Loading Geometry into NetLogo3D

Gabriel Wurzer, Vienna UT (gabriel.wurzer@tuwien.ac.at)

Abstract. NetLogo3D is an Agent-Based Simulation which uses a three-dimensional raster as environment its agents. It has three different ways to represent geometry, (1.) as wireframe made out of vertices and edges, (2.) as solid faces or (3.) as voxel grid. Sadly, loading geometry is only supported for the second option, through a proprietary text format for which no converter exists. The other two options are completely left to the programmer. I find that this is a pity, and have therefore worked on loading geometry from a standard format (Wavefront .obj) and representing it as one of the three mentioned options. The description of the resulting model, available at [1], is the aim of this technical report.

Introduction

NetLogo (Wilensky 1999) has only limited support for loading spatial data:

- In the recent 3D version, one may import polygons from a GIS database or write a parser for a 3D file format. In both cases, one must create a turtle for each vertex and a link for each edge of a face (wireframe).
- Faces themselves can only be loaded as shape (i.e. `load-shapes-3d`). Netlogo uses a proprietary text format for this, without supplying a converter. Even after loading the shape, one may only display the mesh but cannot access any geometrical information (such as the faces).
- Voxelized data can be loaded to a certain extent using bitmaps, by using `import-pcolors` for loading a single layer of cells. One could use external tools for voxelization (e.g. binvox [2]; which is based on Nooruddin and Turk 2003), parse the generated slices (using for example binvox-rw-py [3]) and write each of them to a bitmap that are loaded with the above technique. However, there are quite a few options in this workflow (cell size, voxelization settings and so on), a whole conversion of a mesh will likely not work "out of the box".

But it should. In fact, we want to be able to import a mesh and choose what representation the data should have, without going through a workflow involving external programs. In this report, we show exactly how we have achieved this, giving actual results from our implementation which is available for free.

Proposed workflow

Figure 1 shows an overview of our approach: Geometry being available as Wavefront .obj is read into the program by means of an .obj parser, resulting in a set of vertices (turtles) and edges (links). We also store object and face information using own breeds of turtles (*triangles* of an *object*) which are non-visible. We need that information for later, when we compute either shaded or voxelized versions of the model.

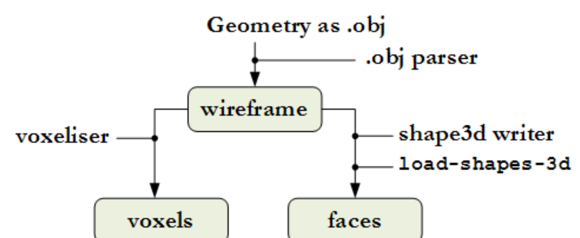


Figure 1. Overview of the approach

Wireframe loading

The loading of the wireframe (see Figure 2) is straightforward. We instantiate a breeded turtle ("*vertex*") for every vertex and multiple breeded links ("*edges*") for every polygon, which we automatically subdivide into triangles. For each triangle, we instantiate a turtle ("triangle") that acts as a data container; it keeps track of its *vertices* a, b, c. Likewise, we create turtles of breed "*object*", which hold the vertices list and triangles list.

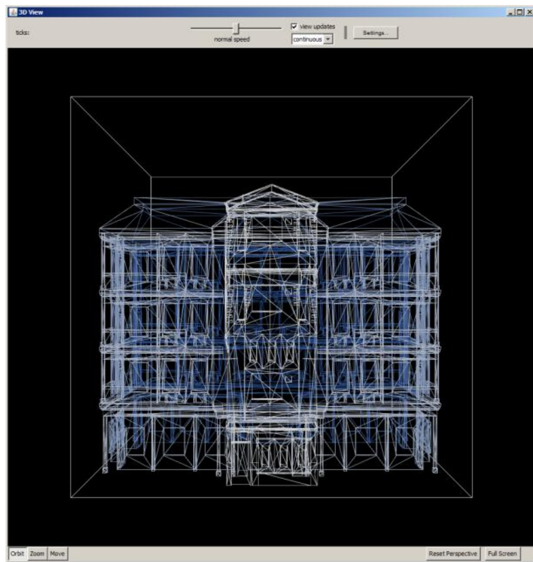


Figure 2. Wireframe view showing only edges.

