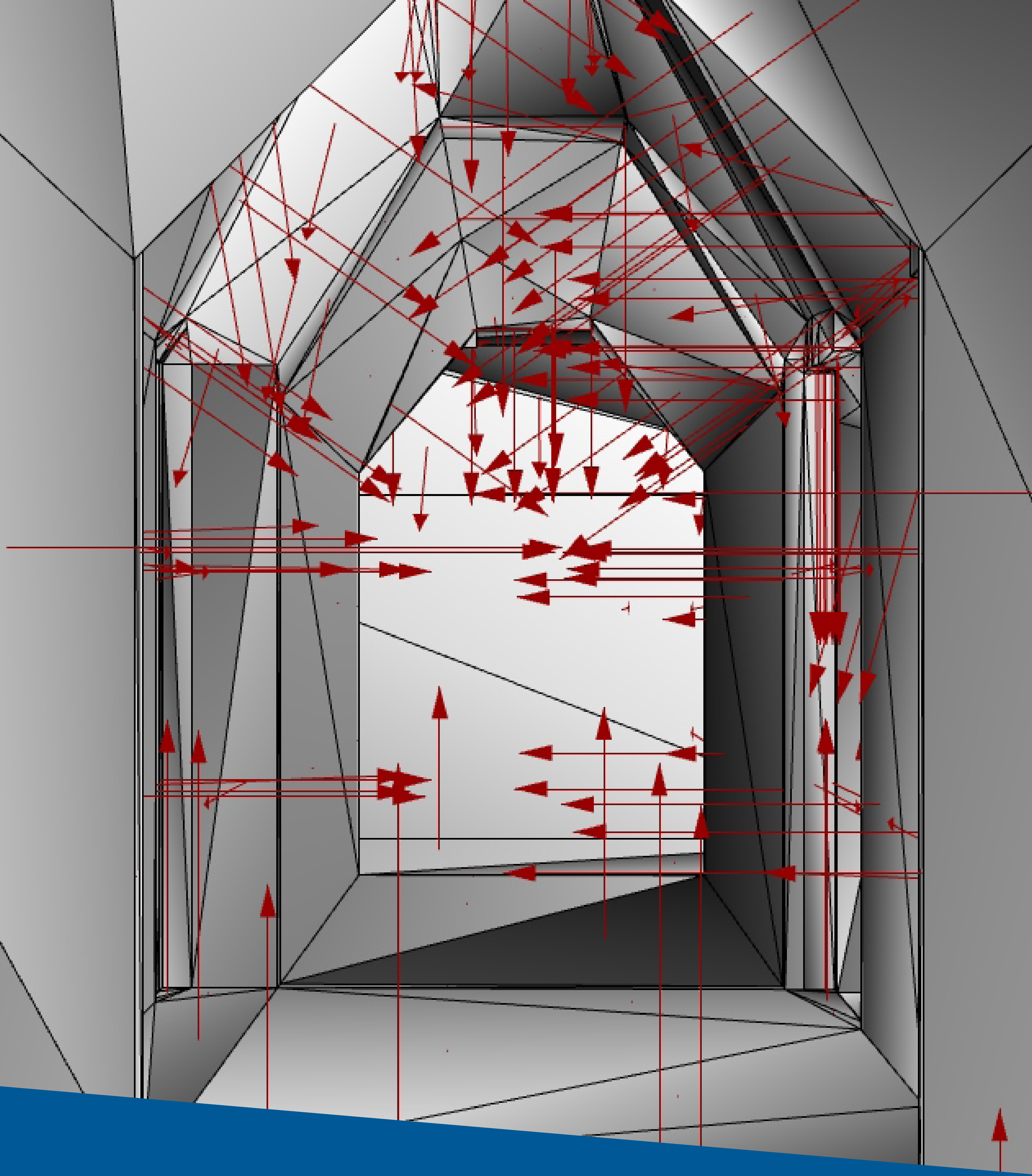


GEO 1004
3D Modelling of the build environment
Assignment 1: Mesh Orientation

Jos Feenstra | 4465768

Code Design

The geo1004_hw01.py file includes, besides a main() function, 4 classes: Vector3, Ray, Triangle, and Mesh. I will explain the script from the bottom to the top.

