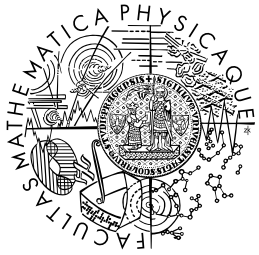


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Luis Sanchez

Generating High-Precision Navigation Mesh

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Jakub Gemrot, Ph.D.

Study programme: Computer Science

Study branch: Computer Graphics and Game Development

Prague 2021

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my supervisor Mgr. Jakub Gemrot, Ph.D., for introducing me to the topic of navmesh generation, providing advice and feedback. I also thank to my colleague Juraj Blaho for his consultations and advice.

Title: Generating High-Precision Navigation Mesh

Author: Bc. Luis Sanchez

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jakub Gemrot, Ph.D., Department of Software and Computer Science Education

Abstract: Navigation meshes are a widely used method for representing the world geometry in a format that can be used by pathfinding algorithms.

Frequently used navigation mesh generation algorithms first discretize the input geometry into a grid of voxels and then reconstruct the mesh out of them. This benefits the simplicity and performance of the algorithm, but comes with drawbacks. If the voxels are too large, the navigation mesh is not precise enough and may even have some pathways missing. If the voxels are too small, creation of the mesh takes too long.

In this thesis we propose and implement an algorithm that creates a navigation mesh directly from the input geometry without using an intermediate voxel representation. This allows us to preserve original detail where required and results in a more precise navigation mesh.

Keywords: navigation mesh, stitching, simplification

Contents

Introduction

In this chapter we explain what navigation meshes are and how they are used in pathfinding. Then we set up goals and desired properties of a navmesh generated by our algorithm. Finally, we provide an outline of the text in this thesis.

Background

In many applications from robotics to simulation, animation and computer games it is necessary to calculate a traversable path between two points in a scene. Such calculations often have to be done quickly, in order to provide a real-time response to the user.

Doing pathfinding directly on the world geometry is usually too unwieldy. The level of detail may be too fine and there could be defects, that while not being problematic in other computations like physics or visibility checks, would be hard to account for in pathfinding.

Examples of such defects can be small holes or intersections between individual parts of the world geometry. Even though a two edges of a mesh may be topologically disconnected, the path over them may still be valid.

Furthermore, there may be other requirements on the properties of a path that might be hard to guarantee. Common case is the need for a path have a sufficient size for an agent to move through.

To reduce the complexity and increase performance, the pathfinding is usually done on some form of a simplified world representation. Navigation meshes, as first proposed by Snook [2000], are one of them.

Navigation meshes

Navigation mesh (or navmesh), originally described by Snook [2000], is a connected mesh of convex faces. Together, they represent an area in which objects of the world can move. The entire mesh is contiguous, unless two parts of the world are disconnected. The mesh cannot be self-intersecting.

Usage With these properties, every walkable point on the surface of a scene can be mapped to a point inside a face of the navmesh. Then the pathfinding problem in the 3D space is reduced to finding a path along the surface of the mesh. Pathing within a single face is trivial – every two points can be reached by a straight line. When pathing between two different faces, first a path across the face edges has to be found. This can be done by graph traversal algorithms like A* by Hart et al. [1968] and then the path can be refined for example by a funnel algorithm described by Hershberger and Snoeyink [1994]. Alternatively, there are other algorithms that produce paths with desired properties.

Format In the most simple case navmeshes are made out of triangles. Because they have fixed number of sides, they can be easily stored in memory and working with triangular meshes is often simpler. However, it may be beneficial to use bigger convex polygons instead. This would reduce the number of elements

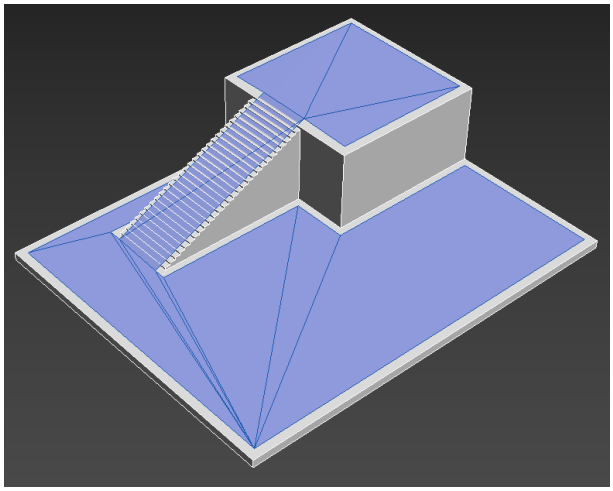


Figure 1: Navigation mesh. Stairs are represented by a ramp.

in the mesh and make areas with trivial pathfinding bigger. Of course, this would require a more sophisticated data representation and would add some additional algorithm complexity.

Wall clearance Optionally, the navigation mesh can exclude areas too close to a wall, where an agent would not be able to move because of its size. This allows to quickly produce paths where there is no risk of the agent bumping into a wall, or getting stuck when trying to walk through a tight corridor. However, to do this, the size of the agent has to be known at the time of navmesh generation. Additionally, two agents of different sizes could each need their own mesh, or would have to use the mesh of the bigger one.

Even when the size of the agent is not accounted for, there are ways to compute a path with arbitrary clearance from walls. One such is described in Shortest Paths with Arbitrary Clearance from Navigation Meshes by Kallmann [2010].

Navmesh generation Several good works about generating navigation meshes have already been written, some of them will be introduced in Related works chapter. Generally, they can be divided into two groups: The ones that create the navmesh from input geometry directly, and the ones that first convert the input into some kind of an intermediate representation, from which the navmesh is then built out of. For example, the intermediate representation can have the form of voxels. This approach is used by Mononen in Recast library (Mononen [2009]).

Using an intermediate voxel representation reduces the complexity of the scene and can hide some defects. Specifically, multiple intersecting faces in the input geometry can all end up being represented as a single voxel in the intermediate representation. However, in the same way, an important detail may be lost. For example, a narrow pathway that is not aligned to the voxel grid may become so narrow that it will not allow passage in the mesh. This can be fixed by reducing the voxel size, but making the voxels too small makes the generation take a long time.

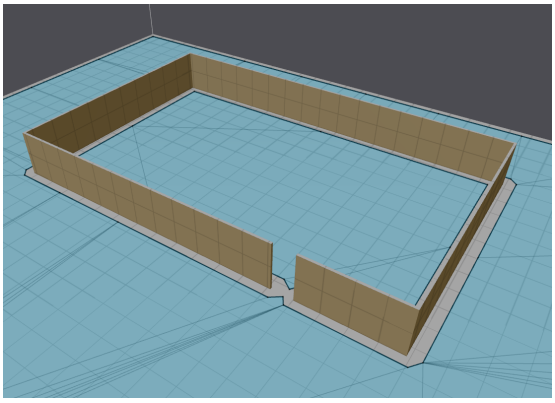


Figure 2: Detail missed by voxelization.

Goals

While there is a readily available open-source library for generating navmeshes (Recast, details in Related works chapter), the algorithm it uses works on a voxel grid and it suffers from the issues mentioned in the previous section. In this thesis, we propose and implement a different algorithm, one that generates the navigation mesh from the input data directly. This will allow to keep high detail until the later stages of the algorithm, where the unnecessary parts can be simplified.

Requirements We start by listing the requirements we put on our algorithm.

Input The most generic representation of any polygonal scene is a triangle soup – an array of triangles with no requirements regarding their position or topology. By taking simple list of ungrouped triangles as input we will place the least restrictions on input data.

Output In one of the previous sections we described two options for the structure of the navmesh – polygons or triangles. Our algorithm will return the results as a triangular mesh, because it is more general. If a polygonal navmesh is required instead, it can be created by running Hertel-Mehlhorn algorithm Hertel and Mehlhorn [1983] on the triangular one.

Detail preservation Throughout the algorithm, we will try to preserve the original detail, simplifying only once the topology of a surrounding area is known or when the data is too fine for numerical stability.

Simplification Having too many faces in mesh could quickly degrade performance of pathfinding. In order to keep the triangle count low, we will need to run some sort of simplification algorithm. However, the while doing so, we have to take care not to remove any detail significant for the navmesh.

Features In the following paragraphs we describe the features included in the navmesh.

Vertical clearance Some of the navmeshes do not prevent an agent from moving in places with insufficient vertical space. This can result in invalid paths and the agent getting stuck. In order to prevent that, we will allow only faces with enough vertical clearance to stay in the final navmesh.

Ramps To allow an agent to travel between two plateaus of different heights, they have to be connected. Such connection can be made by inserting an extra geometry, that will form a ramp between two existing faces. The ramp has to be properly topologically connected to the rest of mesh.

Error tolerance Some of the input geometry may have gaps, self intersections and T-junctions. These all have to be fixed in order to keep maximum connectivity between faces.

Wall clearance To prevent the agent from moving in too tight paths, we will exclude areas too close to a walls or overhang from the navmesh. While this can be also accomplished algorithmically during pathfinding by an algorithm from Kallmann [2010], having the actor radius already baked-in into the mesh is more performant.

Thesis Outline

In the first chapter we will describe two existing navmesh generation algorithms and show cases in which they fail.

The second chapter introduces our navmesh generation algorithm. First we begin by presenting a high level overview and break down the algorithm into steps. Then we describe the individual steps in more detail, with each of them having its own section.

In the third chapter, we will show the results of our work and compare the navmeshes generated by our algorithm against a one that uses voxelization.

Finally, in the fourth chapter we mention suggestions that can be done to improve the results of our algorithm.

1. Related works

In this chapter we describe some existing navmesh generation algorithms and discuss problems they are unable to solve.

1.1 Voxel-based algorithms

Some algorithms make use of voxelization step during the navmesh generation process. Instead of working on the geometry, such algorithms create the mesh from a voxel grid. The voxel grid has uniform detail and hides defects or imprecisions of the original input geometry. However, any detail that cannot be represented on the voxel grid will be lost.

1.1.1 Recast

Example of a voxelization based algorithm is a one used by a state of the art open-source navmesh generation library, Recast Mononen [2009]. The original implementation was done by Mikko Mononen, and the inner workings are described in a presentation on his blogpost Mononen [2010]. In the following text we present a brief overview.

Polygon voxelization Polygons of the input geometry are clipped into cell boundaries. Then they are rasterized by taking a min and max values in each cell column. Finally, they are merged into RLE encoded voxels.

Building navigable space Filtering removes voxels with overhead obstacles or steep slopes. Compensates for conservative rasterization by doing erosion of ledge cells.

Watershed partitioning Watershed partitioning splits the navigable space into regions. Optionally, the regions are eroded by the agent's radius

Region tracing and simplification Contours of the regions are found and traced, then simplified. At this stage, the navmesh is a set of simple polygons, with shared vertex positions.

Triangulation and connecting Simple polygons from the previous step are broken down into triangles. Finally, they are connected into the mesh.

1.1.2 Drawbacks of voxelization

Voxelization sets a limit on detail Voxelizing the input into cells will remove any detail that is smaller than the size of a cell. Even features larger than the grid size may be distorted if they lay across cell boundaries. Additional processing in voxel space, like the 1 cell erosion done by Recast will further worsen the impact.

