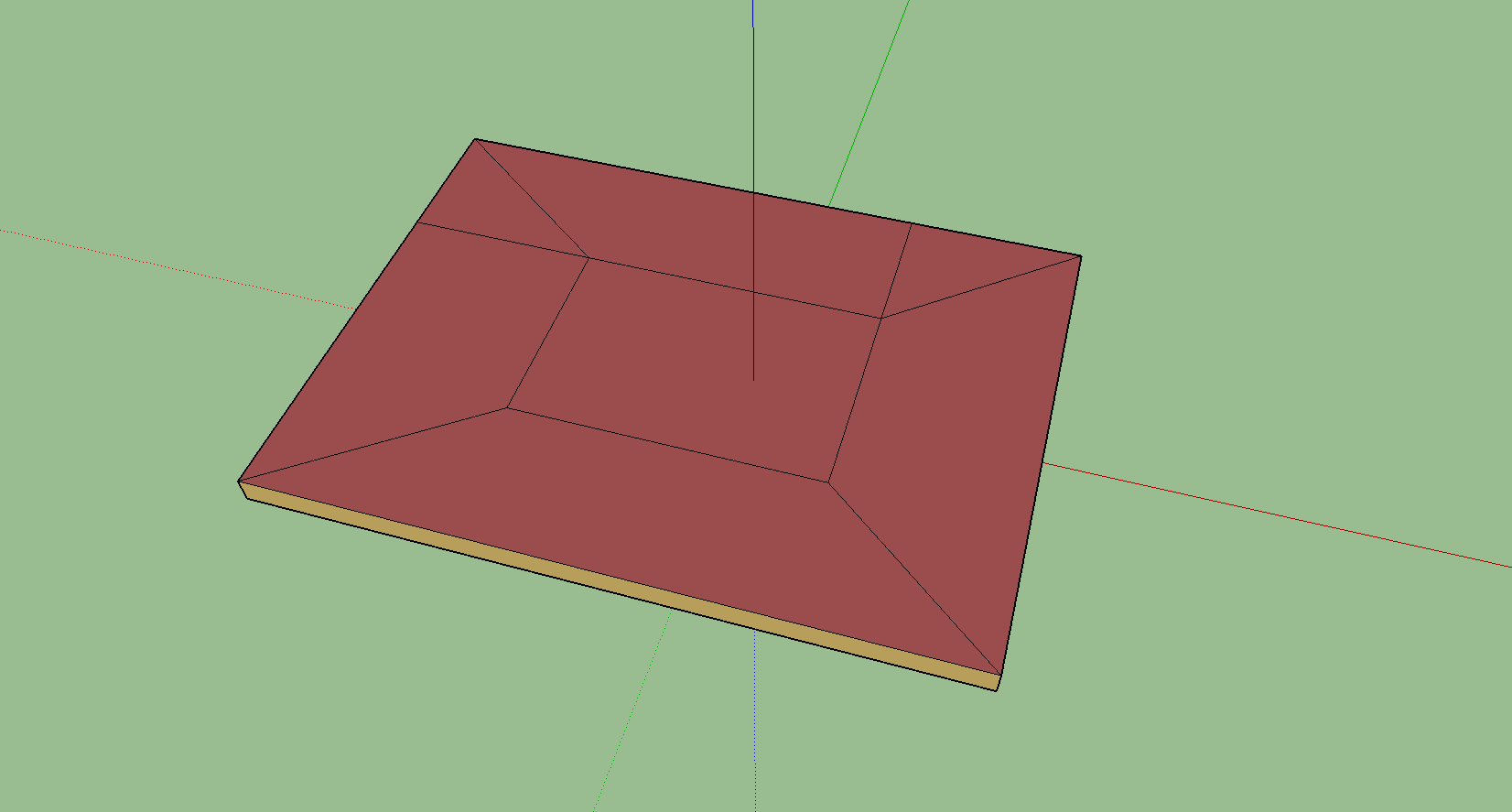
This best describes models where the intersected surfaces have too many triangles, for example in the screenshot below

surfaces have been triangulated because the contain one or more holes or because they are concave and then have been intersected with other surfaces later on. This produces too many surfaces and also makes the intersection and matching process take much longer because there are more surfaces to intersect and match.

