University *of* Glasgow | School of Computing Science

Project Title

Michael Kilian
Tony Lau
Dan Tomosoiu
Hector Grebbell
Peeranat Fupongsiripan

Level 3 Project — 13 March 2013

0.1 Using the 3D Model

0.1.1 jMonkey Model Import

Once the 3D model of the evacuation environment has been completed using Blender, it needs to be converted to a logical data structure (in this case, a Navigation Mesh) that can be used by the project for path-finding purposes. The first step of creating this structure is to convert the Blender file into a jMonkeyEngine (jME) compatible file. This process can be done automatically with the help of the jMonkey Integrated Development Environment: the file created in Blender is converted to a binary encoding of the mesh, which can be interpreted by jME. Once created, this file is permanently stored as a project asset and can be used whenever the user wishes to simulate an evacuation on that specific model. The next step in creating the compatible data structure is to read the binary encoding of the environment and transform it into an initial jME mesh. This is a geometric mesh consisting of collections of three types of geometric primitives:

- **Points**: Holds a vertex representing a single point in space
- **Lines**: Two vertices representing a line segment
- **Triangles**: Three vertices representing a solid triangle primitive

In this particular case, the jME mesh will hold the environment's geometric information using a list of 3D triangles, that were mapped from the imported model's surfaces.

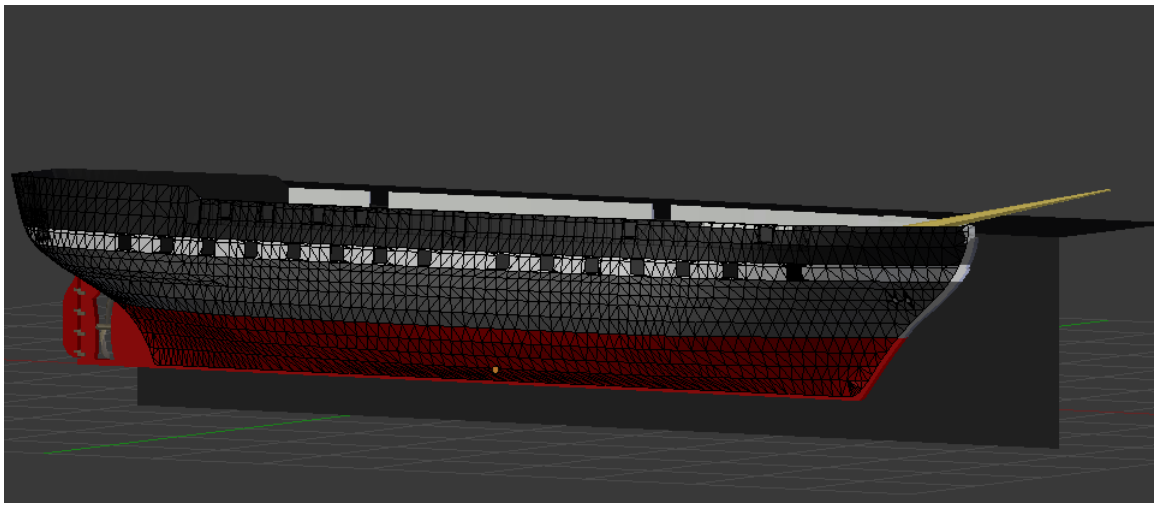


Figure 1: View of the Blender model

0.1.2 Conversion to a Navigation Mesh

In order to allow agents to navigate over a mesh, a collection of the 3D surfaces (in this case, interconnected triangles) can be used. Any point interior to these triangles represents a valid location at which an agent can be positioned at any point in time. Furthermore, the agent can move from any point inside the triangle's surface to any other point inside the same surface.

Each of these triangles can have references (links) to zero up to three other neighbouring triangles, one for each edge of the triangle. If a link exists between two triangle (i.e. they share a common edge), this means that an agent can traverse from one triangle to the other or vice-versa. As a result, a collection of such triangles, or cells, is a convenient structure on which to use graph path-finding algorithms: every cell can be interpreted as a graph node, and any link to other cells as a vertex between two nodes. This whole collection of triangles (cells), which can be abstracted as a graph, is called a Navigation Mesh (navmesh).
The NMgen project focused on the creation of navigation meshes from geometry meshes. The NMgen Java library was used in order to convert the project's imported environment models into agent-navigable structures.
To create a navigation from an existing geometry mesh, a series of sequential processes is required:

1. **Voxelisation**: Create a solid height-field from the source geometry, made from a collection of voxels. A voxel (volumetric pixel) is a 3-dimensional, box-shaped unit used in representations of 3-dimensional images;

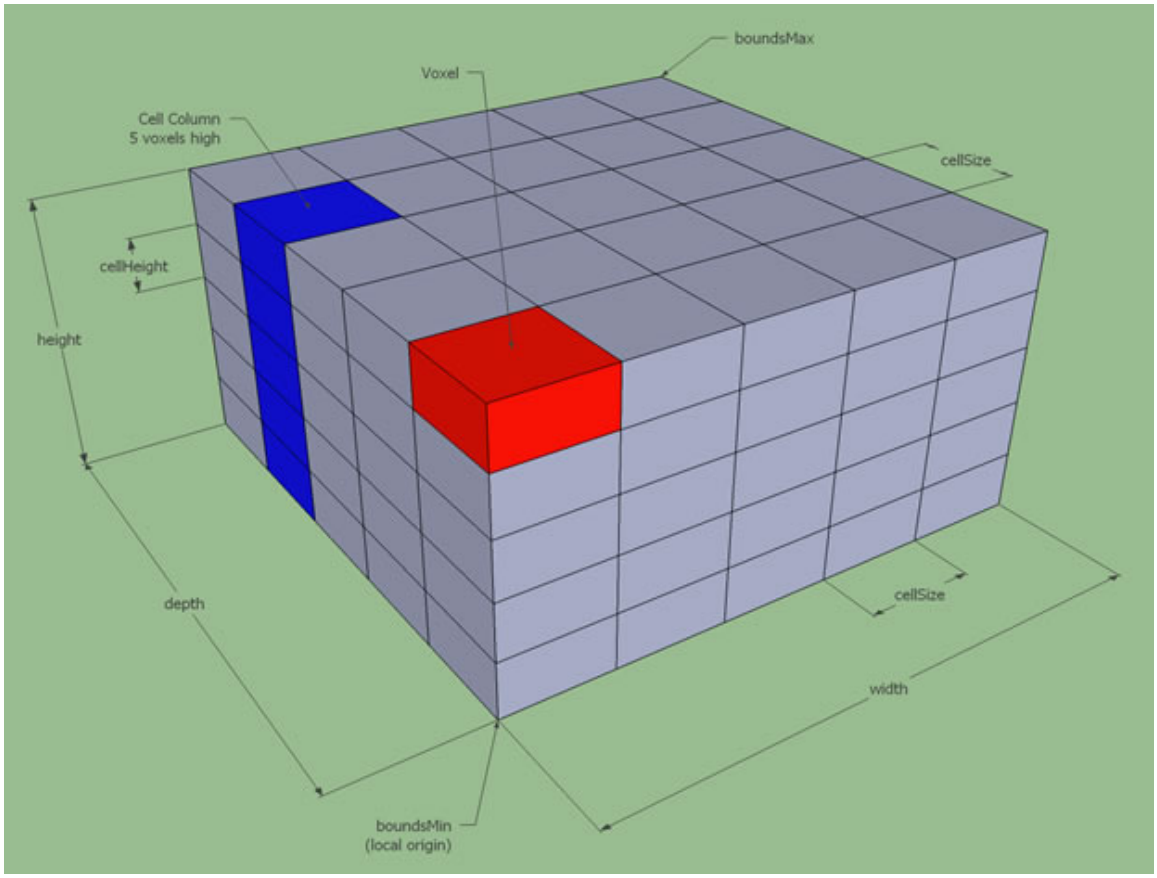
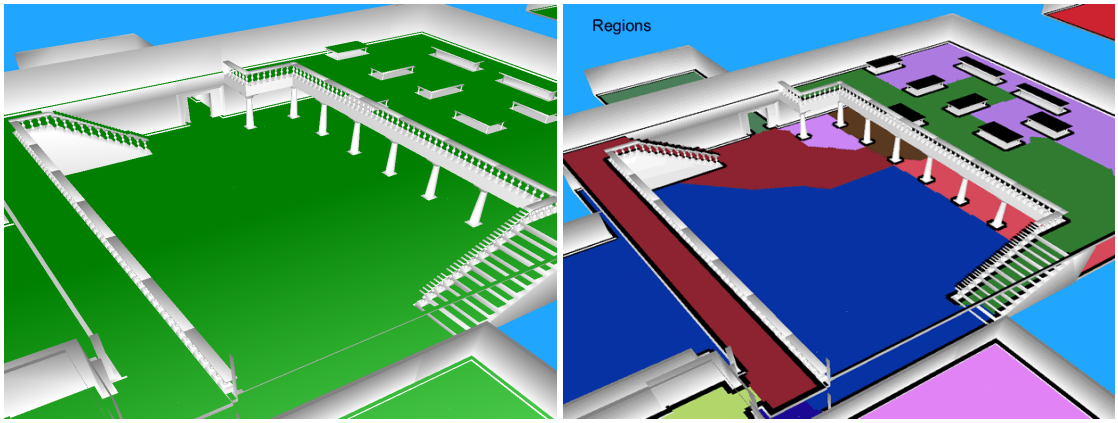