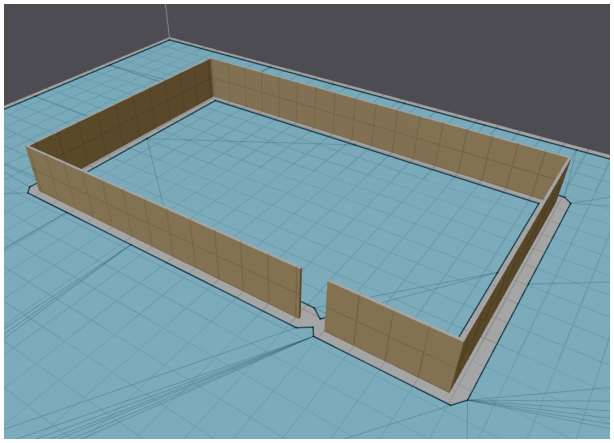


Figure 1.1: Detail lost by voxelization

In figure 1.1, the door frame is 30% wider than the agent and covers the length of approximately 9 voxel cells. But due to the loss of detail caused by voxelization, the navmesh inside the room is not yet connected. Such disconnect will prevent pathfinding algorithm from finding a path into the room. To guarantee connectivity we would need to further reduce the cell size.

1.2 Algorithms that use the geometry directly

In this section we describe an existing algorithm that works directly on the input geometry.

1.2.1 Tozour's Near-Optimal Navigation Mesh algorithm

Tozour in an article inside AI Game Programming Wisdom describes an algorithm for building a near-optimal navigation mesh. It builds on the concept of navmeshes introduced by Snook [2000], but instead of building a triangular mesh, he creates a polygonal one that is sufficiently close to optimal in number of faces.

Slope filter The first step of Tozour's algorithm is to filter away any polygons that are not walkable. This is trivially done by comparing the normal vector of every face against the up vector of the scene. Any polygons with values over a specified threshold will be rejected.

Merging into convex polygons

Neighbour merging Neighbouring triangles are merged into bigger convex polygons using Hertel-Mehlhorn algorithm Hertel and Mehlhorn [1983], which guarantees that it has at most 4 times the number of polygons of the optimal one.

3-2 merging Some polygon merges done in the previous step can be further improved by breaking down three neighbouring polygons and merging them into two.

Culling Faces with surface area under a limit are removed. No preserving of topology is mentioned, so handling of small faces that connect important navmesh areas is a bit unclear.

Handling superimposed geometry This step is meant to cut away parts of navmesh that are being blocked by superimposed geometry like static objects. Possibly it could be used to also cut away parts of the mesh that are being blocked by other navmesh faces, but Tozour [2002] does not mention such use.

The cutting is being done on per-face basis, where each face is recursively subdivided until limit size is reached. If any of the new faces created by subdivision are fully blocked by the other geometry, they are cut from the navmesh and the subdivision is stopped.

Finally, the remaining faces left after subdivision are merged using the same approach as described in Neighbour merging.

Issues with Tozour's algorithm The algorithm does not handle self intersections in the input data. When two walkable surfaces meet vertically, only one of them or its part should remain valid. (The other one is under it). This frequently occurs when terrain surfaces are not exactly aligned or are being intersected by ramps.

In theory, this could be handled in the same way as superimposed geometry, but then the original polygon merging would be wasted and the amount of subdivision could be enormous.

Additionally, the algorithm does not connect parts of navmesh that are not touching. Tozour [2002] proposes to solve this by creating "links" between two disconnected navmesh edges, but does not elaborate on the implementation further.

2. Our navmesh generation algorithm

In this chapter we present our approach to generating a navigation meshes that preserves detail of the original geometry. We will first provide a high-level overview of the algorithm and then we will describe its parts in more detail in their own sections.

2.0.1 Working with raw geometry

We do not want to limit the user too much by restricting the accepted input. Even if the objects of a scene are fully made of valid meshes, their combination in the scene might not. The objects might be intersecting with each other, segments of terrain may be overlapping or have gaps between themselves. In order to be able to accept such scenes as input of our algorithm, we cannot make too many assumptions about their structure. Therefore in the beginning, we will work only with the individual triangles, and only in the later steps we will merge them into a mesh.

While working with the raw geometry data has benefits, it also comes with its own challenges: Not being able to run some kind of a pre-processing voxelizing step that would clean-up the source data means that we need to handle any defects inside the algorithm itself. Some of these may be present in the input, but others may also be created by our transformations. Any computations done on floating point data types will eventually suffer a loss of precision and require the use of offsets and tolerances. Even then the loss of precision can then manifest in unexpected results at any time. We need to make an extra effort to handle degenerate cases at all stages of the algorithm.

2.1 High-level algorithm description

Our goal is to create navmesh out of unstructured triangle data (triangle soup). To achieve this, we will do a series of steps, each of which will transform the geometry, getting it closer to the result we desire. During the process, we will use the term navmesh candidate to refer to the mesh that is being made into the final navigation mesh.

2.1.1 Steps

Our algorithm can be broken down into the following steps:

Slope filter In the same way as Tozour [2002] does in his navmesh generation algorithm, we start by filtering away unwalkable triangles. We compare the normal vector of each face to the world up direction, and if they differ by sufficiently high angle, the triangle is deemed to be too steep to be walked on and is not kept in the mesh.

In our implementation, both the scene up direction and the maximum walkable angle can be customized by a user. Further details in Algorithm settings.

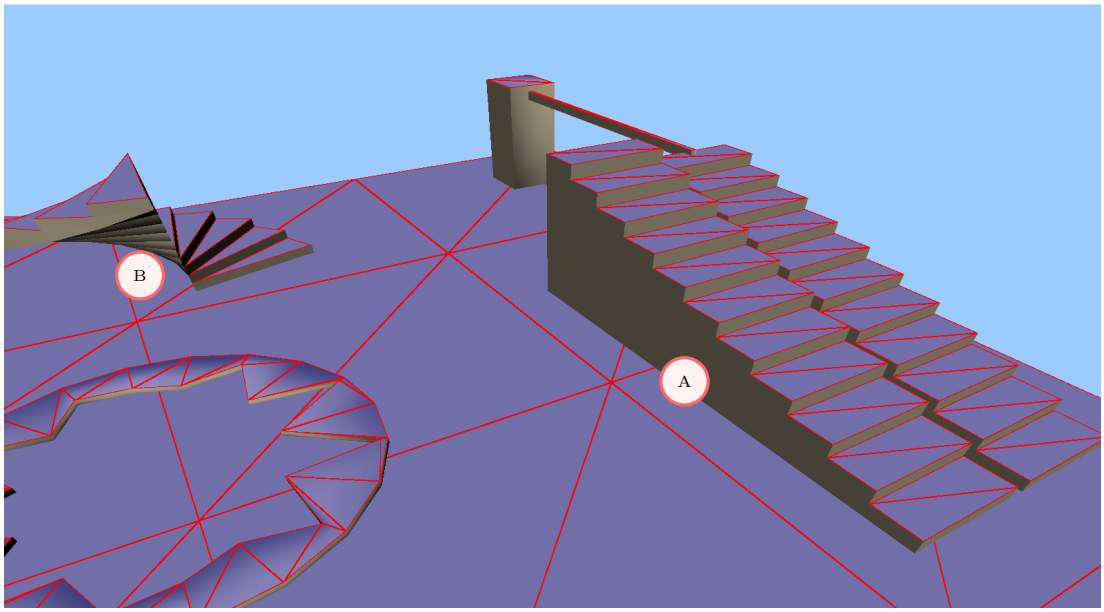


Figure 2.1: Mesh after slope filter

Cutting It can be seen that the results of the previous step contain parts that definitely should not be in a navmesh. The triangles of the floor continue unhindered through the stairs (A in the figure 2.1.) and there are places with insufficient vertical clearance (B in the figure 2.1.)

Both of these cases will by handled in the Cutting stage. We will extend every triangle from the original input into a 3D wedge, and use it to cut into the navmesh candidate. The length of such wedge is based on the height of the agent. Areas within the wedges will be discarded. Details are described in the Cutting section.

During this stage we also collect useful information about the individual edges that will be used later to help us create ramps.

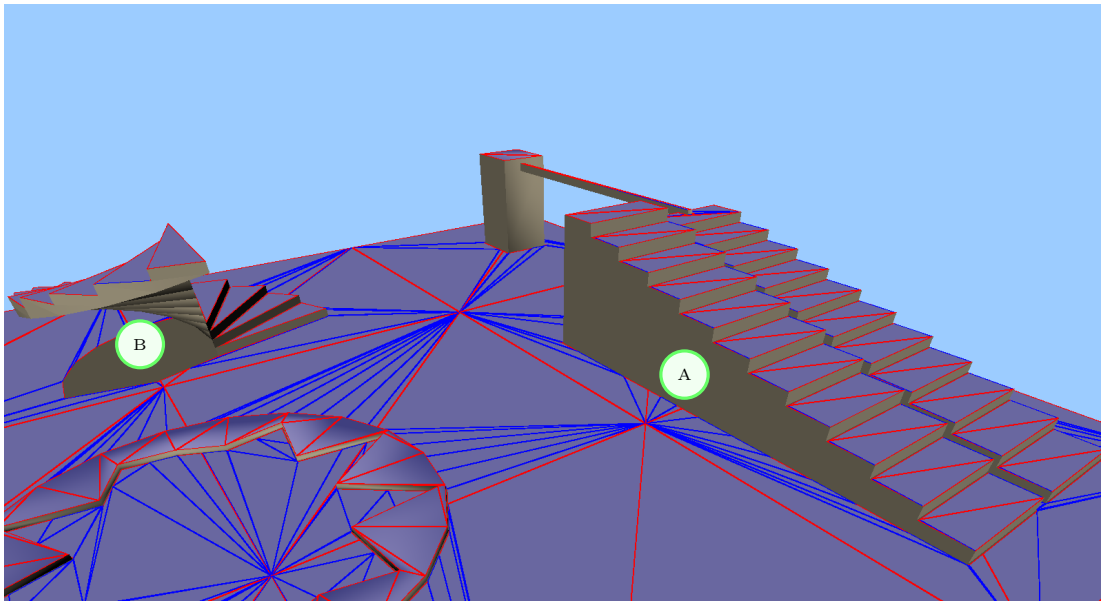


Figure 2.2: Mesh after cutting. A – ground triangles stop at the outline of the wall. B – area was cut because of insufficient space

Gap closing The blue lines in figure 2.2 represent edges that are shared between two triangles, while the red ones are boundaries of just one triangle. It is evident that there are many edges still unshared, even though they belong to two visibly neighbouring triangles. Until now, we worked on every triangle of the navmesh candidate separately and were not at all concerned about the topology. However, future steps will require a properly connected mesh, which we will now create in the gap closing stage.

The purpose of this stage is not just to merge the obviously neighbouring triangles that shared exact vertex positions, but to fix a wider range of errors that come from both the original input or could have been caused by the cutting. Specific examples can include T-junctions, gaps between edges, small intersections of neighbouring faces or extremely thin triangles.

The implementation is based on Progressive Gap Closing for Mesh Repairing by Borodin et al. [2002], which works by applying a series of vertex-vertex or vertex-edge contractions to close holes and fix small self-intersections in the navmesh. The original algorithm can sometimes produce non-manifold vertices and edges that would not be acceptable in a navmesh. To keep our mesh topology valid, we limit the algorithm to do only contractions that do not invalidate it.