The main function is kept as clean as possible. It instanciates a Mesh object with the obj file, calls a function to configure the topology of the mesh, calls a third function to flip the orientation of the triangles, and finally writes the current state of this mesh to the path of the second argument. I assume the reading and writing of the obj files speak for themselves, so I will focus on the second and third method: Mesh.set_all_neighbours() and Mesh.orient_triangles().

The Mesh class contains a list of its vertices, as Vector3 instances, and a list of Triangle instances. The Triangle class contains index values of its vertices and neighbours, among other things. After the Mesh.from_obj() method is called, both these lists are instantiated and filled, but the neighbours of the Triangle instances are not yet know.

Mesh.set_all_neighbours() makes the mesh topologically sound. By using dictionaries, the running time of this part of the script is reduced to O(3n) = O(n). It works by storing the edges of every triangle as a "(int, int)" tuple (with int being the vertex index) as the key in a dictionary. By looking up edges in this dictionary, a triangle can find out if their edges are already present, indicating a shared edge. The script uses a "(min(...), max(...))" construction to ensure two triangles can find each other, no matter their clockwise or counter clockwise orientation. Moreover, the dictionary also keeps track if this "min() max()" flip had to be made. This in turn can be used to know if the two triangles are oriented in the same way (cw or ccw), or are oriented in different ways. This is very useful, as it enables us to flip the entire mesh in the same orientation if so desired. The only thing stopping us is the fact that we do not yet know if this will result in all normals pointing inwards, or outwards.

This is where Mesh.orient_triangles() comes in. It does two things:

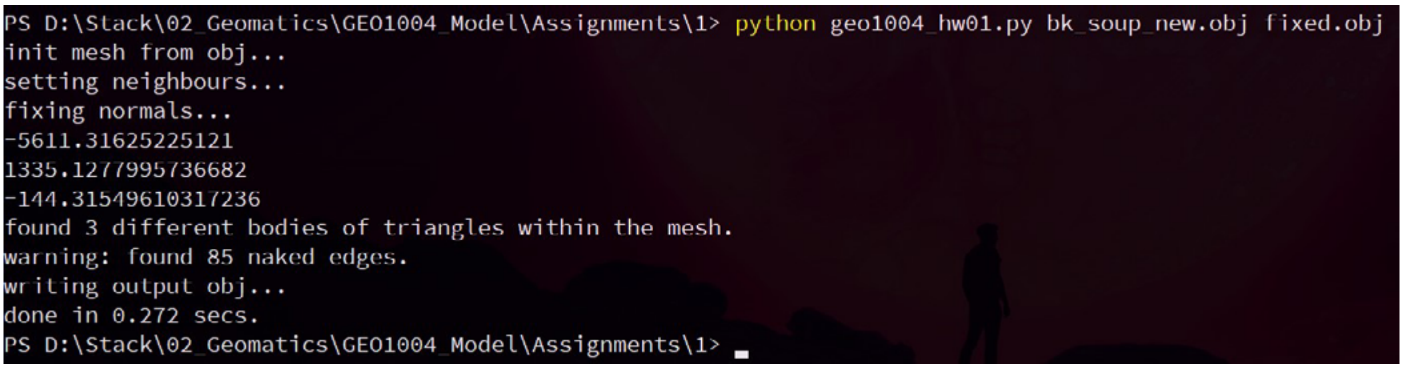
One: Separate the mesh into the different connected groups of triangles, I started calling 'bodies'. This works by keeping track of an "unvisited" set. If there are still unvisited triangles left after the graph traversing part of the code is done, it means there are still unchecked triangles out there, and the algorithm will repeat itself with these leftover triangles to form a new body.

Two: Per body, figure out the correct orientation of 1 starting triangle. The method Mesh.calc_initial_orientation() is used to find this starting triangle and orientation. It asks for the leftover, previously unvisited triangles. With these triangles, we need to calculate a point of which we are a 100% sure it is outside of the mesh, which can be done using their bounding box. This method proceeds by casting a ray from this point to the centre of a random triangle. The direction of the ray does not matter, as long as it hits the mesh and not shoot past it. The first triangle this ray hits, is our starting triangle, and its orientation can be calculated by taking the dot product between the ray and the normal of this triangle. To find out what this ray hits, the ray intersection algorithm brute forces through all leftover triangles. This is not ideal, but finding the closest triangle to the outside point would be just as computationally intensive without auxiliary structures. While I know this part could be optimized, I nonetheless think that this implementation is acceptable within the scope of this assignment.