Voxelisation

For the voxelisation, we have to solve what voxels (i.e. patches) intersect a face of an object. The general approach for doing this in 3D would have been to enclose a voxel by a sphere (Figure 3), project the center point *C* of that sphere onto the triangle's plane and see whether the distance *d* between *C* and the projected point *C'* is smaller the sphere's radius *r*, thus giving an intersection. However, there are some caveats in that method: The projection might lie outside of the triangle, in which case one has to figure out the *closest point within the triangle*, which is either a corner point or a point along the three segments.

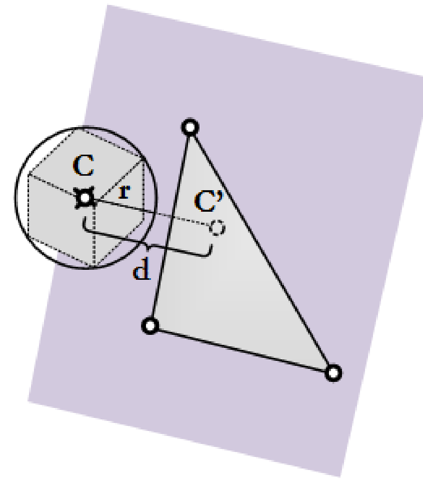


Figure 3. 3D Voxel-In-Triangle Test

Because we were aiming at students of architecture (with limited math knowledge), we rather had to think about a different approach that is easier to understand and implement. The core idea is simple:

- We fill the patches containing the three corner points and the center.
- Then, we look at the longest edge of the triangle to decide if we have to subdivide it.

If the longest edge is of length 1, it might either be contained in one voxel (left in Figure 4), or in two (middle in Figure 4). A longer edge may have in-between voxels (right in Figure 4) and thus we have to subdivide.

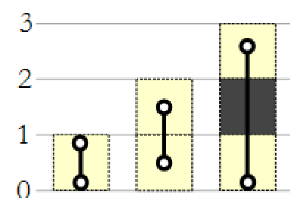


Figure 4. A triangle edge. (left) Edge of length 1 is contained in one voxel. (middle) Same edge contained in two voxels (right) Edges longer than 1 possibly have "in-between" voxels.

In order to fill a voxel, we address the patch at the coordinates of each endpoint and set its color. Thus, the two endpoints should

cover two patches at most. In practice, however, I have found that there can be the case where Netlogo jumps over to the “next-higher” patch, due to numerical instability or boundary conditions for addressing a patch. Thus, I also fill the center point of the triangle, in order cover any such imprecision. The whole algorithm, used on every triangle, is as follows:

```
voxelize (a, b, c)
  center = (a+b+c) / 3
  fill voxels at a,b,c,center
  l = length of longest edge
  if l > 2 then
    …subdivide and run voxelize
     on each of the resulting 4
     triangles
  end
end
```

The subdivision computes the midpoint of each edge. It then connects the original corner points to the new midpoints, building four new triangles on which to run the algorithm.

An example derivation is given in Figure 4:

- Given an initial triangle, we first fill the patches containing the corner points and the center point (left in Figure 5).
- The longest edge has a length greater than 1. We thus subdivide (right in Figure 5) and get the four triangles shown. Each of these is again subjected to the algorithm (recursive call), so long as there is an edge with length > 1.

