# GEO 1004
# 3D Modelling of the build environment
# Assignment 2: Voxelization

Jos Feenstra | 4465768

# Content

# Usage

```
geo1004_hw02.py <Input OBJ> <Output OBJ> <Gridsize> <Clean:0/Dirty:1>
<Input OBJ> the mesh file to rasterize
<Output OBJ> the results of the voxelization, converted back to a mesh
<Gridsize> gridsize to superimpose on the mesh
<Clean:0/Dirty:1> 0: slow but correct brute force method. 1: very fast but slightly ‘wrong‘ method.
```

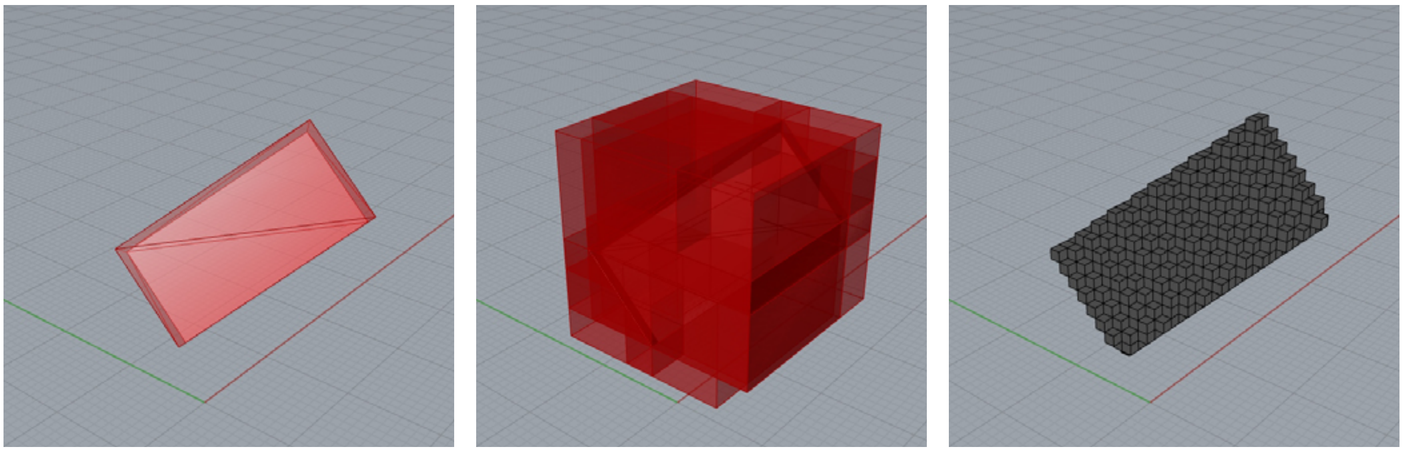


**Figure 1: From left to right: a rotated mesh, its triangles’ overlapping boundaries, and the voxelization result**

# Design

It is important to mention that I've actually handed in two methods. I will be referring to them as the 'Clean' and 'Dirty' method. The Clean method uses a conventional methods of calculating the voxelcloud: per triangle, get its bounding box, and fill it up with intersection targets to ensure 26 connectivity. For each of these targets, if it intersects with the triangle, add it to the voxelcloud. To speed things up, it first calculates the distance to the triangle plane, and discards the voxel if it cannot possibly intersect the triangle.

This can be quite slow due to overlapping bounding boxes (Figure 1). A better approach would have been to use an octree for the mesh, and to use a process of divide and conquer to immediately voxelize a group of triangles within the same area. This ensures bounding boxes will never overlap. This is quite hard to implement within the given time frame.

The dirty method is a 'homecooked' algorithm based upon regular triangle rasterization found in certain shaders. This method tries to find the best 2 dimentional approximation of the 3d triangle, and performs a regular rasterization on it using linear interpolation. These 2d pixels then get elevated back to 3d space using the triangles' corresponding plane.

This approach is exponentially faster than the previous approach, and scales really well with the size of the triangles: the running time for a small triangle intersecting with 4 voxels is not that much different from one which intersects 40.000 voxels. It does however, come with some major drawbacks. Holes still appear with this method, so 26 connectivity cannot be ensured. The "match" with the original mesh is also still off, which has something to do with the projection back to 3d space. Given more time, this algorithm could have been tweaked and corrected to closer to a 'clean' approach, but even in that case, because no true intersections are made, I would still call this method 'incorrect'. Still, I could not forgo its absurd performance speed for large triangles / tiny voxel sizes, so I used it for the larger scale voxelizations.