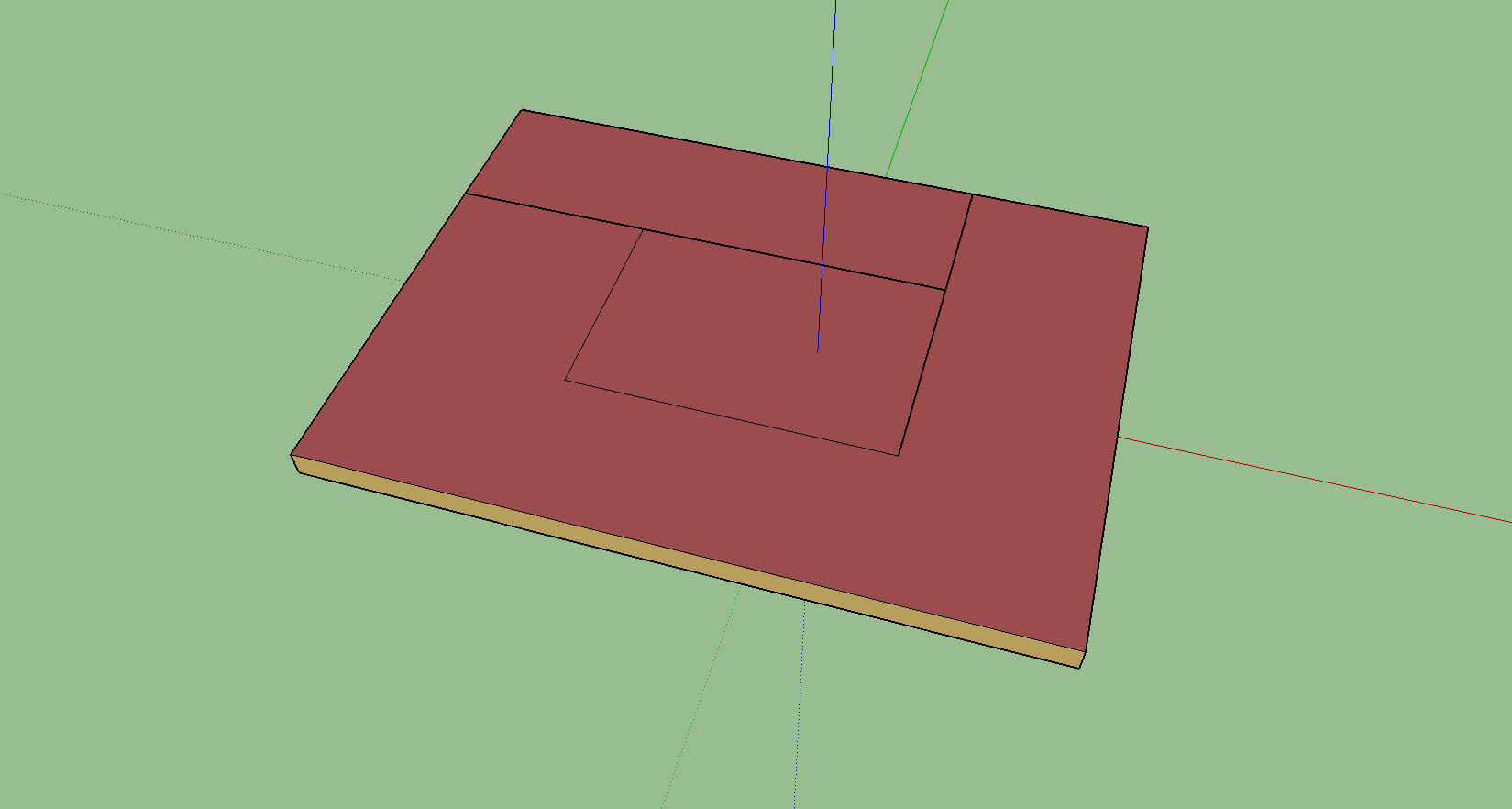


This is also reported in issue 3424, <https://github.com/NREL/OpenStudio/issues/3424>, In this case an example is given where there is a large single space on the bottom story and two smaller spaces in the 2nd story. The outcome from intersecting the surfaces is dependent on the order in which the surfaces are intersected. If the space on the second story is intersected first then the result is a surface with a hole, something that Energy Plus does not allow and so that surface is triangulated into 4 surfaces. These surfaces are then intersected with the other surface on floor 2.

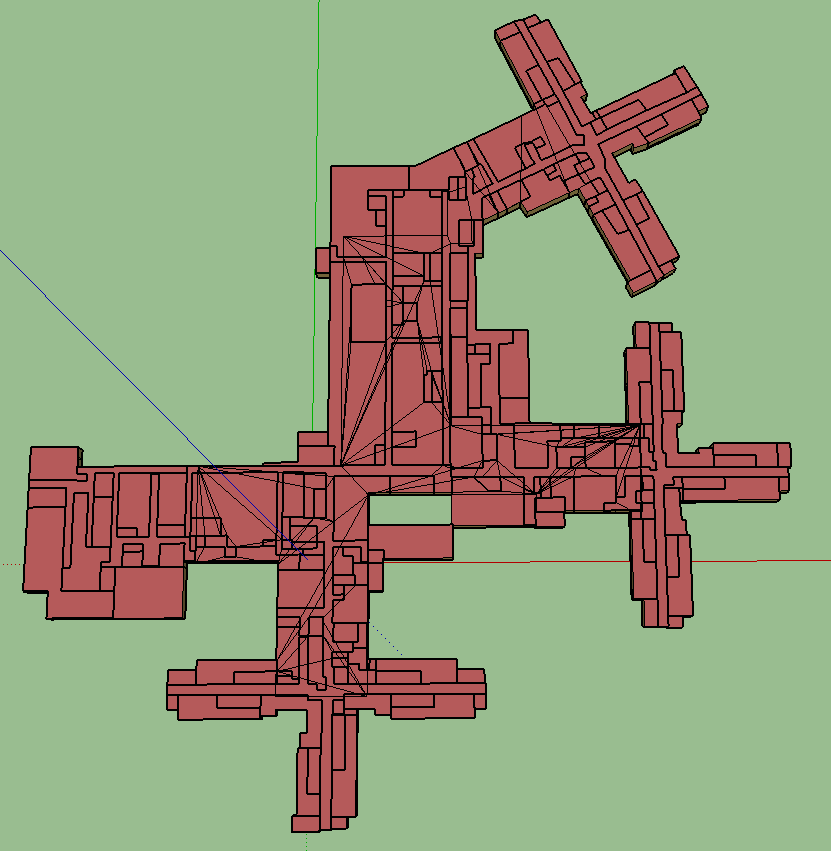
However if the second surface is intersected first the result is a concave surface (\*) which can then be intersected with the remaining surface without needing any further decomposition. The results in the first cased is 7 surfaces, and in the second case 3 surfaces.

[](https://user-images.githubusercontent.com/46730121/52763218-0aab1c80-306f-11e9-8ea0-31b36d6f1514.PNG)

First intersection produces a polygon with a hole which is then decomposed into 4 surfaces resulting in 7 surfaces in all

[](https://user-images.githubusercontent.com/46730121/52763220-0bdc4980-306f-11e9-80d0-2b6b62d54ab7.PNG)

The first intersection does not result in a hole so no decomposition is needed resulting in 3 surfaces

[](https://user-images.githubusercontent.com/46730121/52762991-0f230580-306e-11e9-9886-2cf1e64ebe0b.PNG)

A more complex model showing 'shattering'

The solution in this case is to delay the decomposition of surfaces until all intersections have been completed. Why be concerned with whether the surface is valid for Energy+ if the surfaces have not all been generated yet? As a result the order in which the polygons are intersected will be irrelevant and the number of polygons to intersect will be reduced.