After Mesh.orient_triangles(), the neighbour pair relations discovered in set_all_neighbours() can be used to reorient all triangles of this body in the correct orientation, which is done depth first by a graph traversing algorithm. The last thing is to write this state of the mesh to a .obj file, after which we are done.

Results

The code satisfies all requirements, while keeping the running time low: 0.3 seconds for bk_soup on my machine (see figure 1). images of the resulting meshes are below. The bk_soup.obj file (the reupload) did however show some problems: unconnected / naked edges. My script was able to discover 85 of them. I also checked this by loading the original, unaltered bk_soup file into Rhino 6, my 3D viewer of choice, and it was able to recognise 94 naked edges, shown below (figure 2). The difference between 85 and 94 probably has something to do with the precision both scripts use to judge “are two edges/points the same“. Since the algorithm is able to detect these edges, a fix could be written, but I decided that this would be outside of the scope of this exercise.



```
PS D:\Stack\02_Geomatics\GEO1004_Model\Assignments\1> python geo1004_hw01.py bk_soup_new.obj fixed.obj
init mesh from obj...
setting neighbours...
fixing normals...
-5611.31625225121
1335.1277995736682
-144.31549610317236
found 3 different bodies of triangles within the mesh.
warning: found 85 naked edges.
writing output obj...
done in 0.272 secs.
PS D:\Stack\02_Geomatics\GEO1004_Model\Assignments\1>
```