The algorithm separates the input file per group of voxels belonging to an object with a certain material. These "patches" of voxels are maintained within the "VoxelPatch" class as a python set, making my implementation a **sparse voxel model**. I used a regular three dimentional array up until the implementation of materials, and the large campus dataset, which can only be effectively converted to a mesh using a spare voxel model.

The to_obj() script of the voxelcloud creates the mesh output from the voxelization. It treats the patches of voxels as self contained entities. Per patch, it will not create internal surfaces (surfaces between individual voxels), but it will create a surface with any other patch. This because I believe it should be possible to only select roofs, for example, and be able to see them from all sides. It makes more sense to me given the nature of voxels.

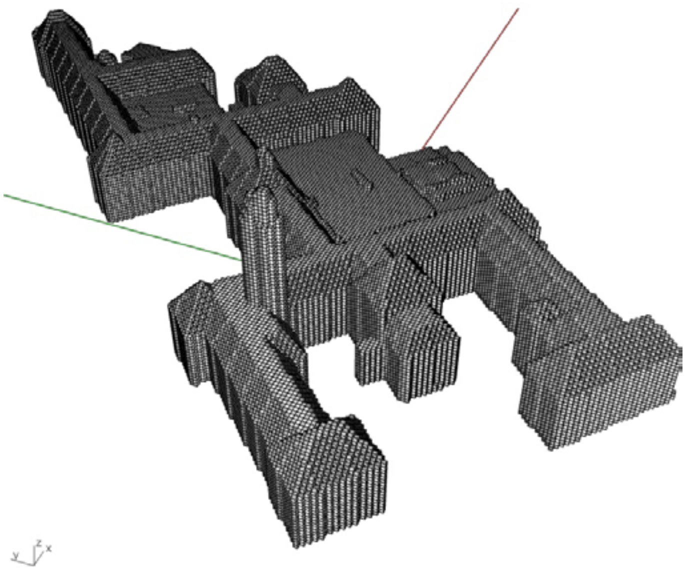
Lastly, I decided not to touch the .mltlib file. For robustness, it would be proper to check if the .mltlib file exists in the target directory, and if not, copy it into this new location. If this was a more 'high end' application I would have done so, but for a small-scale experimental script, i judged this was unnecessary. So, make sure the output file is within the same folder as the mltlib folder to use

## Performance & Results

Figure 2 shows the performance of the Clean and Dirty script side by side, using the bk building as our test subject. It shows why I handed in both: In their current form, they offer a clear trade-off: the Clean script gives a beautiful result without holes, and with a very nice fit to the original data, while the Dirty scripts result isnt that bad on a first glance, and runs at ~1/100 th. of the running time of the clean script in this case, and often way more.

```
Starting...
Gridsize: 1
Mode set to: Clean
init mesh from obj...

voxelize per triangle of mesh. Gridsize: 1...
total number of objects in mesh: 1
object 0 out of 1...
voxelization done in 86.780 secs.

writing output obj...
- converting voxels to mesh...
patch 0 out of 1...
- writing the file
written in 1.423 secs
done!
```

```
Starting...
Gridsize: 1
Mode set to: Dirty
init mesh from obj...

voxelize per triangle of mesh. Gridsize: 1...
total number of objects in mesh: 1
object 0 out of 1...
voxelization done in 0.801 secs.

writing output obj...
- converting voxels to mesh...
patch 0 out of 1...
- writing the file
written in 1.403 secs
done!
```