Figure 2: A voxel grid

2. **Region Generation**: Detect the top surface area of the solid height-field and divide it up into regions of contiguous spans;

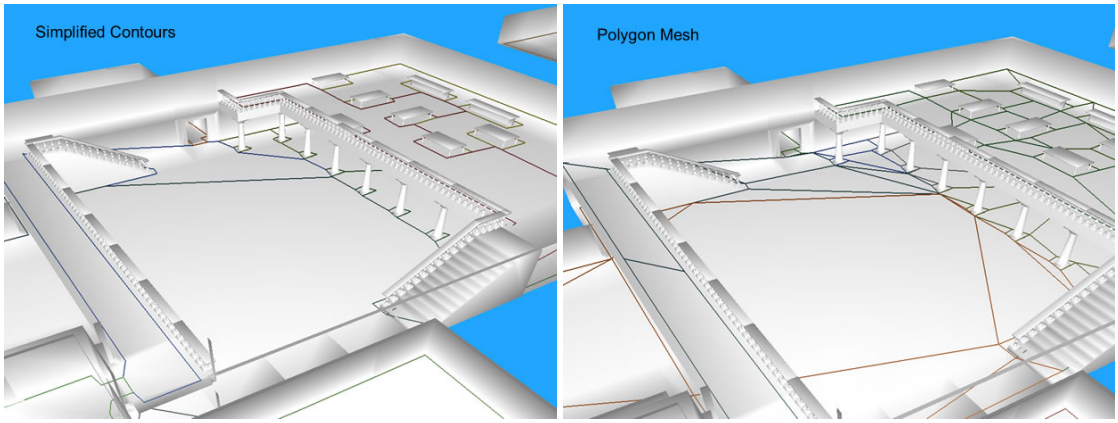


(a) Overview of the model (b) Model split into regions

Figure 3: Region generation

3. **Contour Generation**: Detect the contours of the regions and form them into simple polygons;

4. **Convex Polygon Generation**: Sub-divide the contours into convex polygons;



(a) Contour generation (b) Convex polygon division

Figure 4: Region generation

5. **Detailed Mesh Generation**: Triangulate the polygon mesh and add height detail.

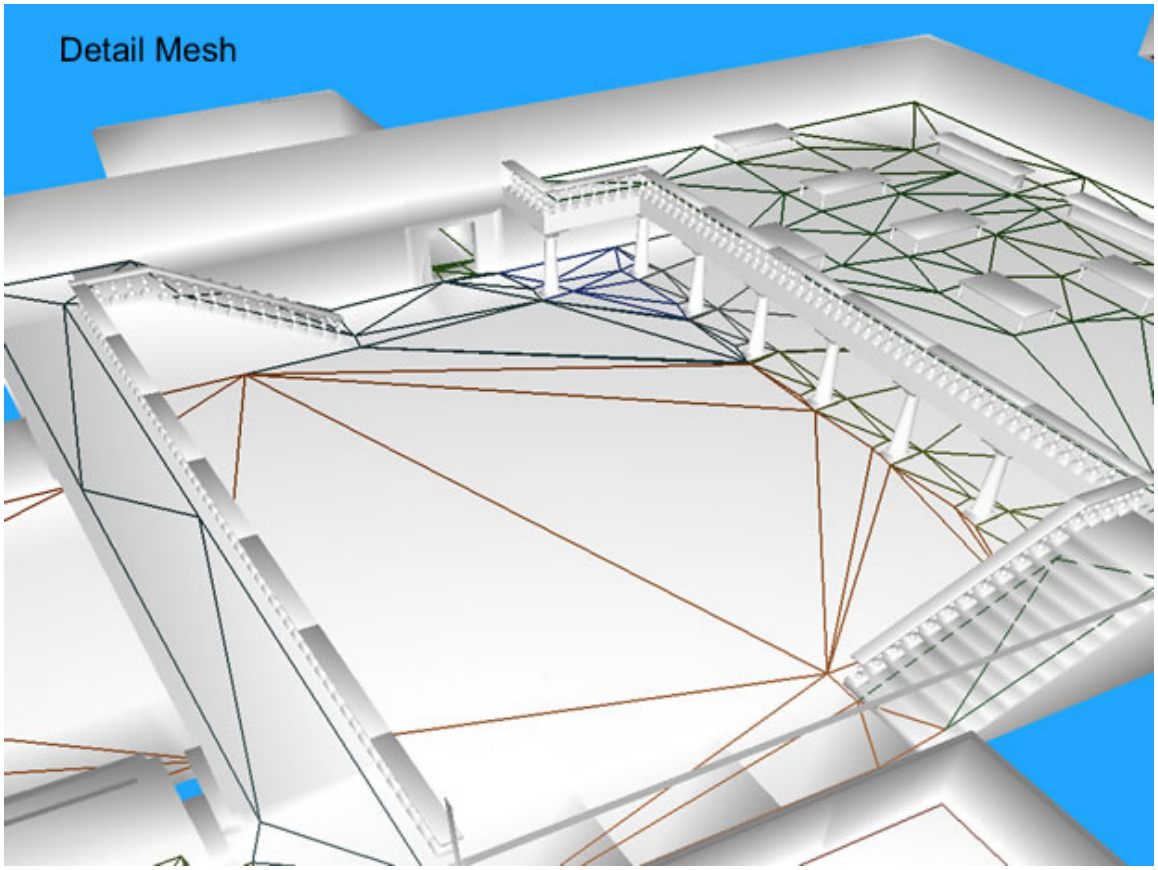


Figure 5: Detailed mesh

In order to obtain a well-formed mesh, several parameters are required to be passed to the generator (note: where parameters are dependent on variables like agent surface area or agent height, the corresponding values should be calculated in relation to the scale of the imported model):

- **Cell size**: Represents the width and depth of the voxel units which will fill the source geometry. This parameter will influence the accuracy of the generated navigation mesh in relation to the supplied geometry mesh. Lower values closely make the result closely match the source geometry but with a proportional increase in computation time and space costs. In order for the navmesh to be well-formed, this parameter needs to be at least several times smaller than the size of an agent's surface area.
- **Cell height**: Represents the height of the voxels used to create the solid height-field. Like the width and depth, the height of the voxel influences the level of accuracy of the navmesh. A well-formed navmesh requires this parameter to be at least several times smaller than the height of an agent's maximum step (step-height).