In the current library implementation the spaces are sorted by floor area, there doesn't seem to be a particularly good reason to do this, maybe it is to ensure the spaces are intersected in the same order every time. For the purposes of testing I removed the line of code that sorts the spaces so now we can verify that yes if the space order is 1,2,3 we get one set of surfaces and if the order is 1,3,2 we get a different set of surfaces.

The biggest issue with this solution is that Surface.computeIntersection creates new surfaces as surfaces are intersected. Surfaces are IDF objects have a vertex list and hence can't have holes. This will require a quite extensive rewrite of the affected areas

* At some point all surfaces should be converted to BoostPolygons.
* There should be some way to link the BoostPolygon back to the original surface, and hence space
* Collect all the polygons belonging to a space together
* Intersect Space/Space, Surface/Surface as we do now but we do Space/Space, Polygon/Polygon.
* When intersection has finished every space has a list of surfaces, every surface has a list of polygons it has been divided into, the first polygon updates the surface, the remaining polygons create new surfaces
* Before this step we look for any polygon with a hole or concave and decompose it.

**Intersection**

Intersection and surface matching work together, intersection decomposes surfaces into common areas between overlapping surfaces, surface matching finds these common areas and pairs the surfaces together.

However it seems obvious that intersection has already paired the surfaces together so there is no need for a separate surface matching.

Intersection works like this:

* Call boost::geometry::intersection to return the common area(s) between two surfaces (A and B)
* If there is at least one common area between the two surfaces, subtract the common areas from surface A and surface B
* In the result from intersection return the first common area and two lists containing the remaining common areas and the results of subtracting the common areas from the original surface
* The return result contains:
  + The common area for surface A and the common area for surface B (this is actually the same polygon)
  + A list of polygons for surface A which contains the remaining common areas (if any) and the result of subtracting the common area(s) from surface A
  + A list of polygons for surface B which contains the remaining common areas (if any) and the result of subtracting the common area(s) from surface B
* The first common area for surface A and B are used to update the vertices for the original surface
* The remaining polygons are used to create new surfaces for Space A and Space B.

What we do know at this stage is that the common areas will never be subdivided again, they are the final surfaces. Therefore we know that these surfaces are adjacent surfaces and therefore we know that there is no need to do surface matching on these surfaces. There is even a comment in the code



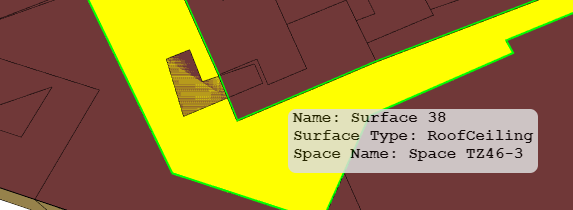
But no explanation as to why this was decided.

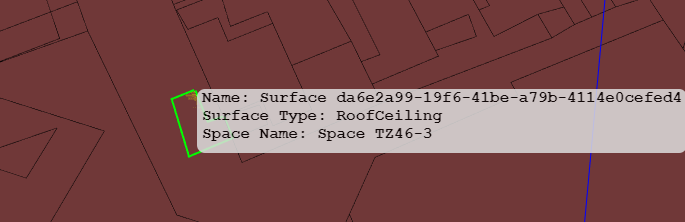
Any surface that has an adjacent surface already defined can be skipped for surface matching.

Intersection gets stuck in a loop. Using the test model two\_stories\_pre\_intersect.osm, the model contains 92 spaces over three stories with a total of 986 surfaces, not a huge model and it shouldn’t take long to do the intersection and matching.