Figure 1: Running time and expected print statements of the script

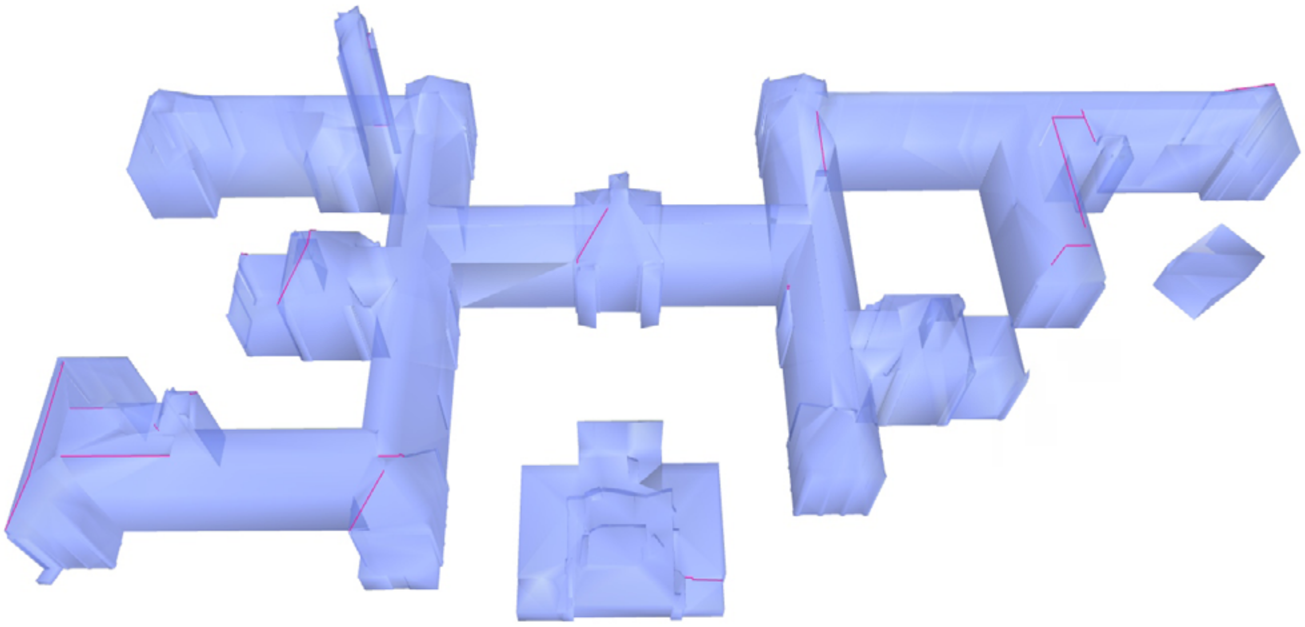


Figure 2: Some naked edges, found by Rhino, shown in purple

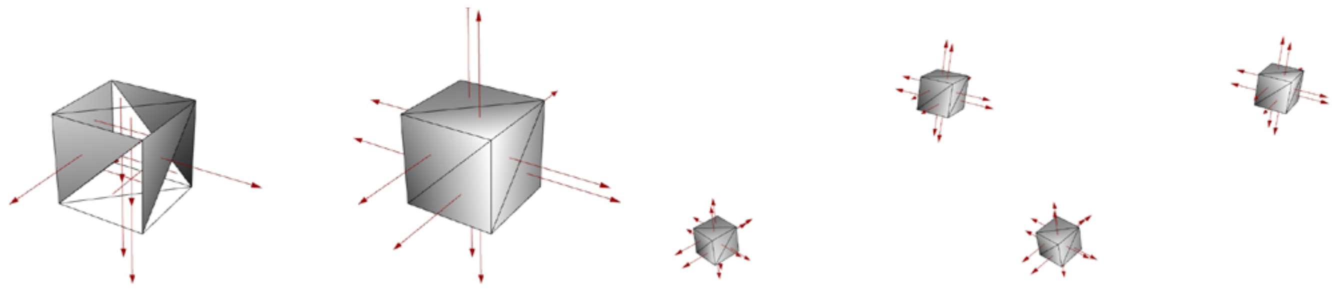


Figure 3: from left to right: original cube, fixed cube, original cubes, fixed cubes.