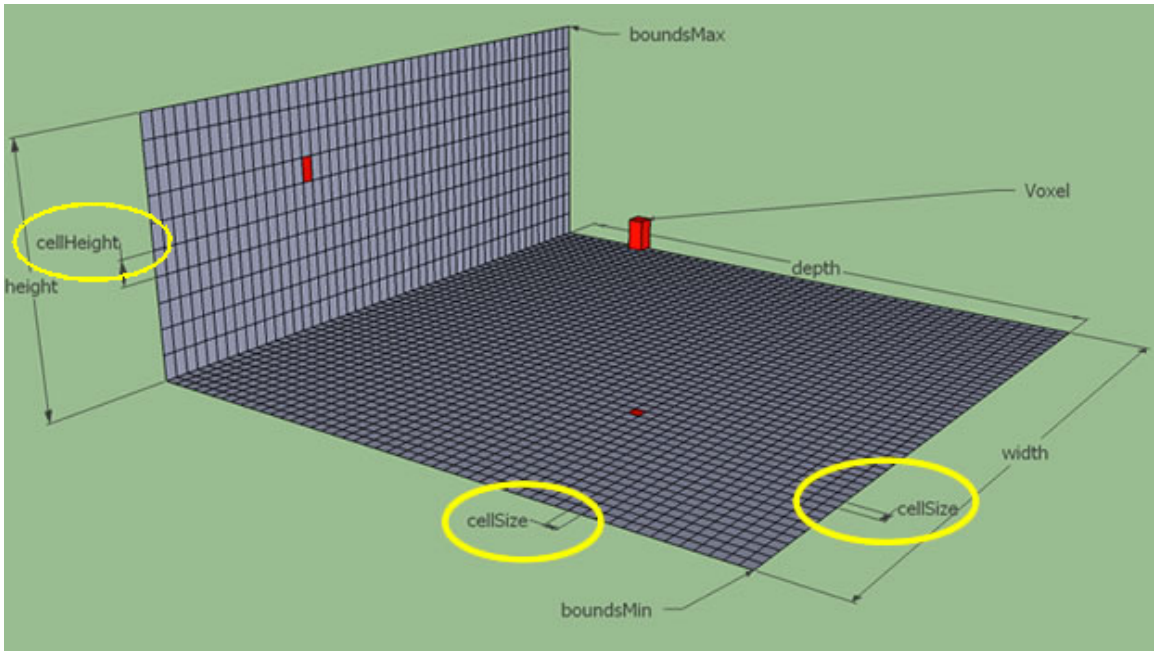


Figure 6: Cell size and height

- **Minimum traversable height**: Represents the minimum distance from the floor to the ceiling that will still allow an agent to pass through. Should be at least the size of maximum agent height. Should also be at least two times the height of a voxel (cell height).

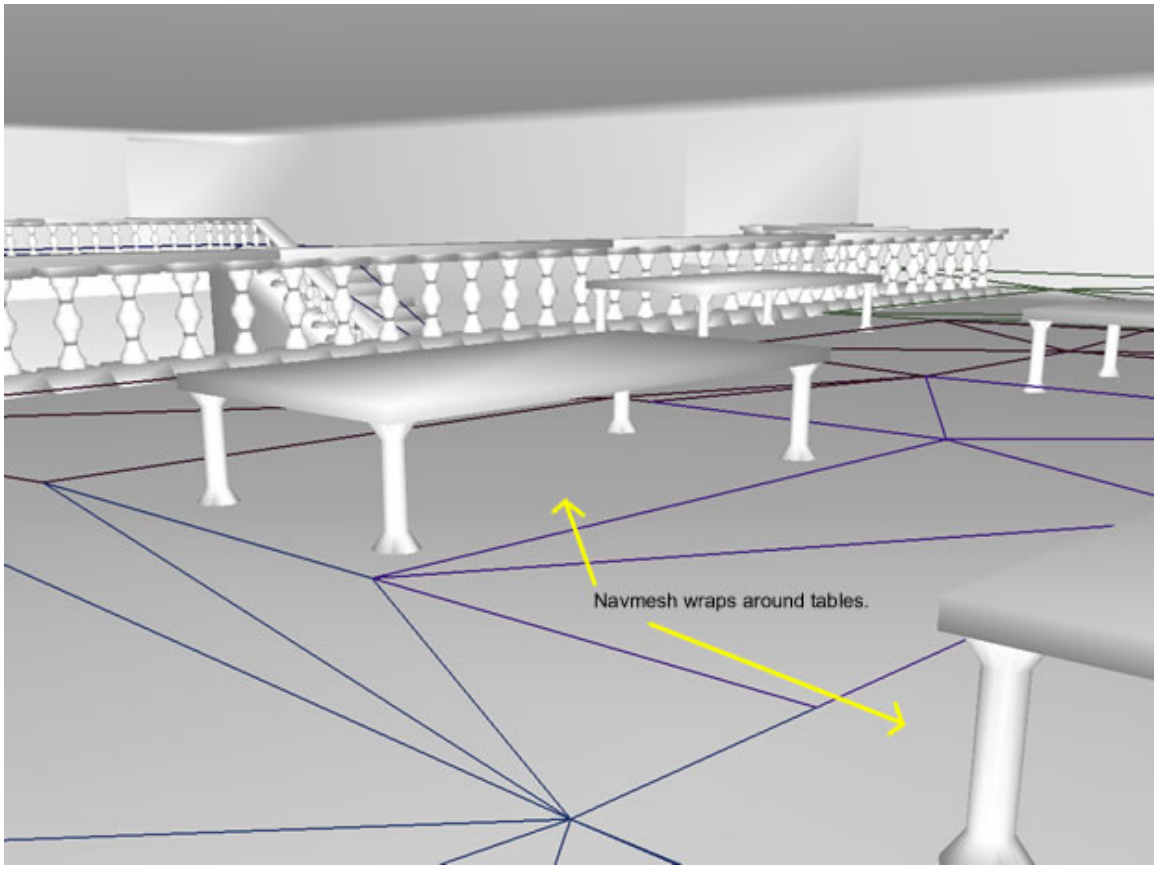


Figure 7: When minimum traversable height set correctly, the mesh does not flow under the tables

- **Maximum traversable step**: Represents the maximum height of a ledge that can be climbed by an agent. Should be at least twice the height of a voxel.