Intersection never completes.

Currently intersecting space TZ47-103 against TZ46-3. When paused, intersectSurfaces had generated 2258 surfaces for TZ46-3, so obviously something had run off the rails here. Surface 38 of TZ46-3 was observed to have what appears to be two overlapping surfaces, see below





Surface 38 and surface da6e2a99….

Further investigation is needed to determine whether the cause of the issue is the two overlapping surfaces. Based on the log file output (space names are not logged) surface 3284b3ad-0970-4c14-a35d-cb4ed8673cc5 from space TZ47-103 is being intersected and the routine never gets past this point. What appears to happen is the following

1. Surface 3284b3ad-0970-4c14-a35d-cb4ed8673cc5 intersects with Surface 129 generates Surface 130
2. Surface 10 intersects with Surface 130 generating surface 131
3. Surface 3284b3ad-0970-4c14-a35d-cb4ed8673cc5 intersects with Surface 131 generates Surface 132
4. Surface 10 intersects with surface 132 generating surface 133
5. Surface 3284b3ad-0970-4c14-a35d-cb4ed8673cc5 intersects with surfaces 133 generating surface 134
6. Surface 10 intersects with Surface 134 generating Surface 135
7. Surface 3284b3ad-0970-4c14-a35d-cb4ed8673cc5 intersects with surface 135 generating surface 136
8. Surface 10 intersects with Surface 136 generating surface 137.

Etc.

Originally Surface 10 is intersected with Surface 38 to generate Surface 129 (10 and 38 are surfaces in the model) - that seems to be the starting point of the mess

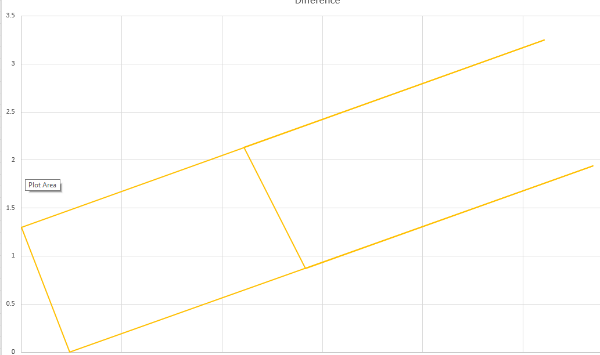
After removing all spaces from the space list that contain intersecting surfaces and re-running the routine now blocks on Space TZ46-81 and Space TZ47-91. TZ47-91 has two downward facing surfaces (Surface 25 & Surface 40762b18) and TZ46-81 has two upward facing surfaces (Surface 48 & Surface fe26e6e5)

**Line 1152 of Surface.cpp an assumption is made that faceVertices needs to be reversed to be CW and otherFaceVertices is already CW - is this a valid assumption?**