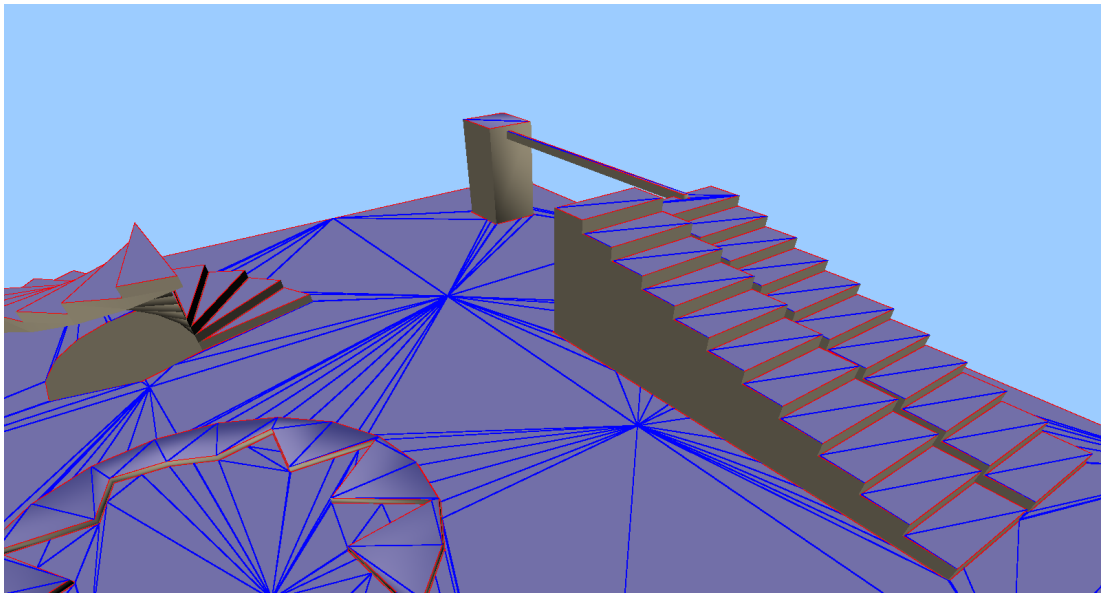


Figure 2.3: Mesh after gap closing. All internal edges of the floor and stair platforms are being shared between all triangles

Unnecessary detail removal We run a light decimation step, that removes unnecessary features from the mesh. This step only affects internal vertices of (almost) flat planes and consecutive vertices of boundaries. The goal is to reduce the complexity of our next operations without sacrificing important detail in the mesh.

Adding ramps During cutting, we stored a type information on each boundary edge that allows us to determine if an edge was created by cutting or if it is a one of the original boundaries. Now, that all the internal edges are fully connected, we proceed by creating ramps from all the non-blocked edges.

We attempt to connect every non blocked edge by a quad to an existing part of the navmesh below. We will keep only the parts of ramps that were able connect. The newly made ramps will become a regular part of the navmesh and will undergo the same cutting and gap closing procedures as the rest of the mesh did.

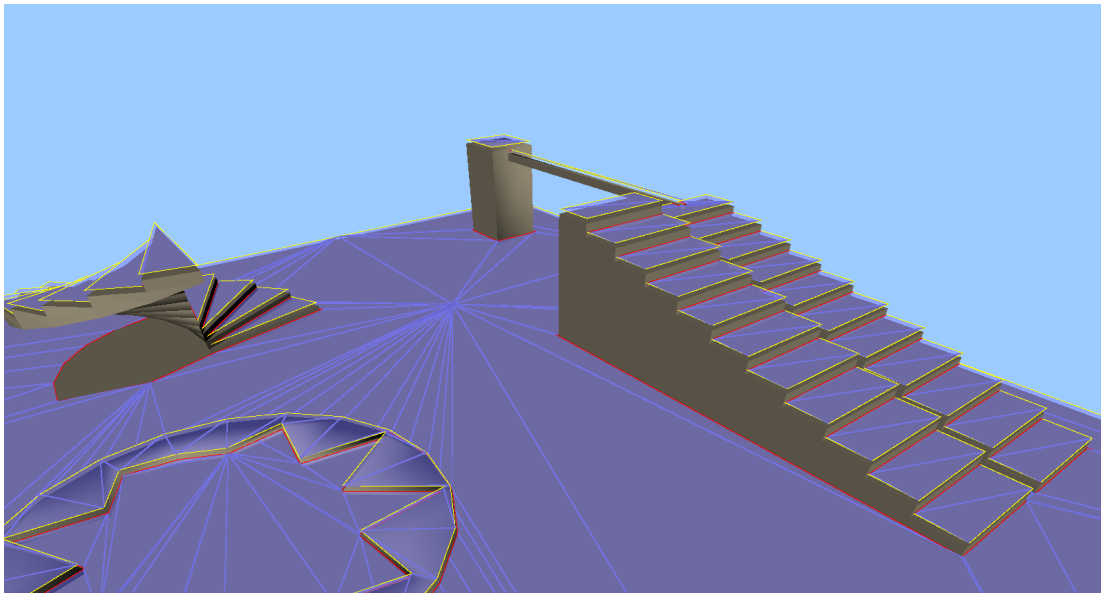


Figure 2.4: Edge type after removing unnecessary detail step. Yellow edges are not blocked and can have ramps.

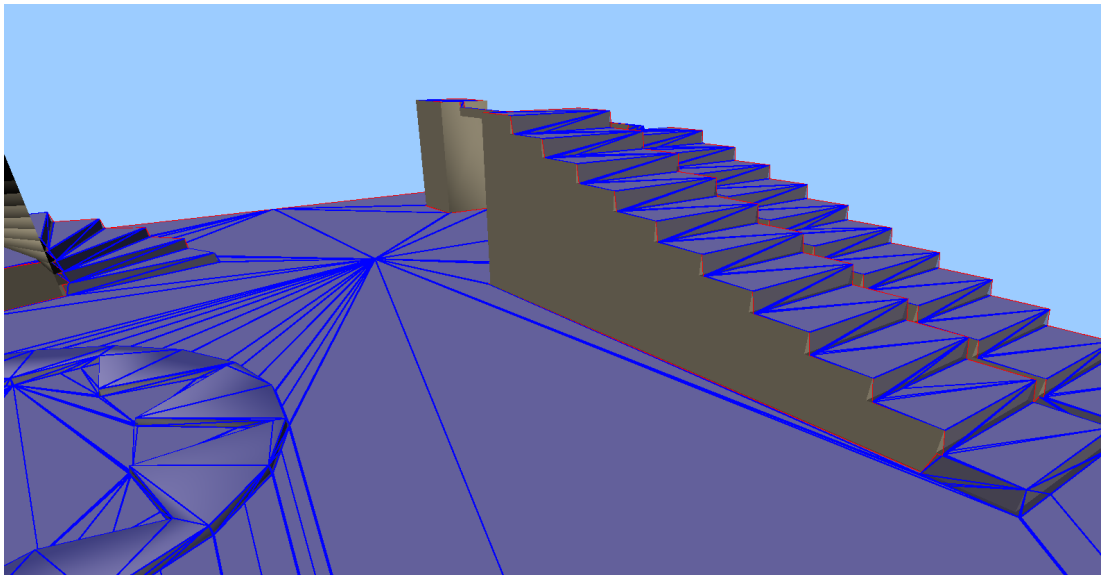


Figure 2.5: Navmesh after adding ramps.

Removing small patches Now that all of the previously disjoint parts of a mesh are connected into big chunks, we remove any remaining small areas. We calculate a combined surface area of every connected face group (patch) and if they are under a threshold, we remove them from the navmesh. This step is intended to delete small areas in unreachable places, such might be an area on top of a lamppost.

Flattening and simplification We run another decimation step. First pass attempts to flatten consecutive ramps into one shared ramp quad. A second

pass does a general mesh simplification while still preserving outer boundaries. To achieve this we will use a quadric error metric to prioritize decimations with the least detail lost, while locking important features to prevent them from being deleted.

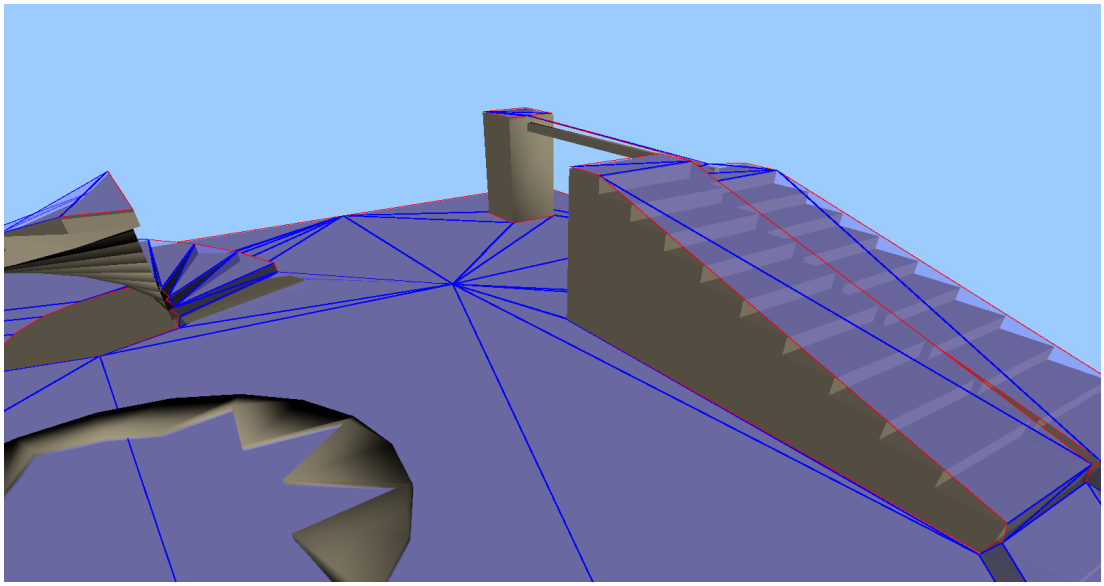


Figure 2.6: Navmesh after flattening and simplification.

Wall clearance This step cuts away areas of the navmesh that are too close to a wall for an agent to move in. It is done by inflating every existing boundary edge of the navmesh into a 3D shape and intersecting it with the remainder of the mesh in a similar way as we previously did during Cutting. In the end we also connect all newly cut triangles together again using Gap closing and remove newly created small patches in the same way as in Removing small patches.

Final simplification In the end we run a final pass of simplifying the mesh again using the error quadrics.

2.1.2 Algorithm settings

This section describes settings that are provided to the user to control the behaviour of our algorithm.

Agent specific settings These settings control the dimensions and properties of an agent that we create the navmesh for. We represent an agent as a cylinder with the following parameters:

Agent height Height of the agent in meters.

Agent radius Radius of the agent in meters.

Maximum walkable slope Maximum steepness of a surface that can be walked on.

Maximum step height Maximum height of a stair step that can be climbed by the agent.

Navmesh settings These are the settings related to the scene and the algorithm itself.

Up vector The vertical direction in the world coordinates.

Stair slope Default angle of ramp placement for stairs.

Gap closing distance Maximum distance of contractions during gap closing.

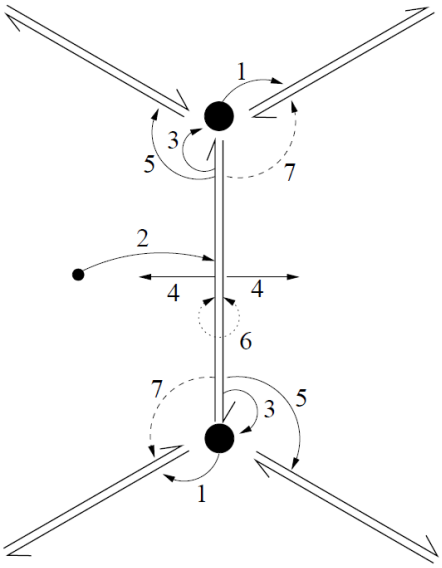
Wall thickness Additional thickness added to triangles that make up walls.