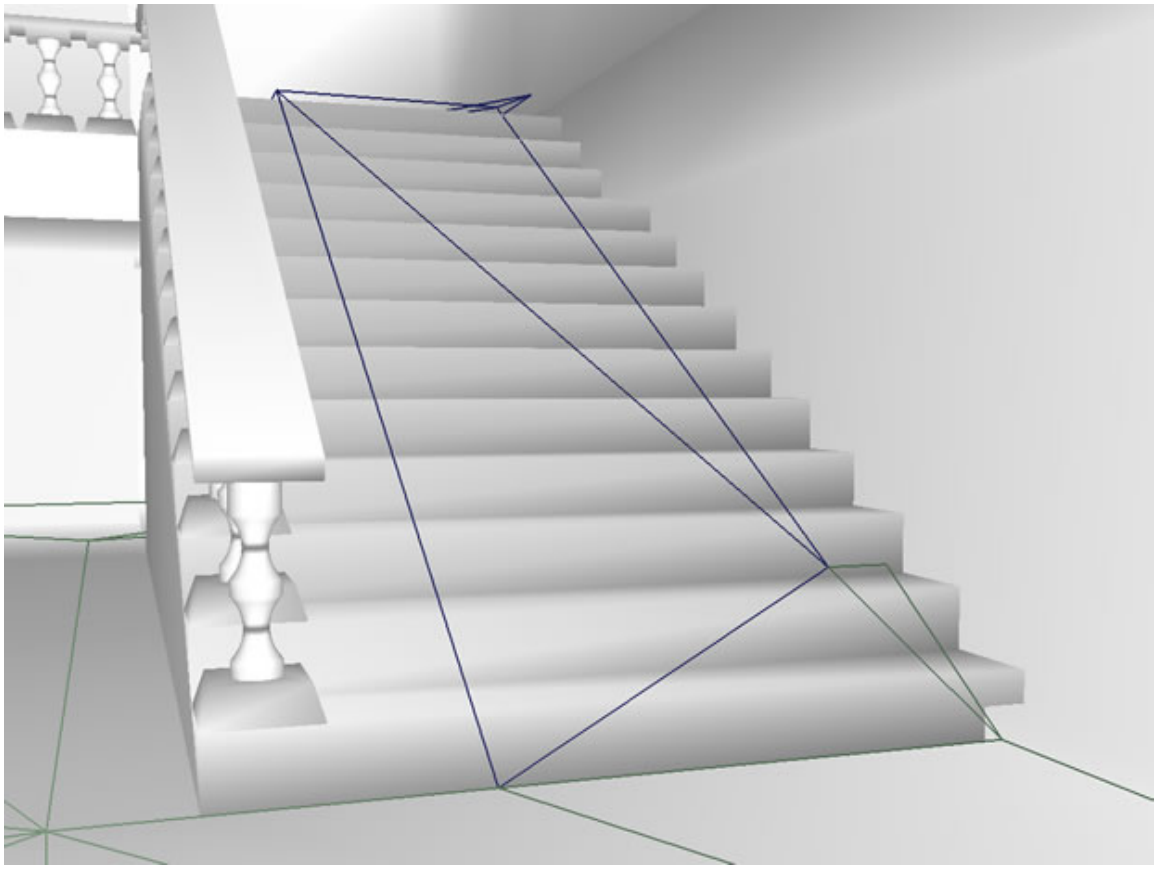


Figure 8: Setting a correct value for the maximum traversable step, elements such as stairs can be detected and made walkable

- **Maximum traversable slope**: Represents the maximum slope of a ramp that is still considered traversable. Any ramps with a higher slope will be considered unwalkable and will be considered an obstacle by the navmesh.
- **Border size of traversable area**: Represents the distance from the walls to the actual walkable area. For a well-formed navmesh, this should be at least the same size as an agent's surface area.

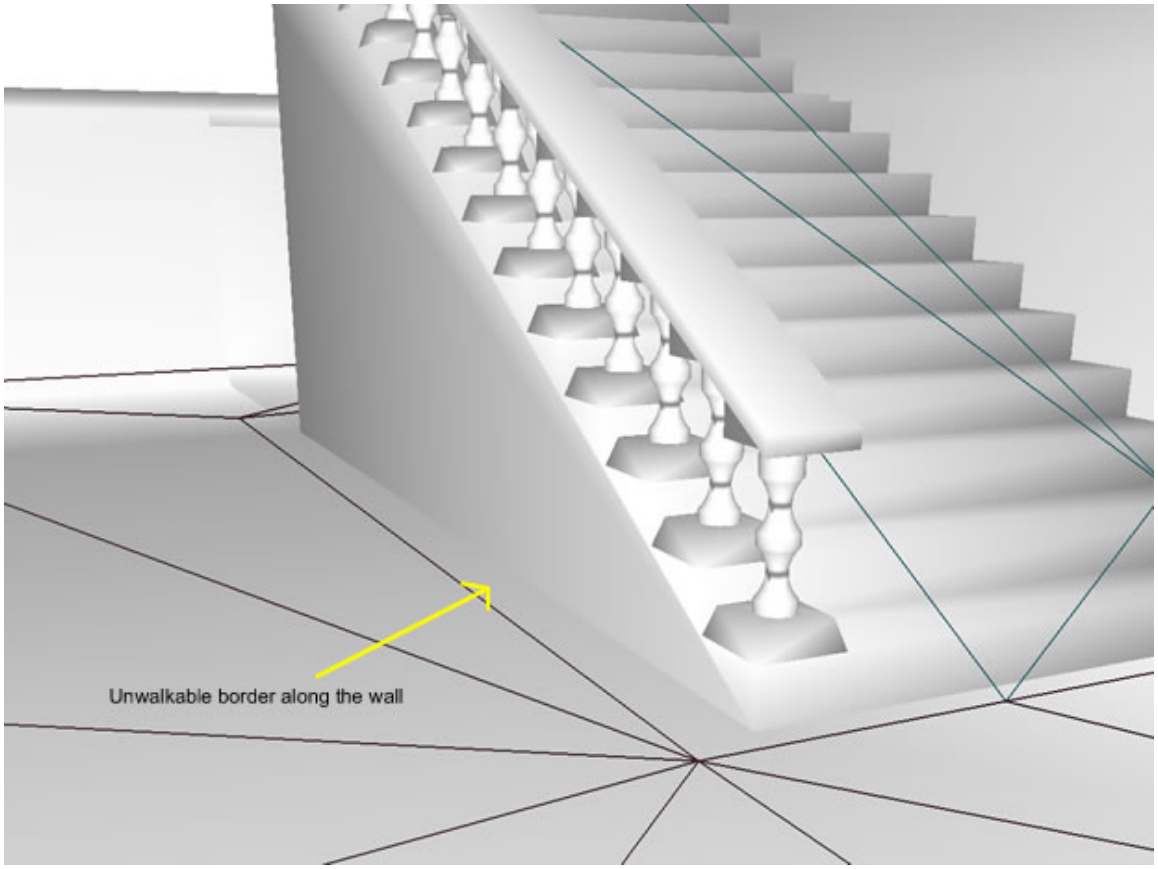
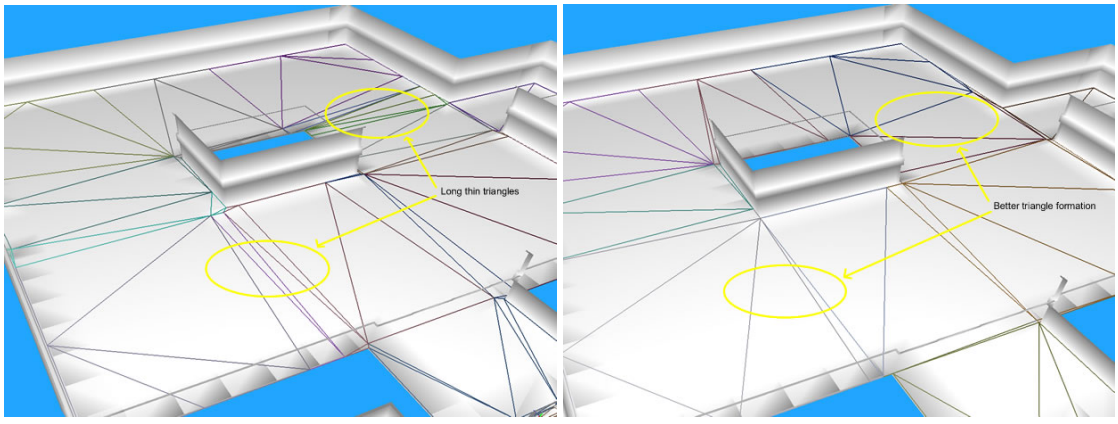