To summaries the findings - its kind of hard to explain…

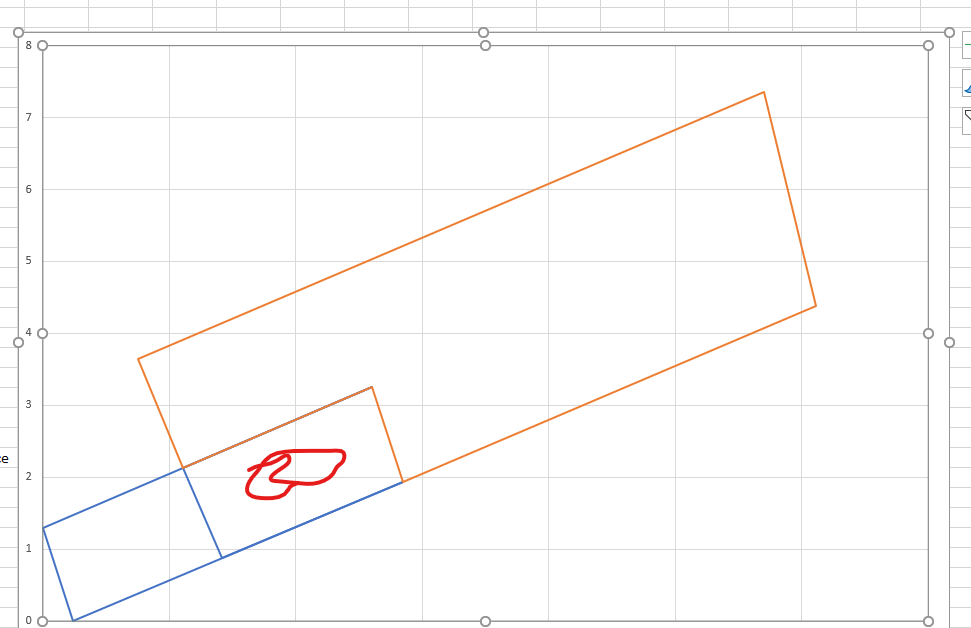
Intersecting Space TZ46-81 with Space TZ47-91

Intersect Surface 48 with Surface 40762b18-ffb0-4ee2-8d38-38cfceeabac5 results in this polygon when the intersection result is subtracted from Surface 48

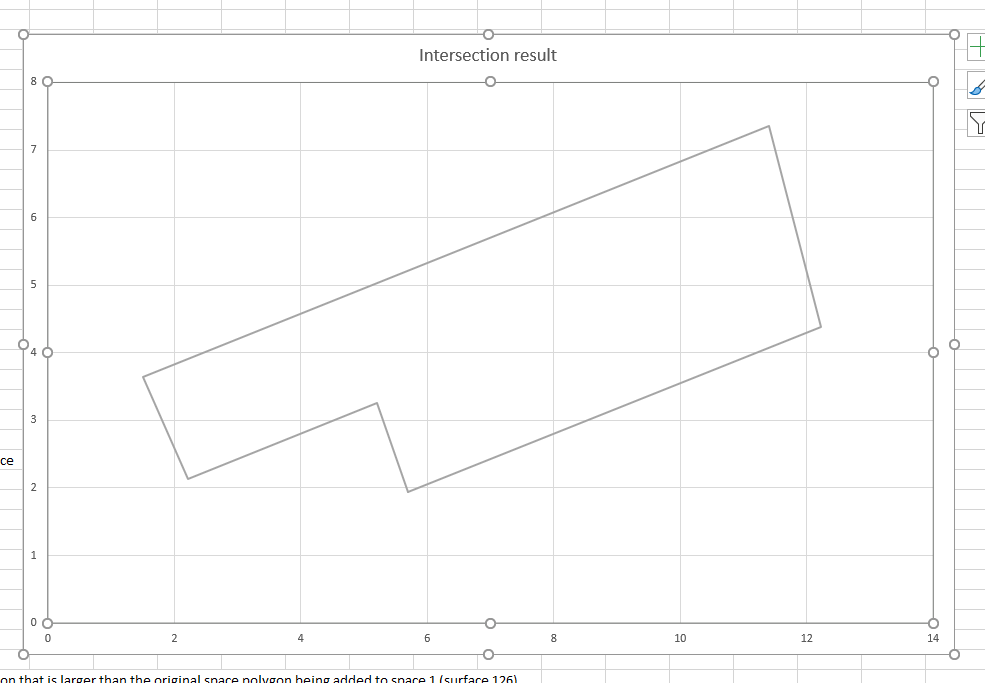


This an invalid polygon, it has spikes but the spikes are not removed by the remove spikes function. It is used to generate Surface 125