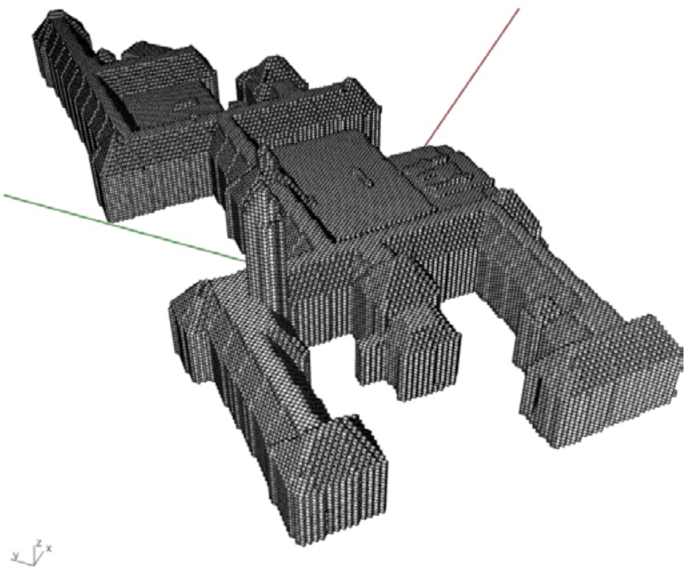


**Figure 2: Clean method (left) vs Dirty method (right), and their results**

Figure 3 and 4 shows in which ways the dirt algorithm fails to meet standards. the holes occur at the edges of triangles, and they occur because of pooly tuned rules about what should happen at edges. Rasterization Algorithms in the field of Computer Graphics have extensive rules about when a pixel does and does not belong to a triangle. This translates poorly to this case, where a 26 connectivity is required with 3d voxels. Given more time, the rules causing 26 connectivity could have been translated to the 2d scale and the plane projection, eliminating these holes and displacements.