Figure 9: The walkable surface will be limited to at least the size of the border. Setting the border size to at least the size of an agent's surface area enables the agent to be positioned on any point of the mesh, without worrying of contact with the walls

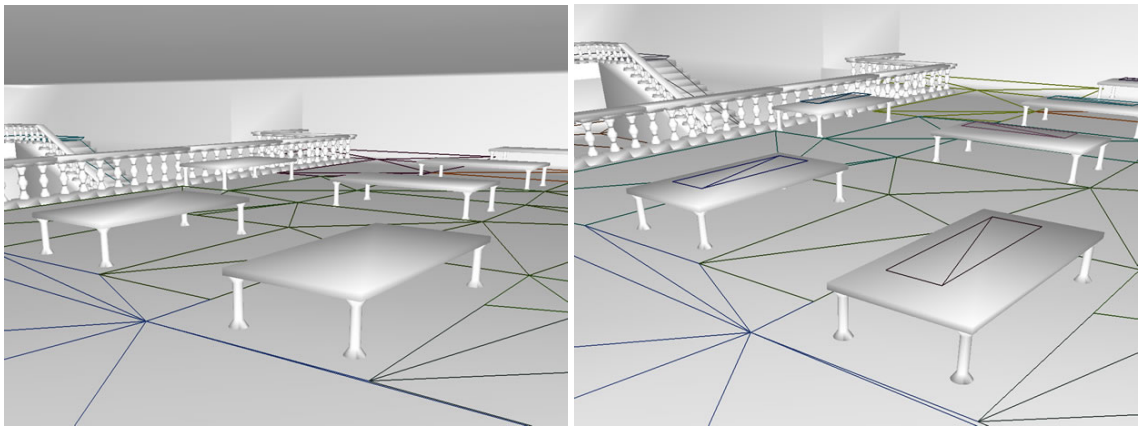
- **Smoothing threshold**: The amount of smoothing (larger region size, fewer thin triangles) to be performed when generating the distance field



(a) Smoothing disabled (b) Smoothing enabled

Figure 10: Smoothing threshold

- **Minimum size of unconnected region**: Represents the surface, in voxels, of regions that should not be added to the navmesh if they are unconnected to other regions (i.e. islands).



(a) A correct value for the minimum unconnected region size does not create a mesh surface for the table tops

(b) Setting the value of the minimum unconnected region size too low, the table tops are interpreted as walkable surfaces

Figure 11: Minimum unconnected region size

- **Merge region size**: Represents the minimum number of voxels a region should have in order to not try and merge it with other adjacent regions. This influences the number of long thin regions.

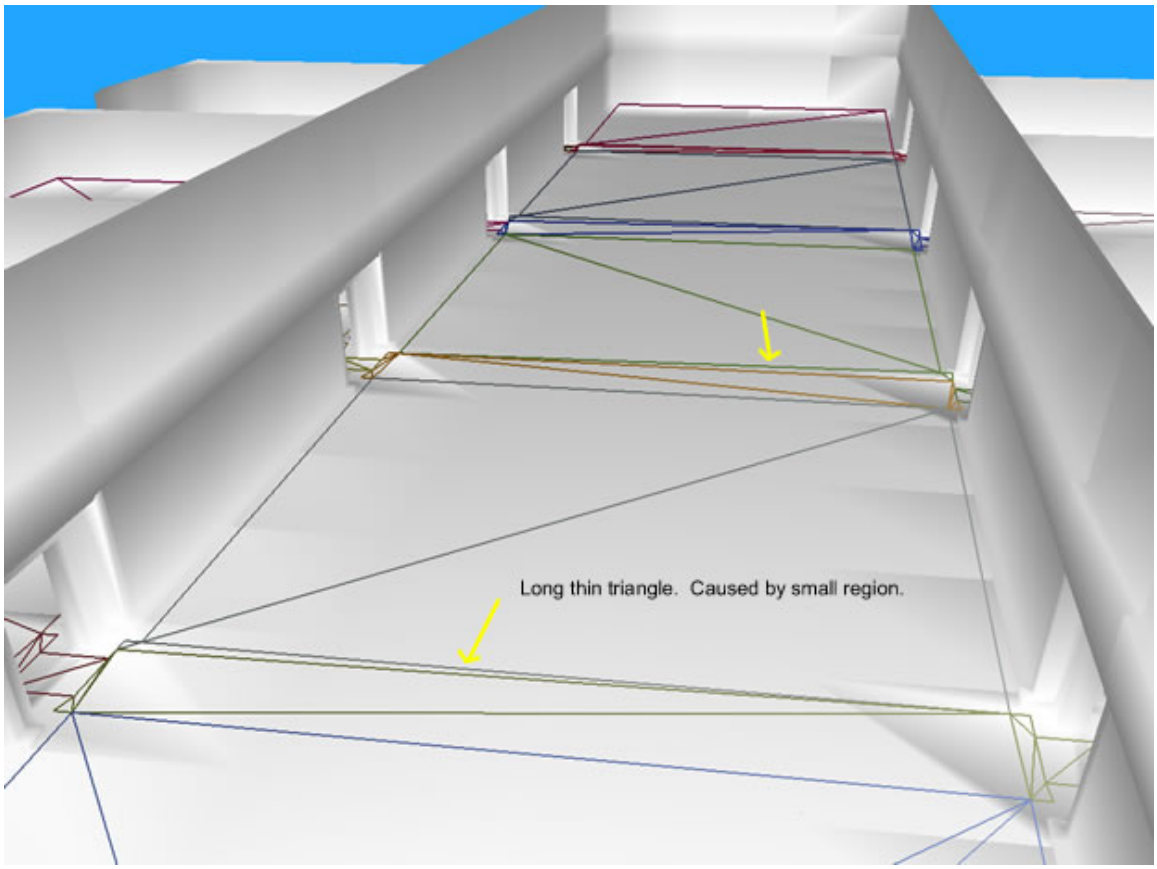


Figure 12: With merge region size set too low, the surfaces can be covered by a large number of long thin triangles