2.2 Fundamentals

We want to store a mesh in a form that allows us to easily traverse the topology and do changes to the mesh data. A suitable solution is a halfedge data structure implemented in the OpenMesh library by Kobbelt et al. [2002].

2.2.1 The halfedge data structure

The halfedge data structure is an representation of mesh data that splits every edge into a pair of directional halfedges and stores connectivity information in them.



1. Vertex → one outgoing halfedge
2. Face → one halfedge
3. Halfedge → target vertex
4. Halfedge → its face
5. Halfedge → next halfedge
6. Halfedge → opposite halfedge (implicit)
7. Halfedge → previous halfedge (optional)

Figure 2.7: Halfedge data structure. Description from Kobbelt et al. [2002]

The benefit of using this representation is that it allows constant time access to a one-ring neighbourhood of a vertex.

Limitations Some topologies cannot be represented via the halfedge data structure. Any edge must not belong to more than two faces, otherwise it becomes a *complex edge*. Every neighbourhood of a vertex must be topologically equivalent to a disc, otherwise it becomes a *complex vertex* Kobbelt et al. [2002].

Even though OpenMesh can handle some complex vertices, it cannot handle complex edges at all. A final navmesh will not contain either of these, but as we need to use the data structure also for the intermediate steps we must uphold these requirements during the whole algorithm.

2.2.2 Some definitions

In this section we define some terms that we will use to refer to features of a mesh represented by the halfedge data structure.

Halfedge

- *boundary halfedge*: A halfedge that does not have an adjacent face.
- *opposite halfedge*: A second halfedge of a halfedge pair that makes up a single edge.
- *next halfedge*: The next halfedge in a halfedge chain. Every halfedge points to its successor. Halfedges within a single face form a closed loop. Boundary halfedges form a loop around the whole boundary of the mesh.
- *opposite vertex* A vertex on the other side of the face attached to this halfedge.

Vertex

- *boundary vertex*: A vertex that belongs to a boundary edge.
- *isolated vertex*: A vertex that does not belong to any edge.

2.3 Import and slope filtering

Now we will look at the individual parts of the algorithm in more detail. This section describes the steps we take while preparing the initial navmesh candidate.

Filtering of valid triangles

Numerical validity For any triangle in the input we first test that it is numerically valid. We define a valid triangle as a one that is both made of vertices with finite coordinates and has a valid normal vector. If a triangle is not valid, it is simply ignored in the input data. The normal vector is not being read from the input, but is calculated anew from the positions of the vertices. We will store it along with the triangle to avoid having to calculate it again.

Loading into OpenMesh We want to use a halfedge data structure provided by OpenMesh to be able to work with the mesh. However, we placed no limitations on the input data - it may contain both *complex edges* and *complex vertices*, which the halfedge data structure is not able to represent. To get around this issue, we have to add every triangle's vertices independently even if they are located at the same coordinates.

This will make all the triangles disconnected from each other and not have topological conflicts. By doing that, we will also lose the ability to traverse the topology of the mesh. We will eventually fix this issue during Gap closing.

Slope filter As a starting point for our navmesh we want to keep only the triangles that an agent could walk on. We compare the angle between a normal vector of each triangle $\vec{n_i}$ and the up direction of the scene $\vec{u}$. A triangle is discarded if the angular difference of the two vectors is bigger then a threshold specified by *maximum walkable slope* from Algorithm settings. Such comparison can be done efficiently by setting a threshold of

$$t = \cos(\textbf{max walkable angle})$$

and rejecting any triangles where

$$\vec{n_i} \cdot \vec{u} < t$$

The same walkability filter of initial triangles is done by Tozour [2002] in his Building Near-optimal Navmesh algorithm.

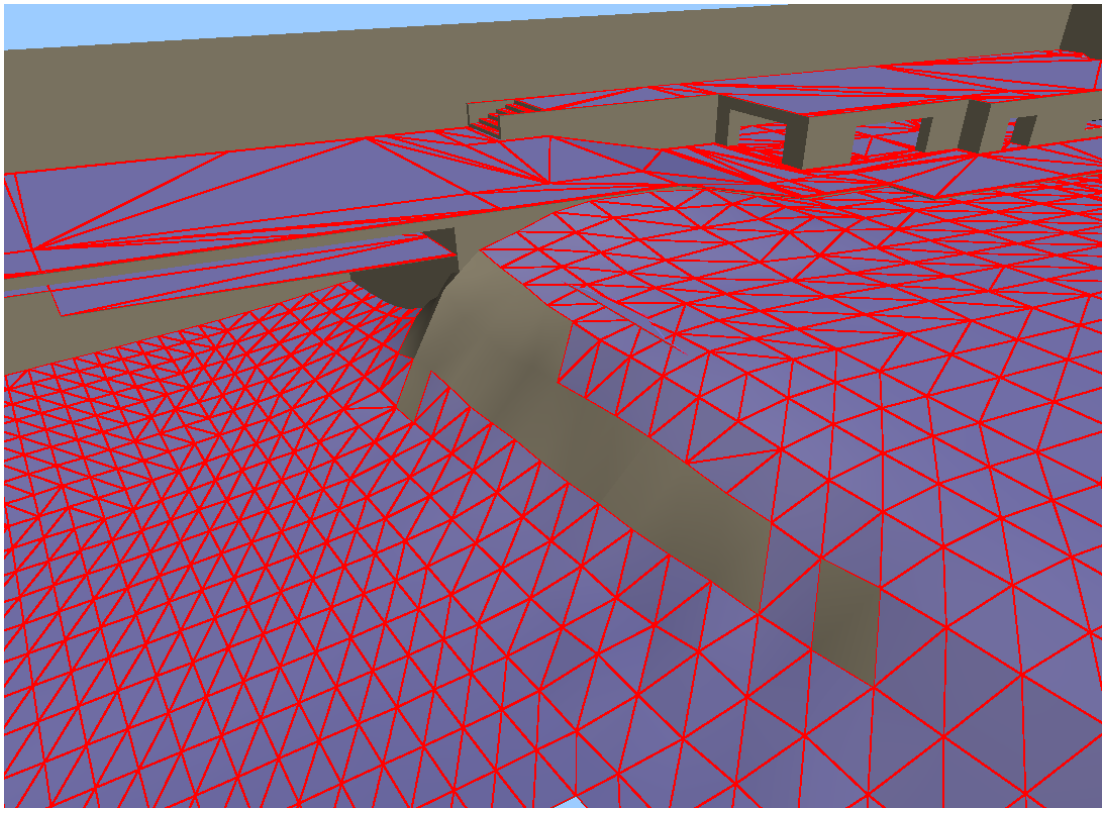


Figure 2.8: Triangles filtered by their slope.

2.4 Cutting

The navmesh candidate from previous step includes areas which are not accessible to an agent. There might be points where the navmesh continues unhindered through a wall or places where an agent could not possibly walk because of insufficient vertical space.

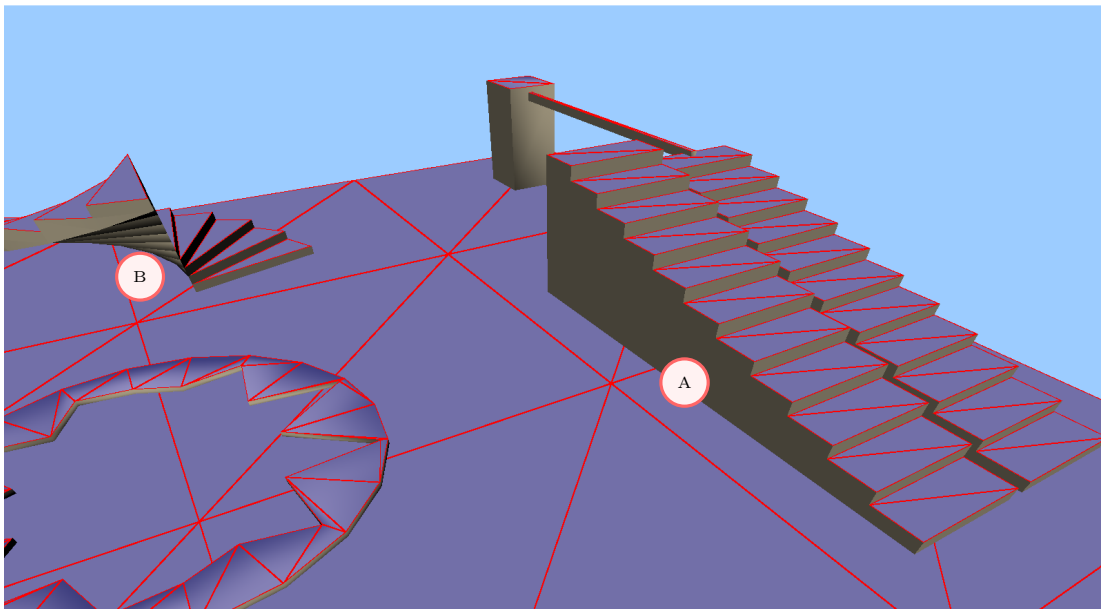


Figure 2.9: Defects in navmesh before cutting. A – navmesh continues through a wall. B – not enough headroom for an agent

The purpose of this step is to handle both such defects by cutting the problematic parts away from navmesh. We will first show how to handle the areas with lack of vertical space, then we will extend the approach to handle wall intersections as well.

2.4.1 Method

For a point to be vertically blocked there needs to be an obstacle above within a short enough distance that an agent would not fit in between. For a single point such case could be detected by a raycasting from the point upwards. However, for the whole triangle a single raycast would not be enough, and doing a sufficient number of them to cover the whole area is impractical.

Instead, we can take advantage of the fact that all of these raycasts would have the same direction and length. If we extruded all the obstacles by the agent's height in the downwards direction, we could simplify the tests in the following way:

Every vertically blocked point would lie within the extruded volume. Similarly, every vertically blocked triangles would be intersecting the extruded volume as well. If we are able to compute the triangle's intersection against the volume exactly, we can determine which parts of the triangle we should keep. See the figure 2.10 for a 2D illustration.

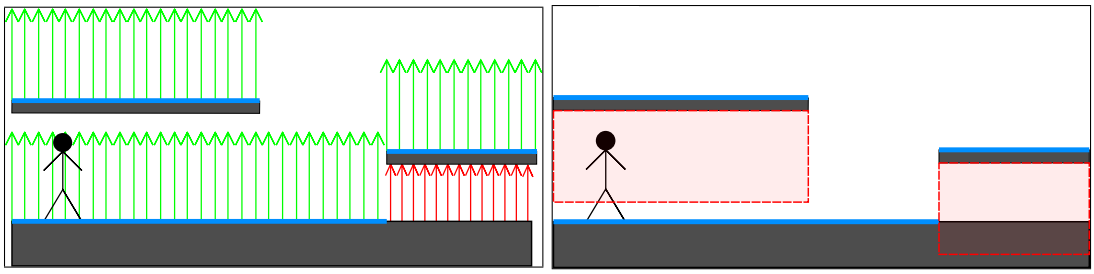


Figure 2.10: Raycasts vs extruding geometry downwards (2D).