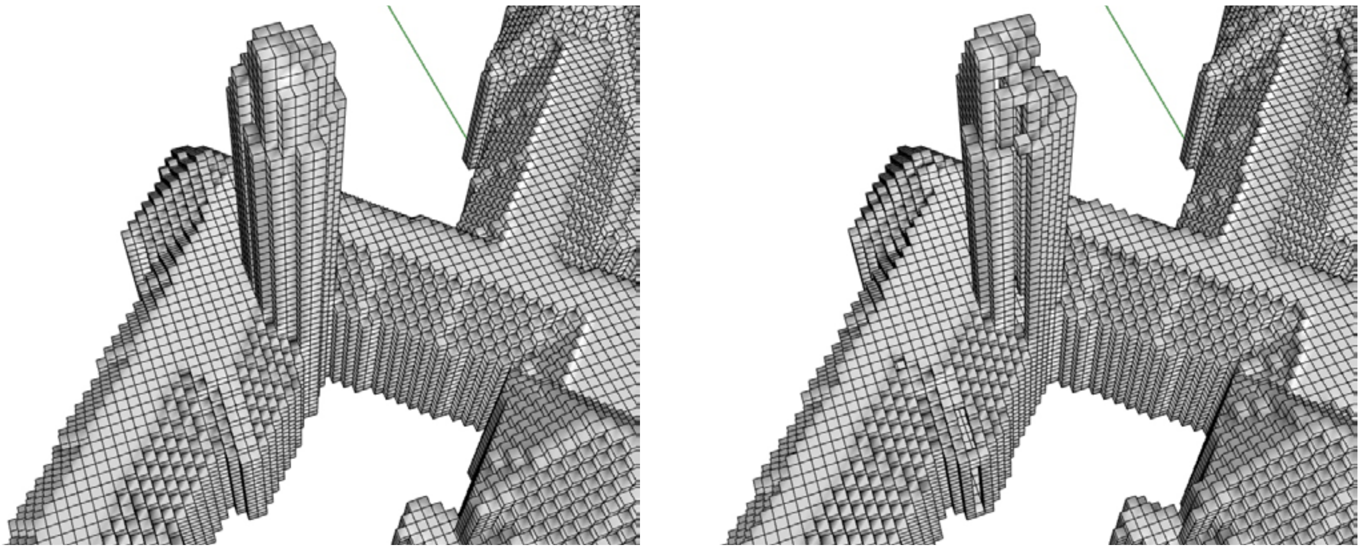


**Figure 2: Clean method (left) vs Dirty method (right). Notice the long hole at the edge of the tower**

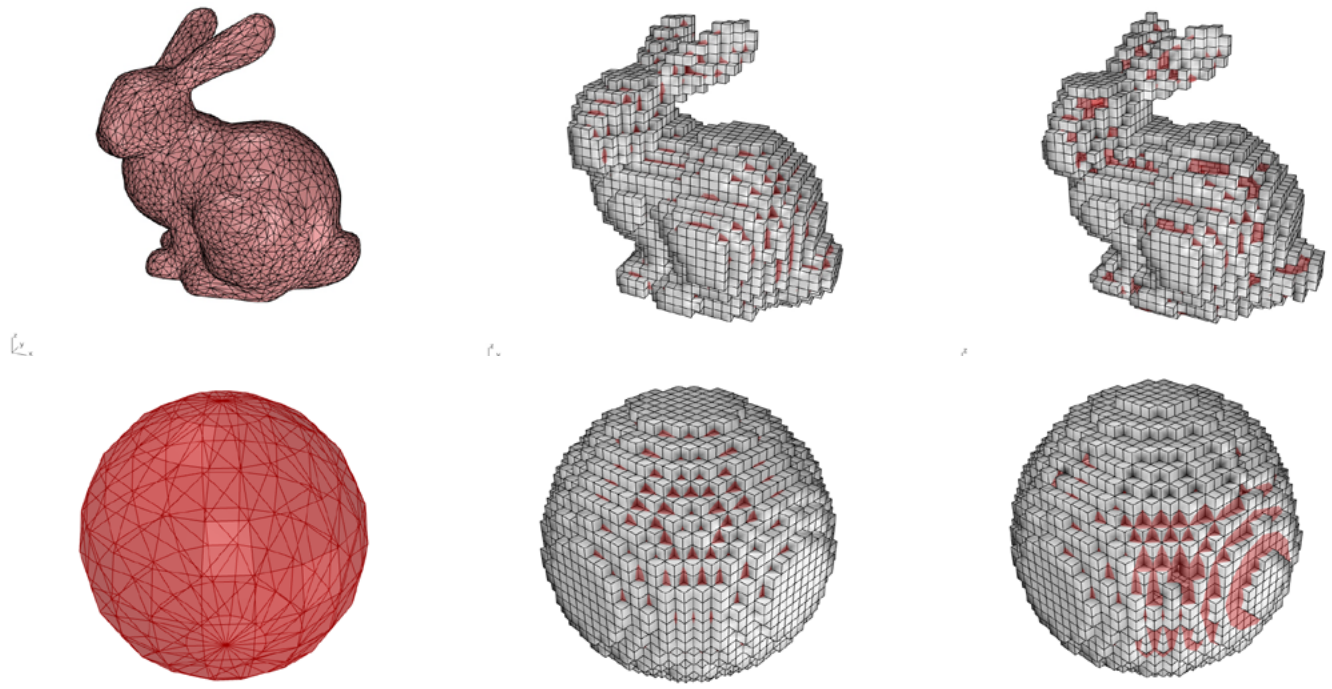


**Figure 4: Left to Right: Original mesh, Clean result, Dirty result**

Lastly, Figure 5 shows the performance of running the campus on a gridsize 10 (Left) and gridsize 1 (Right). Both are created using the Dirty method, and ran at a total 83 seconds and 276 seconds respectively. Notice how the voxelization algorithm itself isn't at all troubled by this 10^3 increase in detail. The number of triangles present in the given mesh weight way stronger than the number of voxels, contrary to the Clean approach.

Also note that the bulk of the running time of the gridsize 1 run lies in the creation of a mesh from the voxel data (per assigned voxel: do you have a neighbour with something that is not your material? If so, bulid a quad.) This of course does scales inverse cubicly with the chosen grid size.

Figure 6 on the next page shows the gridsize 1 result. This model suffers none of the issues seen in the smaller models. This is because the errors are due to something akin to "rounding errors" with small triangles and small numbers.

```
PS D:\Stack\02_Geomatics\GEO1004_Model\Assign-
ments\2\source> python engine.py ../models/
TUDelft_campus.obj ../rhino/voxels.obj 10

init mesh from obj...

voxelize per triangle of mesh. Gridsize: 10...
total number of objects in mesh: 27351
object 0 out of 27351...
(...)
object 27000 out of 27351...
voxelizing done in 80.962 secs.

writing output obj...
- converting voxels to mesh...
patch 0 out of 27351...
(...)
patch 27000 out of 27351...
- writing the file
written in 1.668 secs
done!
```

```
PS D:\Stack\02_Geomatics\GEO1004_Model\Assign-
ments\2\source> python engine.py ../models/
TUDelft_campus.obj ../rhino/voxelcampus_1.obj
1

init mesh from obj...

voxelize per triangle of mesh. Gridsize: 1...
total number of objects in mesh: 27351
object 0 out of 27351...
object 1000 out of 27351...
(...)
object 27000 out of 27351...
voxelization done in 105.849 secs.

writing output obj...
- converting voxels to mesh...
patch 0 out of 27351...
(...)
patch 27000 out of 27351...
- writing the file
written in 170.245 secs
done!
```