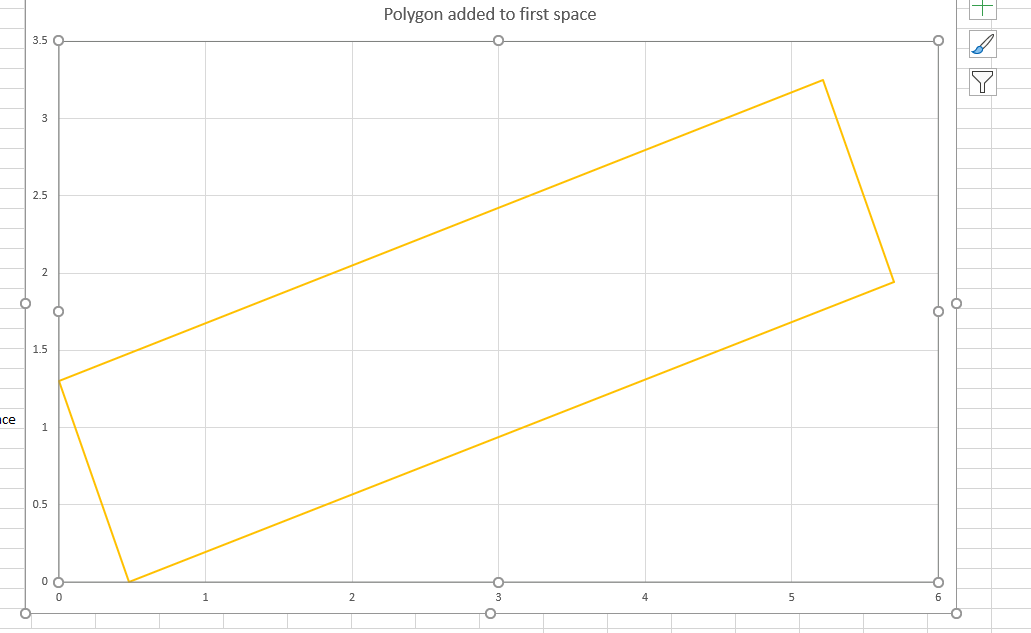
Surface 125 is added to Space TZ46-81 and is then intersected with Surface 25, these are the starting polygons



There is no polygon where the red squiggle is, and this is the result of the intersection



Which is the identical to the second starting polygon and obviously not the common area between the two starting polygons. Below is the polygon that is the ne surface for Space TZ46-81. But this is obviously not the common area between the two polygons



Which creates a polygon identical to the original Surface 48 This is now surface 126 and is intersected once again with Surface 40762b18-ffb0-4ee2-8d38-38cfceeabac5

The timing for intersection with 986 surfaces/81 spaces is 66 seconds (admittedly in the unit tests) which is not really very speedy m(and 274 seconds to match surfaces)

In A Nutshell - what do we have as prototypes?

1. A joinAll that uses boost::buffer to expand polygons before intersecting and shrink them afterwards after all intersection are performed
2. A joinAll that uses boost::buffer to buffer all polygons to combine polygons together
3. JoinAll quits when it detects a hole.
4. Join returns null when a hole is detected. Join is used to determine polygon adjacency. Therefore two adjacent polygons are not considered to be adjacent if adding them together creates a polygon with a hole.
5. A better remove spikes by using boost::buffer to shrink and expand a polygon. Having a polygon with one or more spikes causes intersection to get stuck in a forever loop
6. Surface matching, using a very tight tolerance (default value was 0.001, probably should have been 0.01)
7. Surface 'shattering'. Caused by unnecessary triangulation. Prototyped to demo the effect of delaying surface decomposition until all intersection has been performed
8. Combine intersection and matching. When two surfaces are intersected we get a list of common areas and a list of remaining areas. BY definition the common areas are pairs therefore surface matching has already been performed and there is no need to intersect and match
9. Must validate surfaces. Overlapping surfaces within the same space for example causes intersection to get stuck. Recommend 1) eliminate those spaces from intersection, 2, boolean the overlapping surface into

1. A more reliable joinAll
2. A more reliable removeSpikes
3. Delay of surface decomposition
4. Combine surface intersect/match