In our case, all the obstacles are being represented by a set of triangles. Extruding a triangle by a non-zero vector that is not aligned to its plane will always produce a wedge (figure 2.11). The only way that the normal of a triangle would be aligned with our extrusion vector could be if the triangle itself is orthogonal to the up direction of the scene. We will consider such triangles to be walls and we will handle them later in the Walls paragraph.

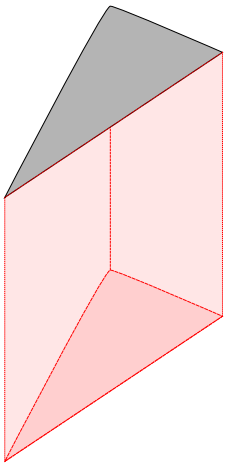


Figure 2.11: Extruding a triangle into a wedge

It is worth noting that the whole concept of extruding geometry is just a mental model used to help us understand the cutting operation. Extruding a triangle does not involve an actual change to the geometry. All wedges are implicit and can be fully represented by the original triangle's vertices and the extrusion vector, which is the same for all of them.

Walls

At the beginning of the Cutting section we postponed the problem of a navmesh intersecting walls until after we set up the method of handling vertical clearance. At the same time, we avoided extrusion of walls into wedges because of aligned normals.

The solution to both these problems is instead of treating wall triangles as a thin object, we should make them thick by extruding them also along the direction of their normal. Intersection of such shapes could then cut a sufficiently wide gap in the navmesh that a navmesh triangle could be split apart and not be reconnected later.

Alternatively, if our intersection algorithm can safely handle wedges of zero volume, we can delay the extrusion along normal until after we intersect the navmesh triangle and the wedge. The result of this intersection would then be a line segment, which we can then convert into a rectangle. This is the approach that we took in our implementation.

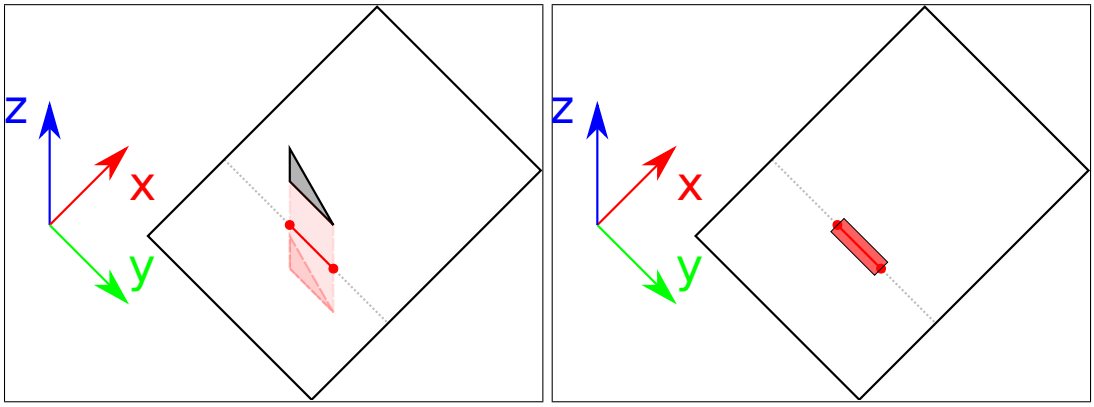


Figure 2.12: Extruding wall intersection in the plane

2.4.2 Implementation

In this part we go into more detail about the actual implementation of cutting. First we describe the initialization step that happens once for the whole mesh, then we describe the cutting operation that happens on each triangle individually.

Initialization

BVH construction To avoid having to intersect every navmesh triangle against every wedge it is beneficial to use some kind of a bounding volume hierarchy and reduce the set of possibly intersecting wedges.

We will use an R-Tree, which is a data-structure for spatial searching proposed by Guttman [1984]. There exists an open-source implementation maintained by Barkan et al. [2020]. In this tree we will store a hierarchy of bounding boxes. The bounding box of every triangle will be computed from a component-wise minimum and maximum of its vertex positions.

BVH query To get a set of potentially intersecting wedges we will perform a RTree search query in a cuboid located within and above every navmesh triangle. Additionally, we further increase the search area by a *wall extrusion distance* in order to include nearby walls. Such walls would create intersections when later extruded as described in Walls paragraph.

Per triangle operations Now we describe the cutting operation done on each navmesh triangle.

Establishing 2D space Any triangle lies within a single plane and in that plane can be defined using 2D coordinates. We can find a suitable basis vectors of this plane's coordinate system from the triangle's normal $\vec{n_i}$ and its longest side vector $\vec{s_i}$:

$$\vec{x} = \frac{\vec{s_i}}{\|\vec{s_i}\|}$$
$$\vec{y} = \frac{\vec{n_i} \times \vec{x}}{\|\vec{n_i} \times \vec{x}\|}$$

This will produce two basis vectors $\vec{x}$, $\vec{y}$ that define a coordinate system of the plane.

We will establish such coordinate system for all the triangles, and the remaining cutting operation will happen in 2D. After finishing with all cutting operations, we will convert from the plane coordinate system back into the 3D coordinates of the scene.

Intersecting wedges We described how to get a list of wedges that possibly intersect our triangle in BVH query paragraph. We then find an intersection of each wedge against the triangle's plane. The result will either be a convex polygon or a singularity in the form of a point or a line. We will ignore such singularities because they do not represent a meaningful detail. Also, we will offset every wedge downwards by some small ϵ, so that a navmesh triangle is not intersecting its own wedge.

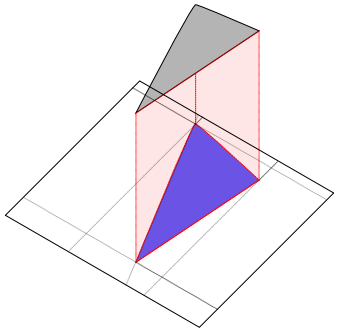


Figure 2.13: Wedge - plane intersection produces a polygon (blue)

We calculate the wedge-plane intersection by breaking down the wedge into triangles and doing 8 triangle-plane intersections instead. Then we transform their result into the plane coordinate system, remove duplicate points and compute a convex hull to find the final polygon. Möller [1997] describes a fast triangle-triangle intersection tests that can be generalized to compute a triangle-plane intersection.

Intersections of wall triangle wedges are being handled as we described in figure 2.12.

Polygonal operations As we described, a wedge-plane intersection results in a convex polygon within the 2D plane. We will grow this polygon in size by a small constant to prevent floating point issues and small gaps from creating long thin slices during subtraction.

The polygon represents an area within the plane that is being blocked by a triangle above. By subtracting the polygon from the original triangle we are able to compute an area that is still walkable. We will do such subtraction for each wedge, until we are left with the final walkable polygon or until the walkable area gets small enough that we can discard it.

In our implementation we are using the Boost Geometry library by Boost [2021] to do the polygon set operations.

Triangulation After completing the subtraction for all the intersecting wedges, we are left with a polygon that represents the remainder of the walkable area within the original navmesh triangle. We will triangulate the polygon using ear clipping algorithm, implemented by Earcut.hpp open-source library by Mapbox [2021].

The resulting triangulation will then be transformed back into 3D and replace the original navmesh triangle. When inserting the triangles into the navmesh we will be adding new vertices and will not connect to the surrounding topology yet. The disconnect will be fixed later in the Gap closing section. However, we can already keep the connections between the new triangles of triangulation to avoid extra work in future.

We preserve the previously computed normal of the old navmesh triangle in the new ones. Since they are within the same plane, their normal is exactly the same and calculating it again would be less precise because of their now smaller size.

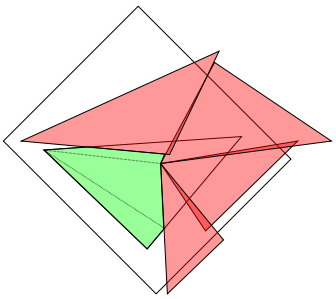


Figure 2.14: Navmesh triangle (green) after subtracting the wedges and doing triangulation.

Edge classification During cutting we can obtain some additional information about the newly created edges, that will be useful to us in the future steps. Based on how an edge was created, we can assign it to one of the three following types:

Edge types:

- **Open edge**
  This edge is a part of the original navmesh triangle's boundary and is not being vertically obstructed.

- **Blocked edge**
  This is a newly created edge inside the original triangle and its outer side is vertically obstructed.

- **Inner wall edge** This edge is a special case of the blocked edge. It is an inner edge of a triangle with its outer part being vertically obstructed, but it is specifically being obstructed by wall and is located on its inner side.

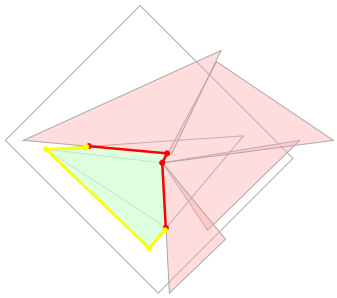


Figure 2.15: Classification of edges from figure 2.14. Red edges are *Blocked*, yellow edges are *Open*

Open edges are trivial to detect – we compare their position to all of the boundaries. To detect the *Inner wall* we have to store the shape of every inner wall outline. (We can find which of the walls is the inner one by looking at the normal vector of the wall triangle.) If the edge is aligned to any inner wall outline, we mark it as *Inner wall*. Otherwise we assume that its type is *Blocked*.

This edge type property will be stored in the mesh and we will need to preserve it during gap closing.

2.4.3 Result

Cutting removed any vertically areas from the mesh, but also introduced many new triangles. The state after cutting can be seen in the figure 2.16.

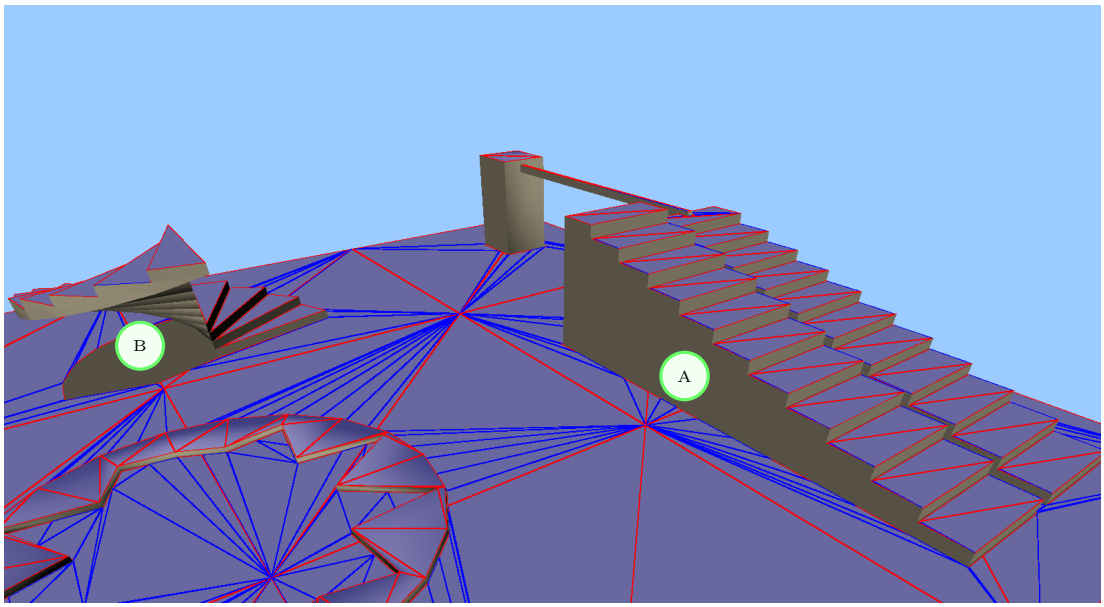


Figure 2.16: Mesh after cutting. Ground triangles stop at the outline of the wall (A). Area was cut because of insufficient space (B).