**Figure 5: running time of the campus dataset with 2 different grid sizes**

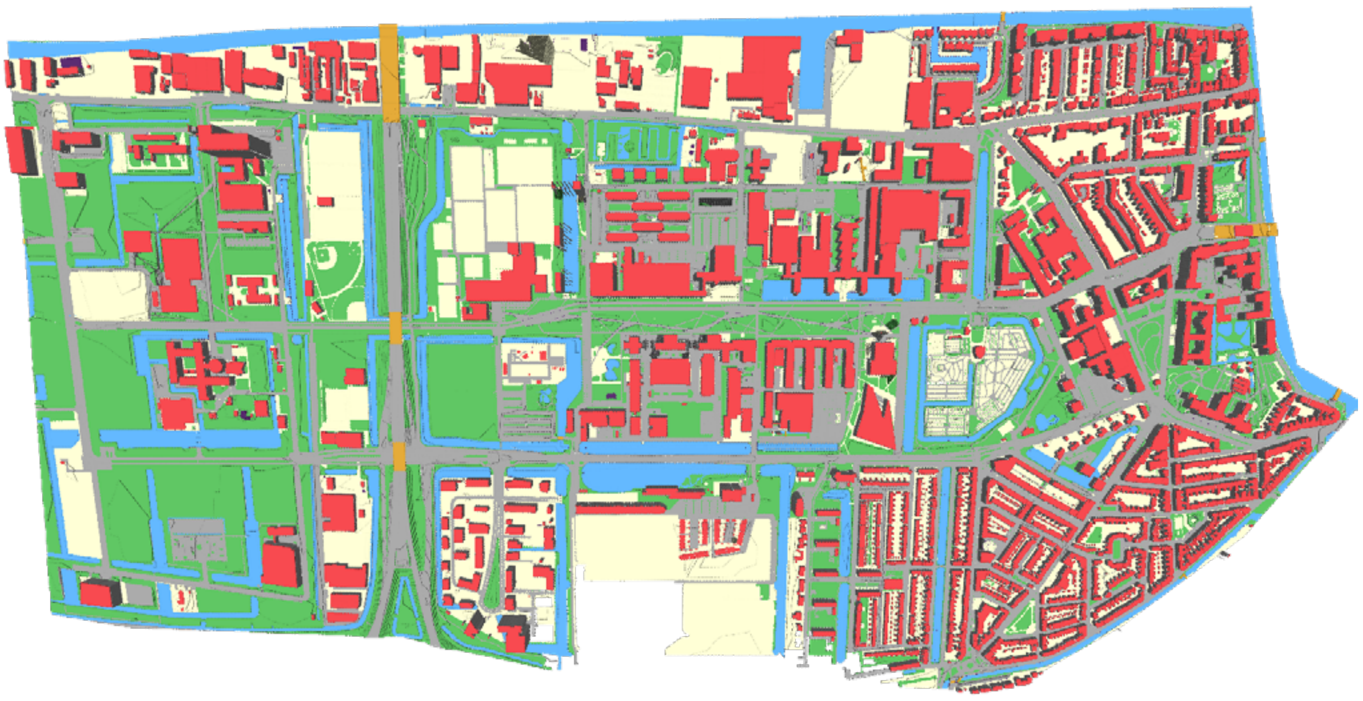
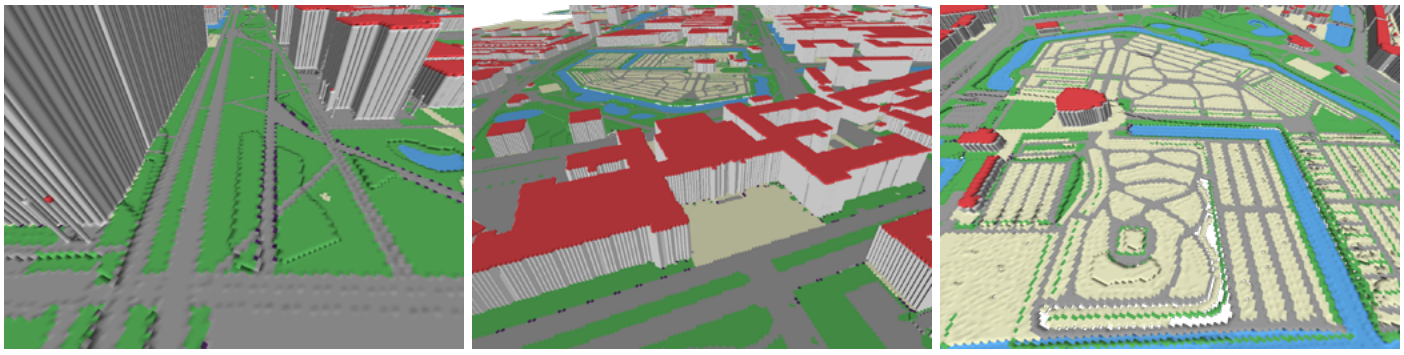