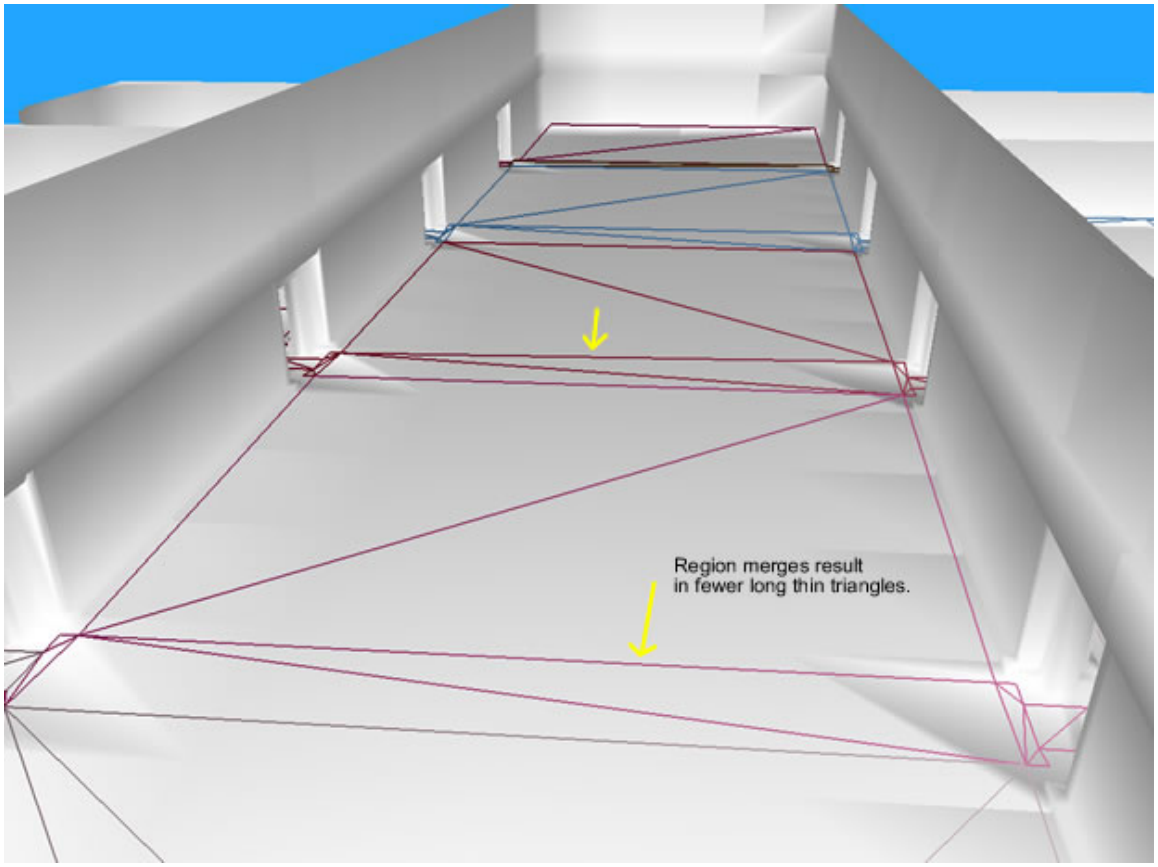


Figure 13: Setting the merge region size to a large enough value, some of the thin triangles are merged together with adjacent triangles (where possible)

- **Maximum edge length**: Represents the maximum length of a triangle's edge. If a triangle has a greater length, it will be split into several legal triangles, by adding more vertices on the border edges.

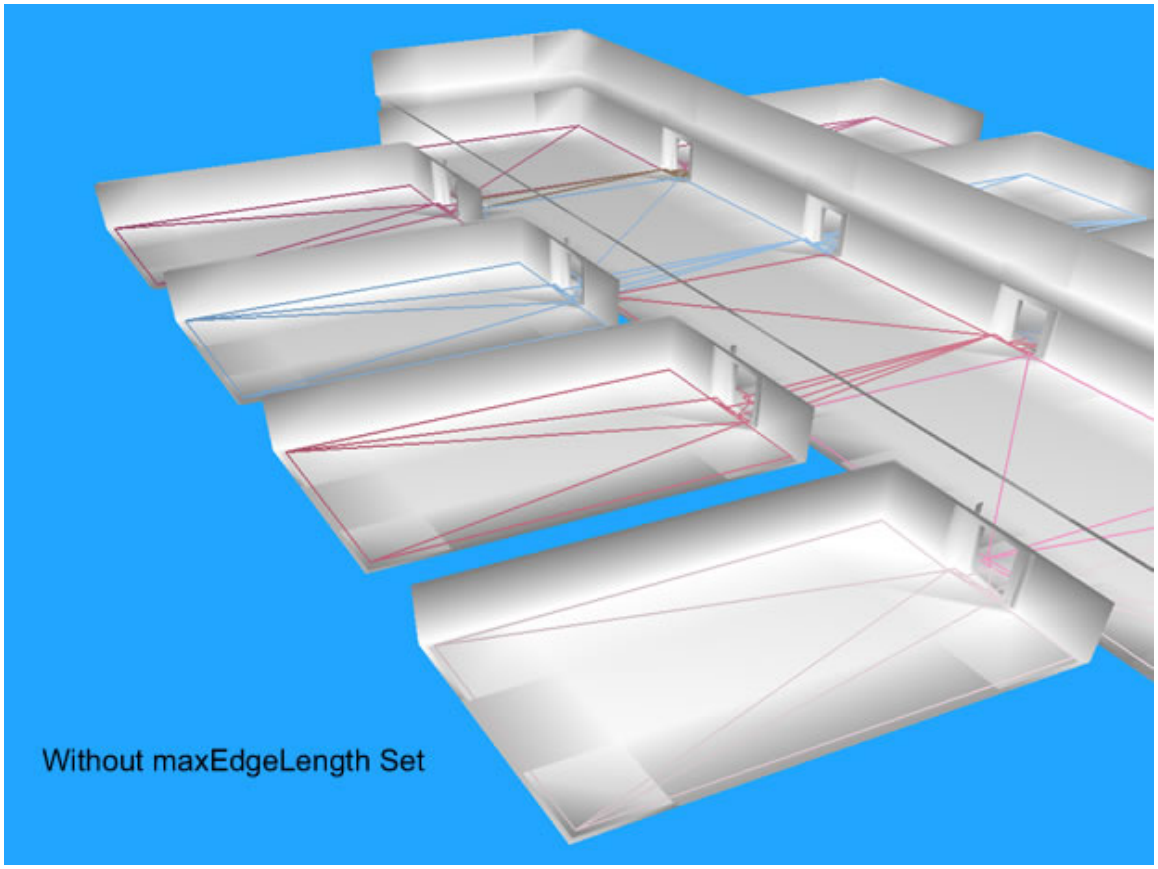


Figure 14: With the maximum edge length setting disabled, very long triangles can be generated