2.5 Gap closing

Until now we have kept the individual triangles disconnected from each other, because we couldn't guarantee that their topology would be a valid mesh. However, cutting and filtering should have removed all of the bigger intersections, and what remains are mostly holes, badly aligned faces and small overlaps.

In this gap closing step, we will try to connect the navmesh triangles into one or multiple meshes. Our implementation is based on Progressive Gap Closing algorithm by Borodin et al. [2002]. We customize some steps in order to be able to work directly on our half edge data structure. All the changes and their implementation will be described in this section.

2.5.1 Progressive Gap Closing by Borodin et al. [2002]

This part describes relevant parts of the Progressive Gap Closing paper that are being used in our algorithm. Our changes will be described later in the Our changes section.

The authors use two contraction operations that they use to fix the mesh:

Vertex contraction This is a generalization of an *edge contraction*, that is regularly used during mesh decimation. By allowing contractions of vertices that are not necessarily sharing an edge, it is able to close together even fully disconnected regions.

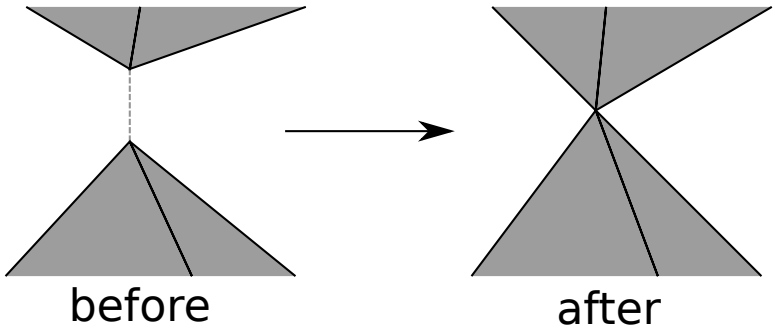


Figure 2.17: Vertex contraction

Vertex-Edge contraction Using vertex contractions alone could produce distortions in case of long narrow gaps. In such situations, it is better to connect the vertex to an edge. Vertex-edge contraction between vertex v and an edge e is defined in the following steps:

1. Project the vertex v onto an edge e
2. Split the edge e into two edges, creating a new triangle in the process
3. Perform a vertex-vertex contraction as defined in the previous paragraph

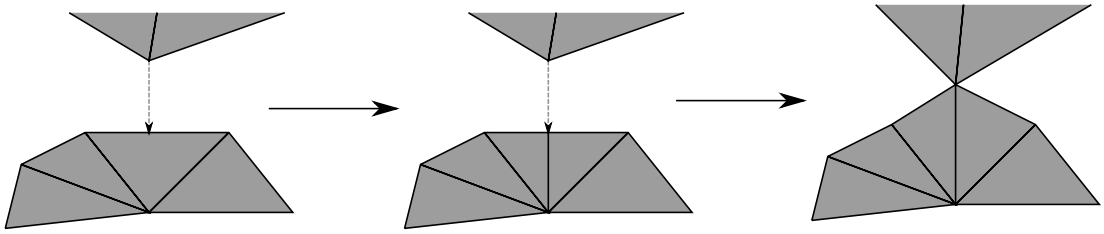


Figure 2.18: Vertex-edge contraction

Fixing the mesh The progressive gap closing algorithm does a pre-processing step, where they identify closest possible contraction pairs. Vertex-edge contractions are preferred, unless the vertex would be created too close to an existing one. The possible contractions are put into a priority queue sorted by distance between the contracted features.

In every iteration of the algorithm, they perform the contraction with minimal distance. For every affected feature they update the its planned contraction and its priority in the queue.

2.5.2 Our changes

Now we will describe the changes that we did to be able to use the Progressive Gap Closing algorithm in our work.

Limit The original algorithm is intended to provide a sequence of contractions in a 3D model and letting a user determine when to stop. In our case, we do not require such interactivity. Instead, we would like to fix all holes in the mesh while keeping relevant detail intact. We will keep doing any available contraction operations within a specific distance threshold. This threshold has to be chosen low enough so that any navmesh triangles along walls that were cut will not be merged together again.

Keeping mesh topology manifold The mesh representation used by Borodin et al. in the original algorithm is flexible enough to allow all kinds of meshes and the contraction operator might introduce non-manifold vertices and edges. Borodin et al. [2002] suggest to fix this by applying a Cutting and stitching algorithm by Gueziec et al. [2001].

In our case, the geometry after cutting should already closely resemble the topology of a mesh, and by choosing the contractions carefully, we can keep it valid. In our approach, which we will describe in the next section, we will focus on preventing any invalid contractions from happening.

2.5.3 Preventing fails during contraction operations

In order to be able to store the navmesh candidate in a halfedge data structure and to keep it a valid navmesh, we need to avoid introducing some topologies into the mesh.

Defective topologies

Authors of OpenMesh, Kobbelt et al. [2002], describes two topology cases that cannot be represented in its halfedge data structure:

Complex edge Any edge must not belong to more than two faces, otherwise it becomes a *complex edge* As every edge is made of two halfedges, it also implies an additional limitation: A halfedge cannot belong to more than one face.

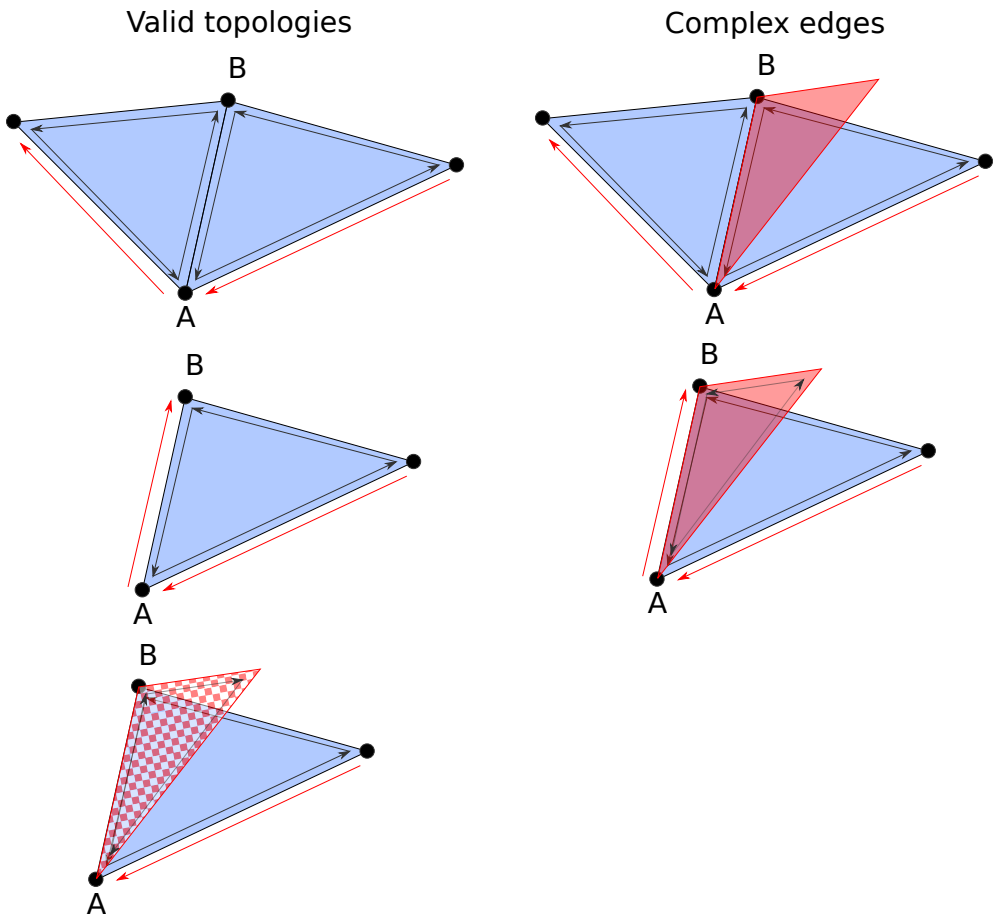


Figure 2.19: Examples of complex edges. Adding the red triangle on the right side would create a complex edge between vertices A and B. However, the red checkerboard triangle on the bottom left can be added, because its normal is pointing away from the viewer. Topologically this is equivalent to the first arrangement on the left.

Complex vertex Every neighbourhood of a vertex must be topologically equivalent to a disc, otherwise it becomes a *complex vertex*.

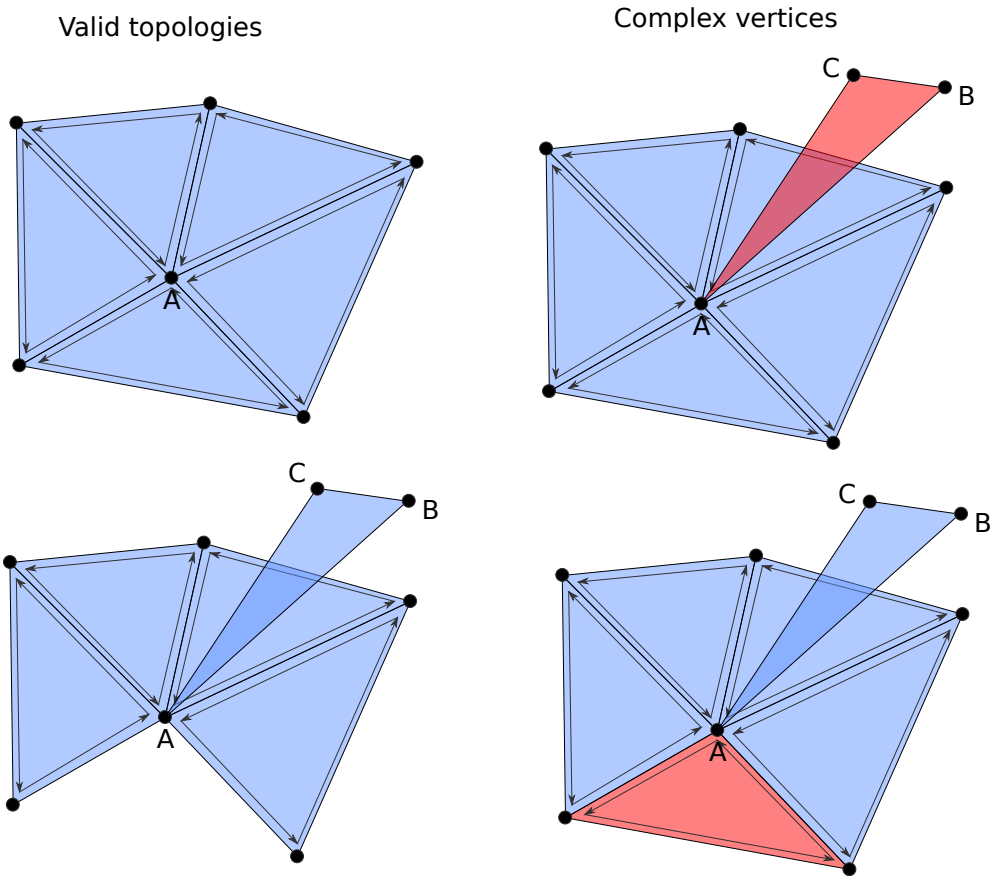


Figure 2.20: Examples of complex vertices. The addition of the red triangle would create a complex vertex. The arrangement on the bottom left is still a valid topology for halfedge DS (even though non-manifold). Only when we try to complete the disc by adding another triangle (red) we get a *complex vertex*.