**Figure 6: The voxelized campus with a grid size of 1 (Dirty method).**

## Running time

I find it difficult to assign a big O notation to both scripts, since most of the theoretical running times derived from the code do not properly reflect the differences in performance I have benchmarked. The Clean script runs at a worst case of O(h*w*d*n), where h, w, d is the height, width and depth in voxels, and n is the number of triangles. Practically it is less, for the bounding boxes overlap, and many voxels of its full bounding box get discarded before heavy calculations start. I would say that the results on the previous page show that the Dirty script runs more or less at O(n), where n is the number of triangles.

## Discussion / process

They say good programmers throw away more than 90% of their code. I don't know if I am a good programmer, but my code definitely suffered heavy losses. Still, I think this is a good thing.

I have tried a number of alternative methods for voxelization. For example, the original mesh class was much larger than the version handed in. I tried certain heuristics based upon a well oriented and topologically sound mesh model, which was easy by recycling the code for the first assignment. By grouping all triangles present in a certain bounding box, I could avoid calculating voxels within the same bounding box twice. This started to look akin to an octree 'divide and conquer' method, mentioned in Nourian et al. (2016) among other places. However, this proved to be quite some work, and I became more interested in getting good results from the 'dirty' appoach, since it dodged the issue of filling a full bounding box with potential voxels altogether. This meant that the mesh class which was handed in is now only a shell of what it once was, only used for making the obj input ready for processing.

The writing of this voxelization script was quite challenging for a number of reasons. A lot of auxiliary code was needed before I could even begin to start voxelization. Reading and writing an obj file, how to visually debug the different steps of the code, etc. The code handed in still contains some of this code to demonstrate how this was handeled: a 'to_txt()' function is present in the voxelcloud script, which was used to parse the original three dimentional array directly to a csv-like format, which is of course way quicker than converting voxels back to a mesh. To be honest, while I understand the necessity of a good visualisation of the voxels, I do think that converting the voxels back to an obj file defeats the purpose of performing a voxelization in the first place. Plenty of my fellow students spent more time on the reuse of vertices in this obj parser, than the entirety of the voxelization algorithm itself.

The uncertainty of a good workspace due to the COVID-19 virus did affect the work productivity in the last couple of days, but not to the point that a delay for the deadline was necessary.

I greatly enjoyed this exercise. I could really appreciate the 'thrownness' of trying to figure this algorithm out for ourselves. Being able to write good algorithms for certain applications is both a science and a craft I would say, and these types of assignments are excellent for developing both. This setup also gave us to room to experiment with our own methods and to really get a feel for the algorithms we are told. Credits to Laurens van Rijssel for the idea of a line-scan approach. We discussed methods a lot together.

## References

Nourian, P., Gonçalves, R., Zlatanova, S., Arroyo Ohori, K., Vu Vo, A. (2016). *Voxelization algorithms for geospatial applications: Computational methods for voxelating spatial datasets of 3D city models containing 3D surface, curve and point data models.* TU-Delft. Accessed at 16/03/2020 from https://www.sciencedirect.com/science/article/pii/S2215016116000029#!