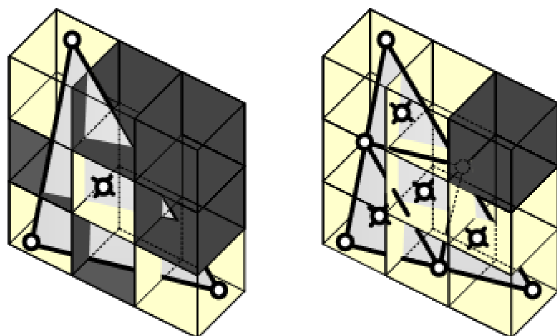
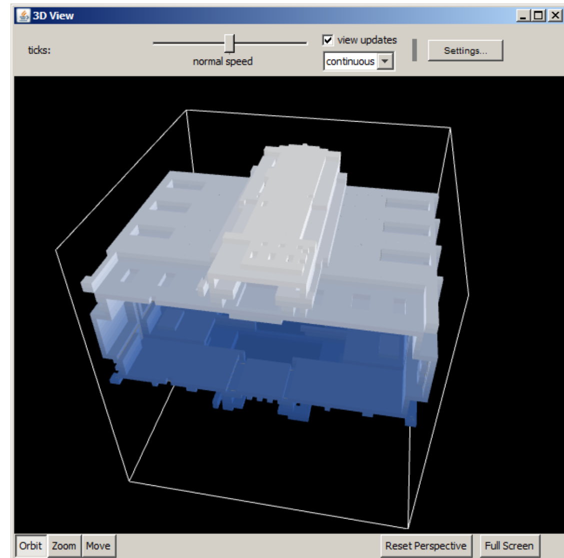
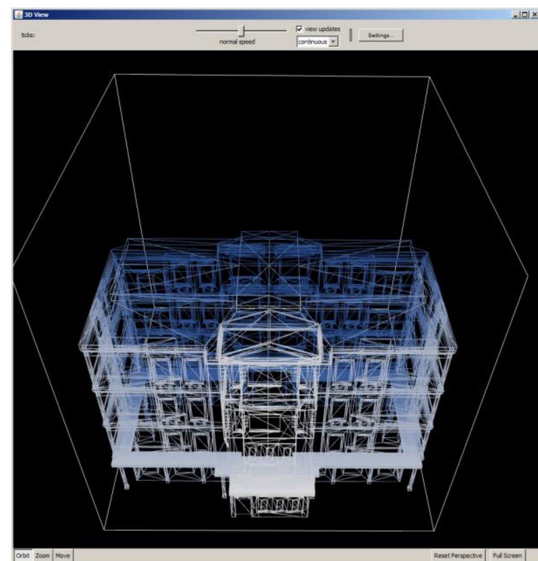


Figure 5. Example derivation. (left) First step, (right) second step.



(a) Voxelized mesh from a bottom view



(b) Wireframe plus one voxel layer.

Figure 6. Voxelised mesh, depth-colored.

The result of the voxelisation is given in Figure 6: We might either depict the whole building (Figure 6a) or use a section (Figure 6b).

Show faces

For showing faces, we have to write a Netlogo3D shape file to disk, load that via `load-shapes-3d` and assign the newly-imported shape to a turtle. The shape3D writer simply takes the information already present in the scene, i.e. the triangles list of

an object, and converts it into the format that Netlogo3D uses. The loading is straightforward, however, Netlogo wants that there already exists a shape of the same name in the 2D shape library. Since it is impossible to programmatically create a new shape in that, we have chosen to add a predefined one called “custom shape” in a pre-step. The exported 3D mesh therefore also needs to be called “custom shape” in the 3D shape file. This is not a big issue, as the name of the 3D shape file is free for us to choose, i.e.: even though the shape name is always the same, we get a different geometry depending on the file we load.

The last part of showing the shape was a bit awkward. We assign the shape to a turtle, i.e. `set shape "custom shape"`, but then, the mesh does not show up. Through trial and error, I found that the size of the turtle must be exactly 0.3 so that the geometry matches up (see Figure 7). This is probably a bug.

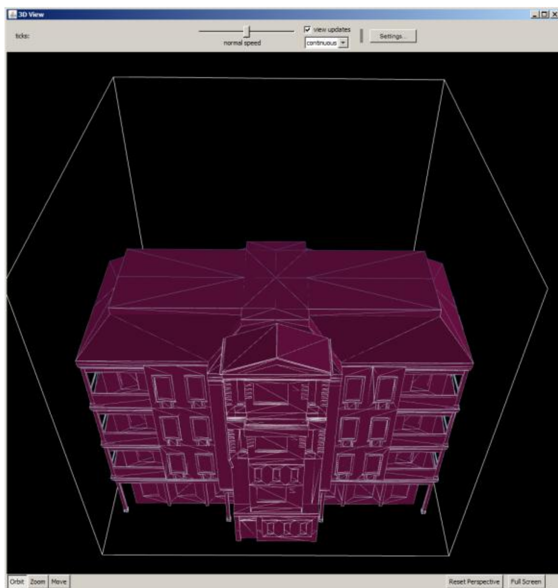


Figure 7: Faces in Netlogo3D

As final visualization, we can also superimpose the voxelized version on top of the solid mesh (Figure 8). A further extension would be the introduction of an arbitrary slice plane, however, this is left to the model developer and hence to future work.