Safe vertex contractions

Now we describe how to detect situations where doing a vertex-vertex contraction would create any of the previously mentioned defective topologies.

Temporary halfedge To help us in the future definition, we will first introduce a new concept. Lets define a *temporary halfedge* as a halfedge that belongs to a face that will disappear during a contraction of vertices V_a and V_b. See figure 2.21.

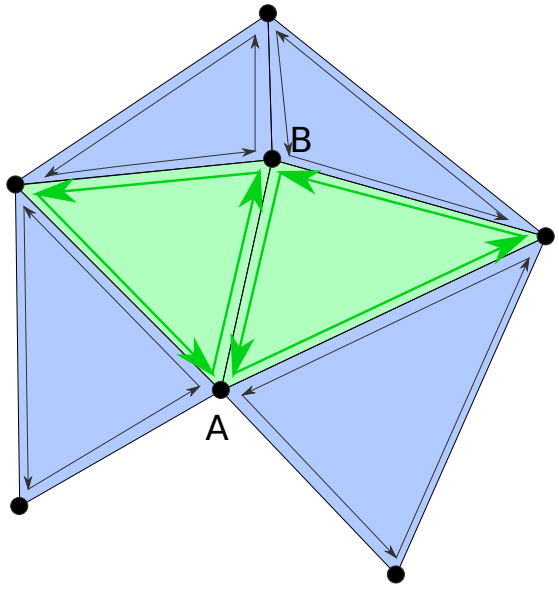


Figure 2.21: *Temporary halfedges.* Vertex contraction of vertices A and B will remove the faces highlighted in green. Their inner edges (green) will become boundaries, thus we call them *temporary halfedges*

Avoiding complex edges A vertex contraction of vertices V_a and V_b will not create a complex edge if all of the following is true:

- All neighbour vertices V_n of vertex V_a that are connected via outgoing non-boundary and non-temporary halfedge ($V_a \rightarrow V_n$) are either not also neighbours of V_b, or the halfedge from V_b to the neighbour ($V_b \rightarrow V_n$) is a boundary or temporary halfedge.
- All neighbour vertices V_n of vertex V_a that are connected via incoming non-boundary and non-temporary halfedge ($V_n \rightarrow V_a$) are either not also neighbours of V_b, or the halfedge from the neighbour V_n to vertex V_b ($V_n \rightarrow V_b$) is a boundary or temporary halfedge.
- The vertex contraction is not folding a quad over its diagonal.

These rules guarantee that any edges in the shared neighbourhood that would be duplicated after merging vertices V_a and V_b can be merged together, without any of their faces trying to share a halfedge.

Triangle fan Now we introduce another concept that will help us in the next part. Lets define a *fan around a vertex* as list of consecutively connected triangles in a counter-clockwise direction around a vertex. The fan begins with an incoming boundary halfedge and ends with the last incoming non-boundary halfedge inside the last triangle. We place no requirements on the outward facing edge of the fan – it can be connected to other triangles in the mesh.

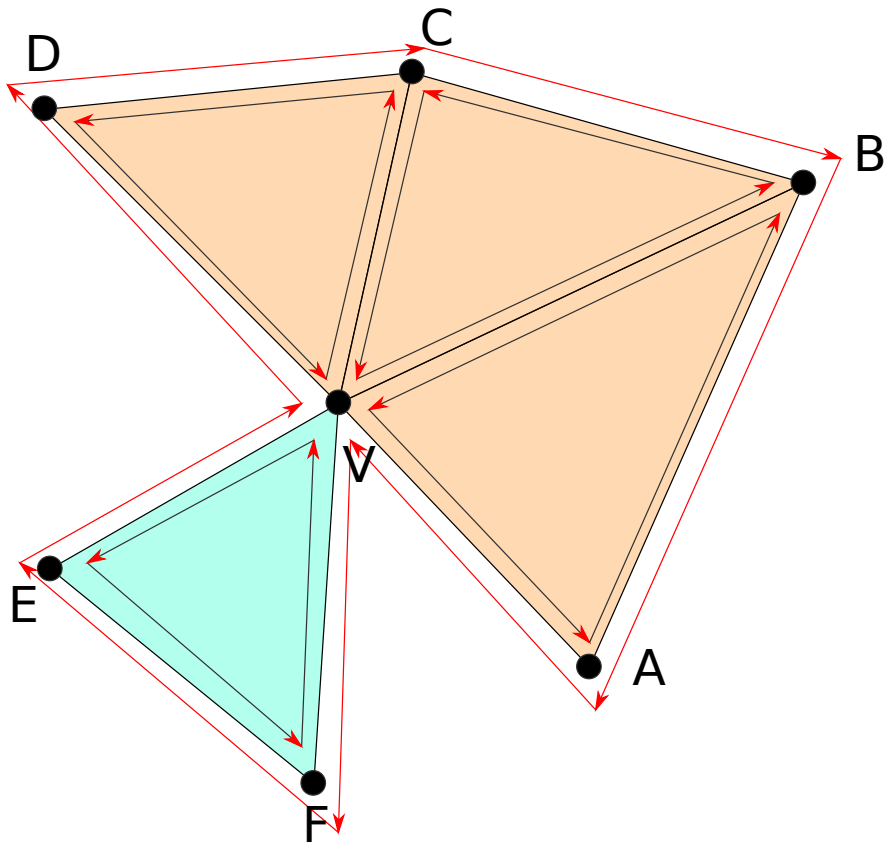


Figure 2.22: *Fan around a vertex.* The vertex V has two fans: First fan (orange) spans vertices A, B, C, D. The second fan (green) spans vertices E and F.

Avoiding complex vertices A vertex contraction of vertices V_a and V_b will not create a complex vertex if all the following is true:

- If any of the vertices V_a or V_b is not a boundary vertex, the other vertex must be isolated.
- If any of the shared neighbours V_{sn} of vertices V_a or V_b has a fan that spans from V_a to V_b, it must be the only vertex fan around V_{sn}. Note: To be precise, to create a *complex vertex*, it is enough that both of the vertices are contained in a single fan. But such case would get caught as a *complex edge* anyway, so we will not bother trying to detect it here.
- Lets now treat all the temporary halfedges as if they were boundaries. Additionally, lets construct a directed graph from all the fans around V_a and V_b in the following way: For each fan lets take the starting vertex V_f and end vertex V_t and add an edge $V_f \to V_t$ into the graph. If such graph contains a cycle, the cycle must span all the nodes in this graph.

These rules guarantee that the neighbourhood of V_a and V_b can be merged in a way that will either keep the result a non-manifold vertex, or will make it a non-boundary vertex surrounded by a topological disc.

By following these rules we can avoid creating complex vertices and edges during vertex-contraction and make it safe. Now we will make also the vertex-edge contractions safe by defining their rules.

Safe vertex-edge contractions

A vertex-edge contraction between vertex V and a boundary edge E between V_{e1} and V_{e2} is safe if the following are true:

- The vertex V and edge E are not a part of an existing face
- The vertex V is not connected to the opposite vertex of E
- If a halfedge $V_{e1} \rightarrow V$ exists, it must be a boundary halfedge. (To merge without creating a *complex edge*.) Additionally, if there is a fan around V_{e1} that begins with V_{e2} and ends with V, it must be the only fan around V_{e1}. (To merge without creating a *complex vertex*.)
- If a halfedge $V \rightarrow V_{e2}$ exists, it must be a boundary halfedge. (To merge without creating a *complex edge*.) Additionally, if there is a fan around V_{e2} that begins with V and ends with V_{e1}, it must be the only fan around V_{e2}

These rules guarantee that the vertex V can safely split an edge E and any of their shared neighbours can merge.

2.5.4 Implementation

This section will describe the implementation of gap closing in more detail. Before we describe the actual process, we first introduce some operations that we will use.

Removing small triangles

Some extremely thin triangles might be missed during cutting or some might even be created by the algorithm. This can happen when geometry degenerates too far or a set operation is not accurate enough. Such triangles can prevent us from completely closing all the holes in a mesh (figure 2.23).

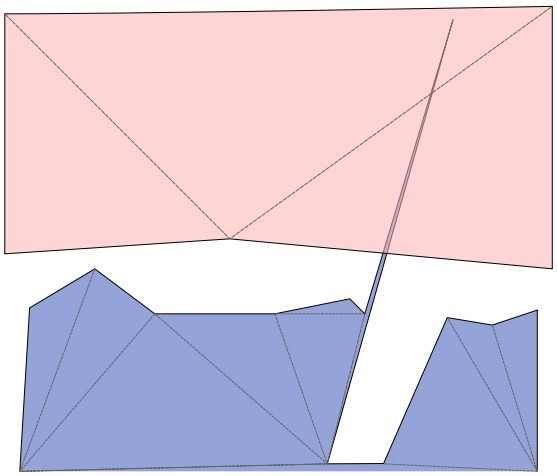


Figure 2.23: Thin triangle prevents gap closing.

As a pre-processing step we remove all triangles that have any of their altitudes below a limit. In order not to lose an edge type that might be stored in one of the triangle's edges, we have to propagate it onto the next boundary.

We will take advantage of the fact that even before running gap closing, we already have some limited connectivity in the mesh. During triangulation that we did at the end of Cutting, we connected triangles that were created from the same polygon. Now when we want to delete a small triangle, we can pass its edge type onto a neighbour. In case that the small triangle is isolated we can safely delete it – there is no meaningful edge to keep the edge type in.

If we did not preserve the edge type, we would not be able to determine safe placement for ramps and would end up with a lot of attempted ramps being cut away later.

Vertex contraction implementation

This section describes how the vertex contraction operation is implemented on the halfedge data structure of OpenMesh. Before doing any contraction, we first check that it will not produce any unwanted topologies as described in Preventing fails during contraction operations.

A contraction of vertices V_a and V_b will reconnect all the neighbours of V_a into V_b in a way that merges their topologies into one.

A simple vertex contraction is shown in figure 2.17, however the usual case is much more complicated. Each of the vertices might be surrounded by number of fans (figure 2.22), some of which might share vertices with fans of the other vertex.

Thankfully, we already verified that the contraction will be possible, using the approach described in Safe vertex contractions. We can begin by marking all the surrounding edges of V_a and V_b as modified. After the contraction is done, we will update the assigned contractions for all marked edges.

Now we will proceed with the actual vertex merge step:

Vertex merge

1. If the vertices V_a and V_b are connected via an edge, there exists either one or two faces that would become degenerate by the contraction. Such faces are safe to remove. However, if any of such faces have boundary edges, we must preserve their type.