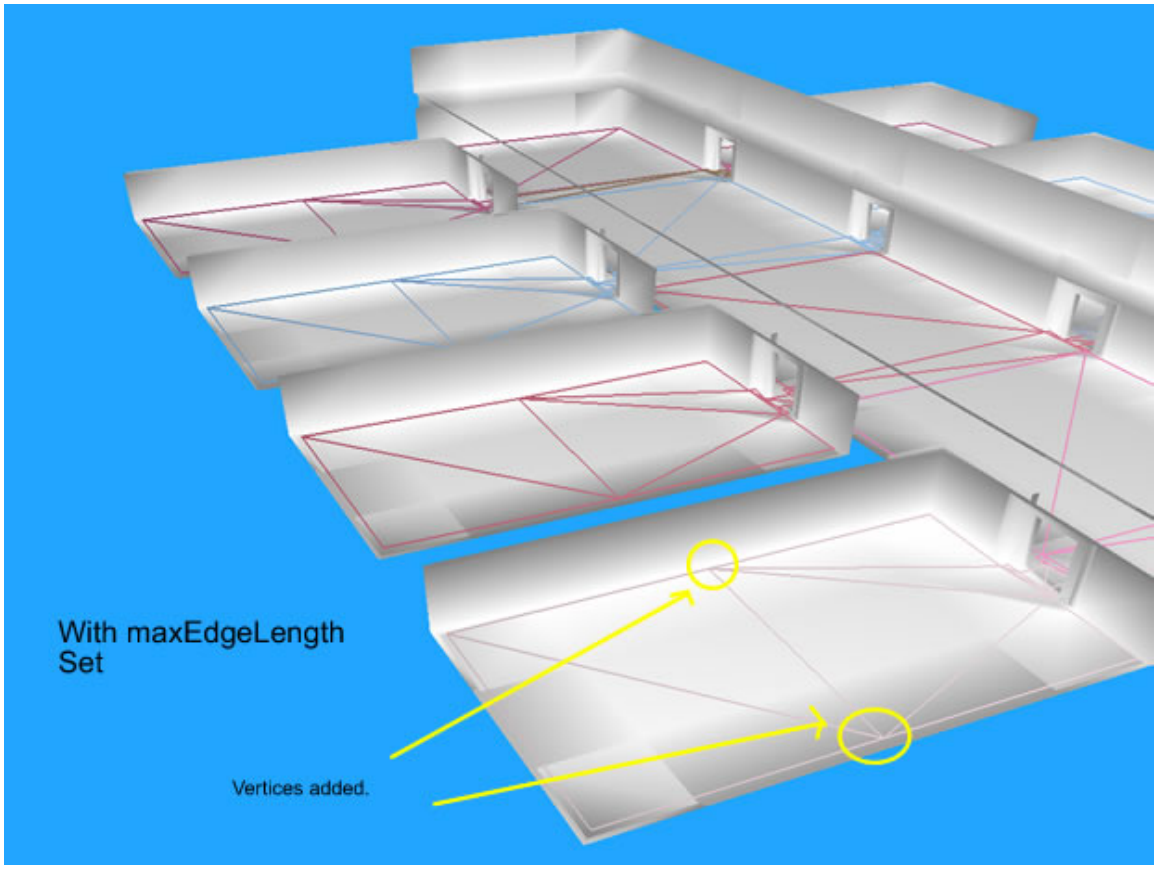


Figure 15: With the maximum edge length setting enabled, more vertices are added along the edge of the initial long triangles

- **Edge maximum deviation**: Influences the accuracy of the edges following the source geometry. The lower the value, the higher the accuracy, but at increased triangle count and processing cost.