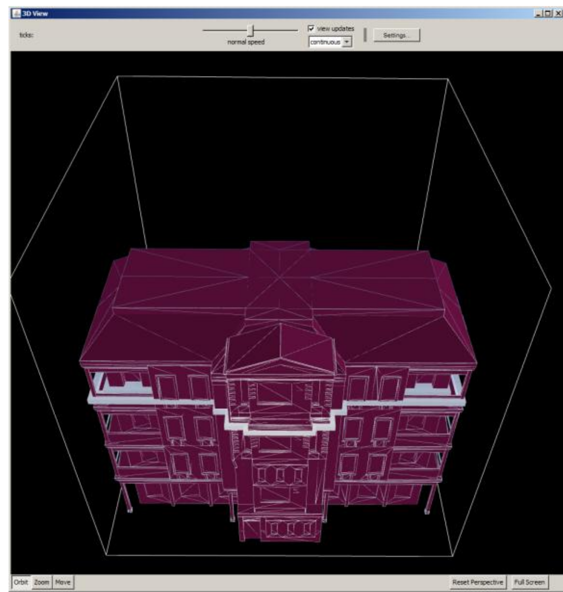


Figure 8: Voxels superimposed on the mesh

Discussion

Our model is pretty straightforward. The geometry we import needs to be modeled in the positive quadrant (X+,Y+,Z+), our world has to have its origin in the southwest bottom corner.

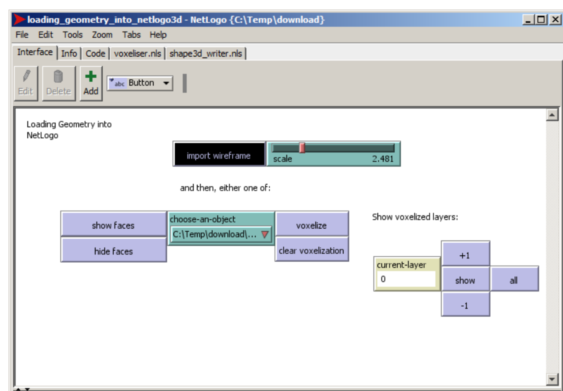


Figure 9: UI of our model during testing

We did our tests on a mesh containing 5935 vertices and 8631 faces, as shown in the figures, with Netlogo3D 5.0.5. It took 40s to import the wireframe, on a Intel Core i5 Dual Core with 2.67 GHz and 4GB RAM running Windows 7. As shown in Figure 9, we had the command center minimized and the 3D View set to “no view updates”. With

the same settings, we ran our “show faces” algorithm, which took under 5s. The voxelize algorithm largely depends on the world resolution. We used 64x64x64 patches, the algorithm ran 15s until completion. All in all, we feel that this is a reasonable performance given that the algorithms used were geared at simplicity, not performance.

Acknowledgements

I want to acknowledge the author of the 3D geometry used, jas.sur on TurboSquid, for his ‘cozy little suburban apartment’, which I use under the terms of the Royality-Free License given by TurboSquid. Thank you! I am packaging a copy of the geometry with the Netlogo model, so that people have a test case.

I furthermore wish to thank Seth Tisue for his Goo Extensions, which allows me to change user interface elements at runtime in NetLogo. I use his extension as by public domain.

Model Download

You can retrieve the model at www.iemar.tuwien.ac.at/?page_id=1299

References

Nooruddin, F. and Turk, G. 2003. Simplification and Repair of Polygonal Models Using Volumetric Techniques. IEEE Transactions on Visualization and Computer Graphics, 9, 191-205.

Wilensky, U. 1999. NetLogo. http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL.

[1] www.iemar.tuwien.ac.at/?page_id=1299 [accessed 26.10.2014].

[1] www.cs.princeton.edu/~min/binvox [accessed 26.10.2014].

[2] github.com/dimatura/binvox-rw-py