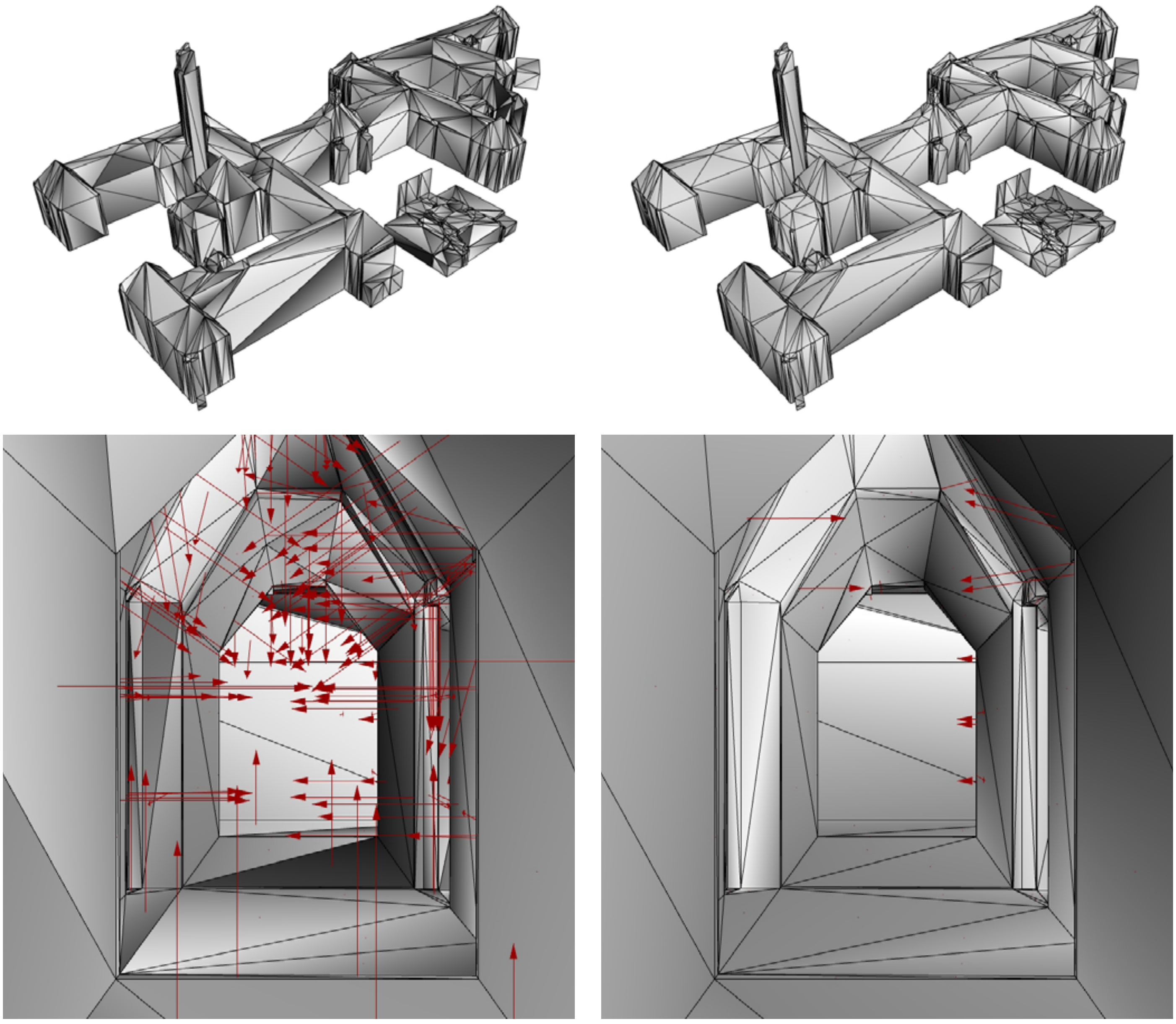


Figure 4: 3D model of the architecture faculty of Delft. Left: original, random normals. Right: fixed normals: all pointing outwards (some minor influence of sliver triangles)

Rhino also has its own “unify normals“ script. It does not figure out what the inside or outside is of a mesh, but it does flip all triangles in the same direction (cw or ccw). I tested it against my code, and found some minor differences(figure 5). It seems that rhino does not recognise the Bouwpub (small building on the bottom right), as it seems completely reversed. Other than that, by jumping back and forth between these two scripts, one can discover 3 other vectors, spread throughout the main building, where the two scripts differ. These belonged to very ‘chaotic‘ triangles, so the ambiguity was justified in my opinion. For the most part, as Figure X indicates, the scripts provide very similar results.