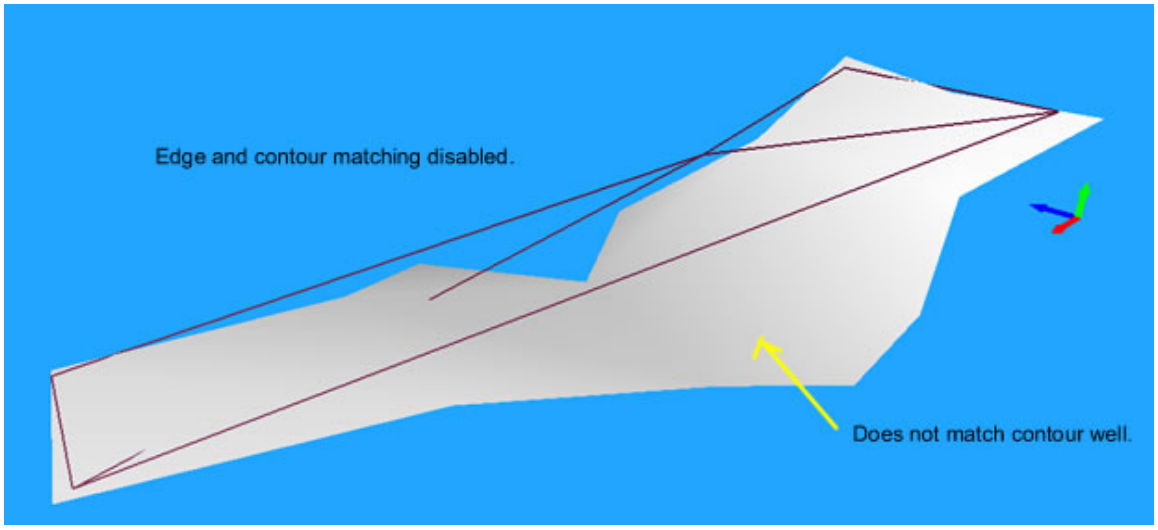


Figure 16: Edge maximum deviation disabled

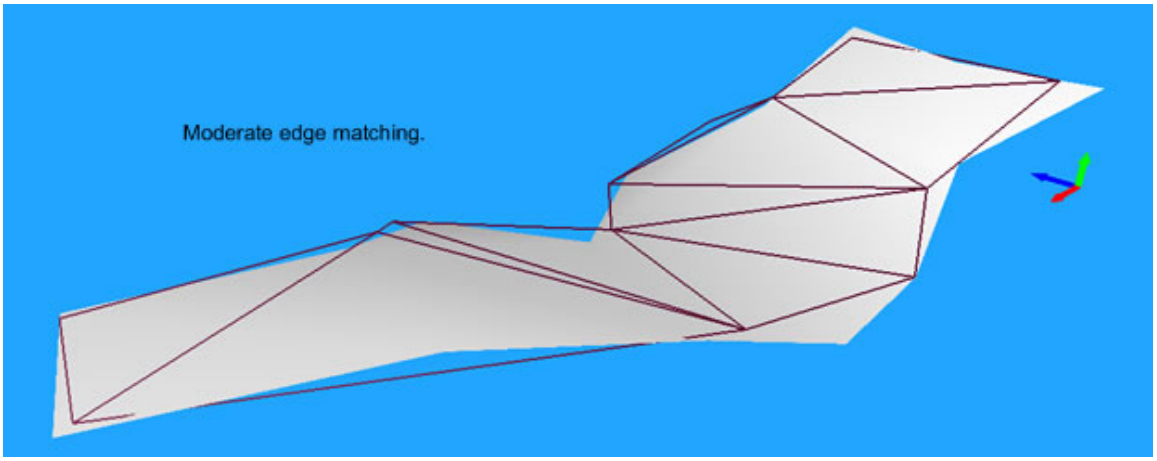


Figure 17: Moderate edge matching

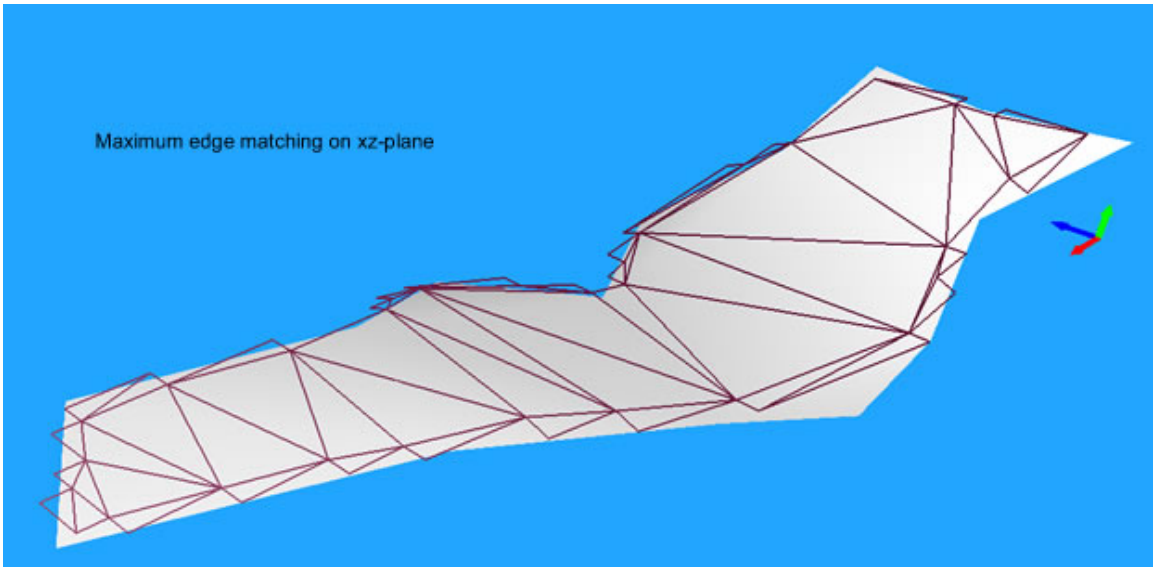


Figure 18: Higher fidelity edge matching

- **Max vertex per polygon**: If set to higher than 3, increases the cost of the computation but can result in better formed regions. However, for the current project, a value of 3 should be used, since the algorithms designed for path-finding only support 3-sided cells.

The problem that arises with such a high number of variables is the fact that end-users will not be able to import new models from different evacuation environments without a significant overhead. If wrong parameters are used for the mesh generation, the set of algorithms used might generate badly formed meshes, which do not map the source environment closely enough or even fail all-together. Still, with proper scaling of the model, either in blender stage or after importing the geometrical mesh, importing models of different magnitudes can be achieved. Since the space available for this report is limited, navigation mesh generation was presented only succinctly. For more in-depth information about each of the steps taken and how the previously discussed parameters influence the resulted mesh, the reader can refer to the NMGen Project documentation. reference

0.2 Navigation

0.2.1 Logical structure

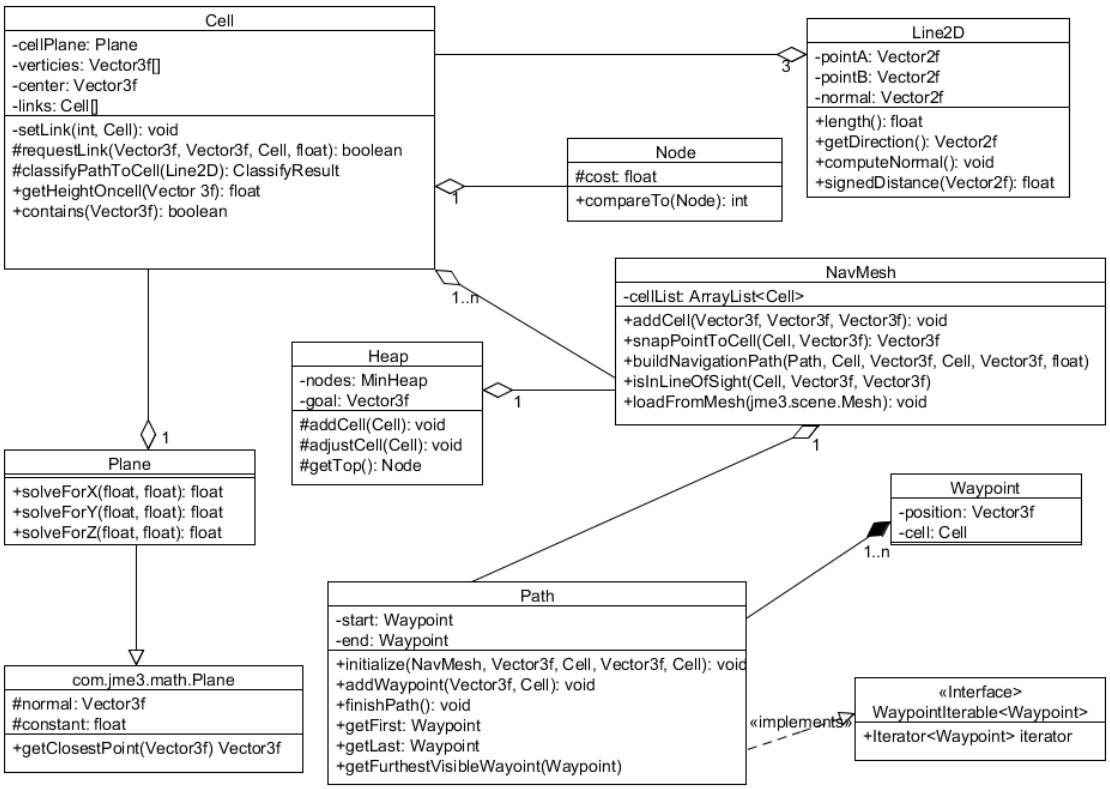


Figure 19: Navmesh Package

Figure 19 shows the relationship between the various classes responsible for path generation. The NavMesh class contains a list of Cells, each being a triangle representing a piece of the surface mapped by the mesh. Each cell has references to its three vertices (of Vector3f type), its three edges (of Line2D type) and its corresponding neighbours, or links (of type Cell) - three or less, depending on the number of neighbours. Each cell also has a corresponding plane reference, used to help with computing 3D calculations. The NavMesh initialisation is made by calling the loadFromMesh method, with the mesh resulted from the previous generation step as a parameter. As a result, the cell list will be populated with the relevant data.
Upon creation, each agent object representing a person will be assigned a new Path. By initialising the path with the relevant NavMesh reference, each agent can then independently use the reference to call the NavMesh's buildNavigationPath method. While calling this method, the Path in which to write the result is passed as a parameter, along with the start-point vector and end-point vector, the respective cells in which each vector reside, and a floating point number representing the size of the increments in which to place waypoints. More details about the way the path is calculated will be provided in the next section 0.2.2.
Once the buildNavigationPath has returned control to its caller, the Path passed as a parameter will contain a list of Waypoints - a tuple of intermediary vectors, in increments of the floating point

number provided, along with the cells in which each vector reside. Since Path implements the Iterable interface, the agent owning it can simply iterate over the waypoints contained and move along them sequentially.

0.2.2 Pathfinding

The buildNavigationPath method from the NavMesh class is in charge of the actual pathfinding routine. Since the navigable surface of the environment is mapped into cells which link to their neighbouring cells, this data structure can be regarded as a non-oriented graph. Therefore, graph pathfinding algorithms like Dijkstra[**?**] or A*[**?**]. While the number of agents required for the initial environment is small (200), A* would be a more scalable option, being faster than Dijkstra thanks to heuristics.
The pathfinding method used in buildNavigationPath is using A*: a priority queue structure (Min-Heap) is used to store intermediary paths from the source to destination, sorted by their lowest expected cost.

//a bit more to add, tomorrow