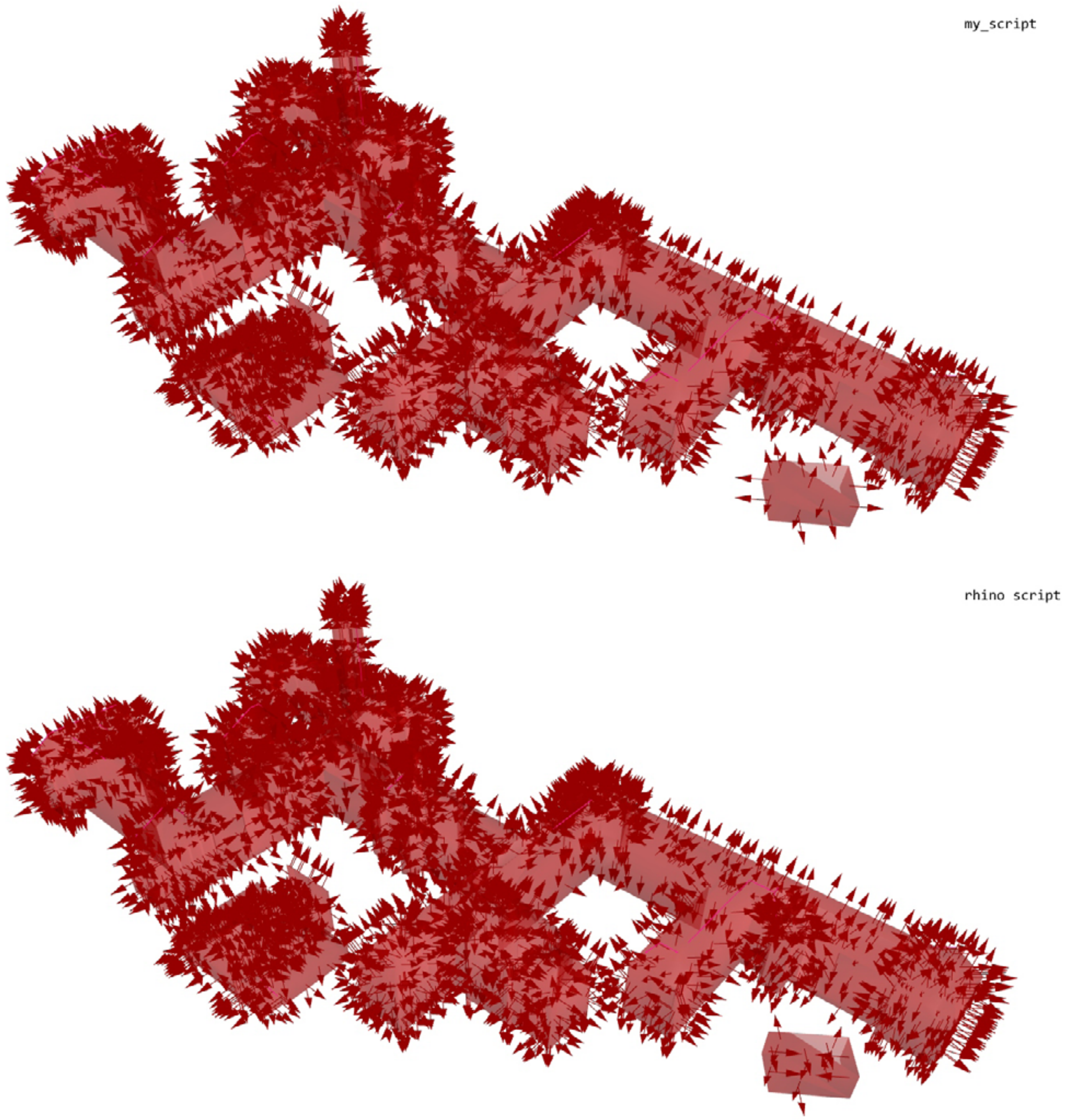
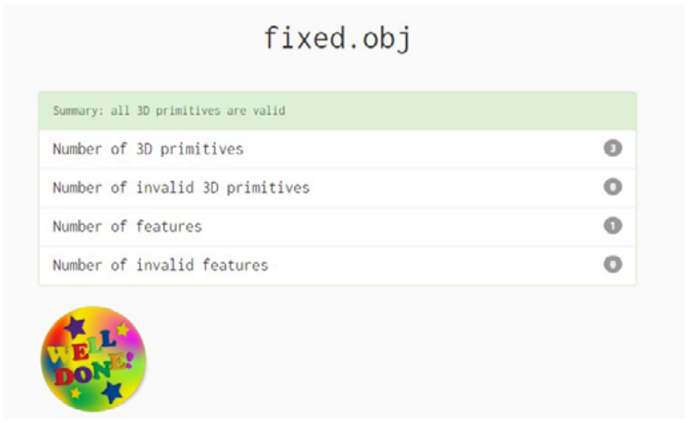


Figure 5: Comparison between Rhino’s way of normal flipping (bottom) versus the code of this exercise (top).

The Val3dity webtool showed no errors (figure 6).



fixed.obj

Summary: all 3D primitives are valid

Number of 3D primitives	3
Number of invalid 3D primitives	0
Number of features	1
Number of invalid features	0

Figure 6: Results from Val3dity

Process (discussion)

In the process of writing the code, The biggest question was the question of organisation. What should go where? I had to choose between a object oriented representation of the mesh (Vertex and Triangle objects) and a representation as two lists, a list of point tuples (x,y,z) and triangles as a list of triangle tuples (p1, p2, p3, nb1, nb2, nb3). I ended up with an object oriented approach. The Mesh class contains two lists, one with Vertex objects, one with Triangle objects. This enabled well-written vector math using overloaded operators, as well as nicely organized methods.

Another interesting subject was choosing how the initial triangle orientation should be determined. Throughout the development of the code, I used multiple variants of the ray casting algorithm, and implemented it in different ways. Many implementations seemed to work, but did not survive Murphy’s Law: theoretically, it could for example intersect triangles exactly on an edge or vertex, which complicates normal calculation. I also kept feeling that there had to be a simpler way of doing this part. The method I settled with is not ideal, and a bit convoluted, but at least I cannot figure out a way to break it. I suspect we will learn the “correct“ way of finding the initial orientation of a triangle soon enough.