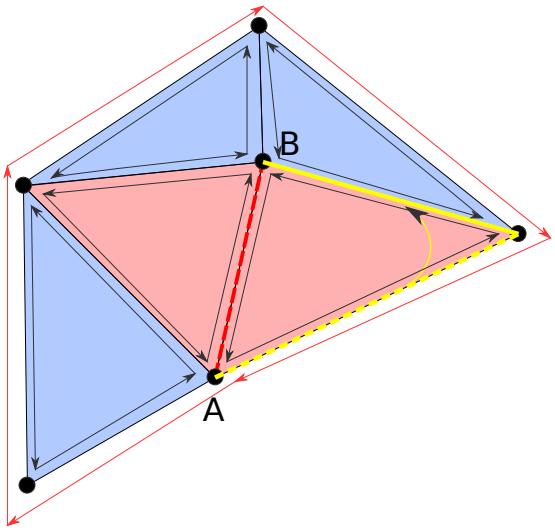


Figure 2.24: Contraction of A $\rightarrow$ B will degenerate the red triangles. The edge type of the dashed yellow edge needs to be copied into the yellow one.

2. If the vertex V_b is now isolated, we can simply move all incoming halfedges of V_a over to V_b.

3. If the vertex V_a is not isolated, we have to transfer all the fans of V_a into V_b and merge any halfedge loops that would remain. More details below.

Transferring fans We transfer every fan around vertex V_a to vertex V_b. To do that, we have to:

- Change destination vertex of every incoming halfedge of the fan from V_a to V_b.
- Reconnect the incoming boundary halfedge of V_a immediately before the fan to the halfedge after the fan. (bottom left in figure 2.25)

- Insert the boundary halfedges around the fan into the boundary loop of V_b. (Bottom right in figure 2.25.)

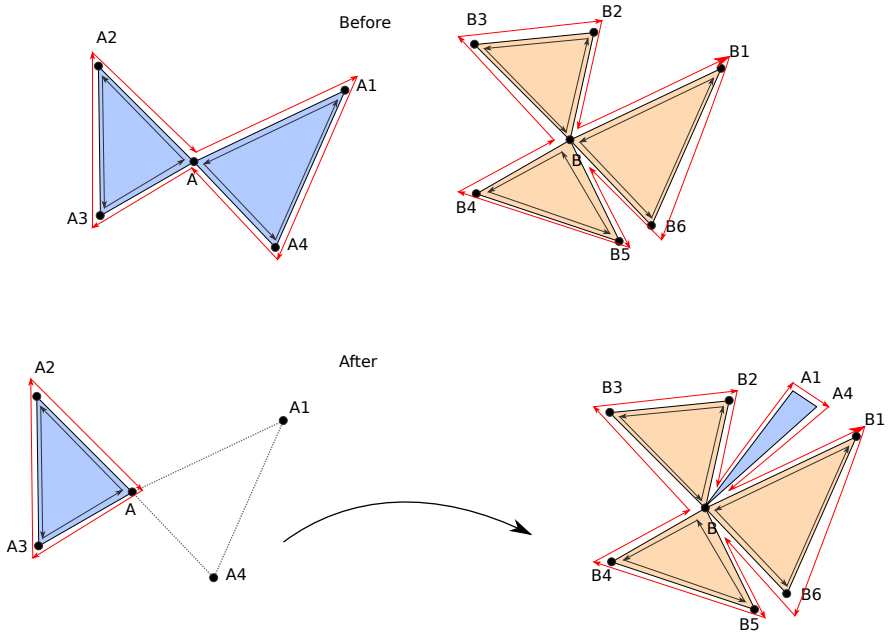


Figure 2.25: Transferring fans.

Ordering and merging fans During the transfer of fans we only preserved the boundary loop around V_b and did not check if any of the vertices are shared. There might have been a case where two or more consecutive fans could be merged into one. Such fans might not even be placed in the right order yet, so we need to order them first and then we can merge.

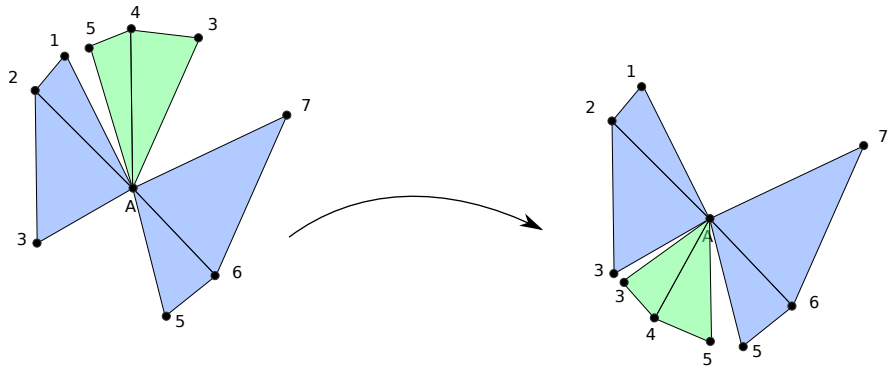


Figure 2.26: We change the order of the fans in the boundary loop such that fans that share a vertex are after each other.

Notice that in figure 2.26 there is still one more problem remaining: The vertices 3 and 5 have two edges that connect them to the centre vertex A. Essentially there is a halfedge loop of $3 \to A \to 3$ that would be a *complex edge*. We need to find all such cases and collapse the loop by making both the faces share one edge and removing the other. We also need to properly connect the incoming and outgoing halfedges of A.

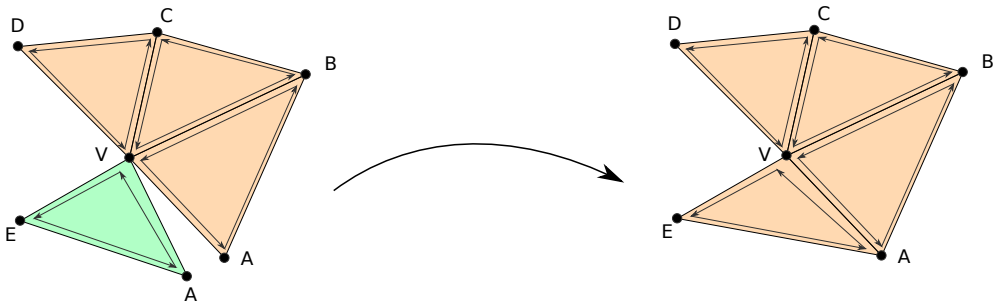


Figure 2.27: Collapsing a halfedge loop between $A \to V \to A$.

Positioning We have successfully merged the surrounding triangles of two vertices topologically, but we have left the vertex in its original position. Borodin et al. [2002] suggest to use some λ constant to blend the two positions of the vertices together. One obvious candidate could be $\lambda = \frac{1}{2}$ to essentially use an arithmetic average. However for our case this turned out problematic and we had to found a better solution.

When we add ramps during later part of the algorithm, a vertex that has been contracted multiple times could have moved far enough from its original position that a ramp originating from the edge of a stair would start to clip its geometry. This could prove disastrous during cutting where this would disconnect the ramp. Figure 2.28 illustrates such case. Additionally, blending two positions together during every contraction would also introduce increasing inaccuracies because of floating point precision.

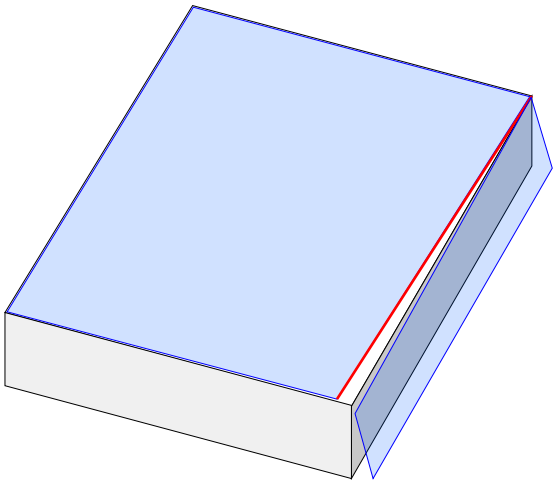


Figure 2.28: Creeping stairs issue

In order to prevent this, we always pick one of the two original positions based on a heuristic. For both positions we calculate the combined surface area of all triangles in the neighbourhood of the vertex, and pick the position with larger surface area. This heuristic will favour vertex positions that do not needlessly reduce the area of navmesh.

Vertex-edge contraction implementation

In this section we describe the implementation of vertex-edge contraction on the halfedge data structure of OpenMesh. Before we do the contraction, we make sure that it will result in a valid topology using the process described in Safe vertex-edge contractions.

A vertex-edge contraction will connect a vertex V to an edge E by splitting the edge into two edges E_1 and E_2 that will share a common vertex V. A simple vertex-edge contraction is shown in figure 2.18, but in the usual case, the topology around vertex is more complicated. The vertex V might be surrounded by a number of triangle fans, any of which might share vertices with the edge E.

In the Vertex contraction implementation section we described how to approach merging of two vertices, and the same contraction will be the basis of contracting a vertex with an edge. We can split the vertex-edge contraction in following parts:

- Splitting an edge E by an temporary vertex V_t. V_t will be a new isolated vertex that we create just for this step. The split is a relatively simple operation that creates one additional face and two new edges. Because the vertex is isolated, we don't need to merge any neighbourhoods, just connect the edges. (Figure 2.29)

- Merging a vertex V_t into V, the same way as we described in Vertex merge paragraph in the previous section. We can treat the vertex V_t like a vertex with one fan made of two triangles (surrounded by the two boundary edges E_1, E_2.) In the end of the merge, the vertex V_t will be removed and the contraction will be complete.

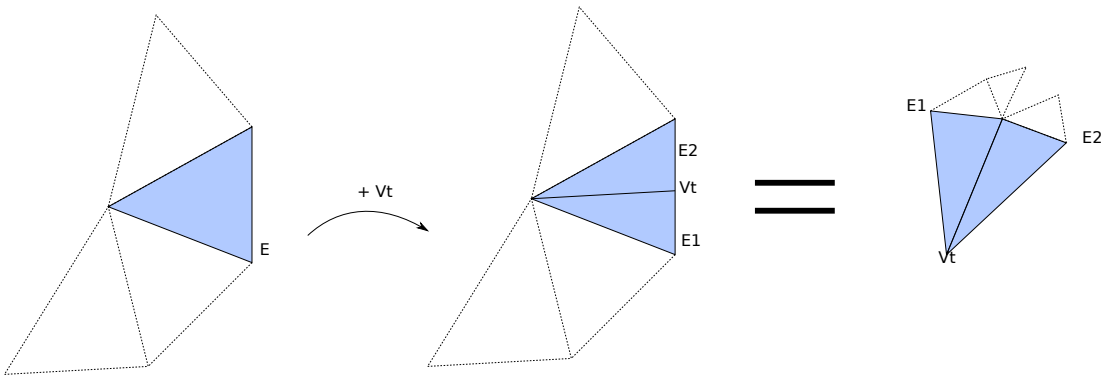


Figure 2.29: Edge split by isolated vertex converts vertex-edge contraction to vertex-vertex contraction.

Splitting an edge by an empty vertex is already implemented in the OpenMesh API. However, there are two additional things that we need to take care of. As before, we must pass the edge type property from the edge E onto the new edge. Additionally we must also mark them both as modified, so any contractions of E will now pick the more appropriate of E_1 or E_2.

Algorithm

In the previous parts we described the necessary theory and building blocks that we will now use to describe the algorithm.

The algorithm can be broken down in the initialization step and a series of gap closing iterations. The initialization cleans up the mesh and sets up the data structures. Every iteration applies a sequence of contractions on the mesh.

During iteration we may encounter contractions that would result in invalid topology, as described in Safe vertex contractions. We will place such contractions in a *ban list* and will avoid them for the rest of this iteration.

It is possible that a sequence of contractions does a change to the topology in a way that will allow one of the previously banned contractions to continue. Before we start the next iteration we will unban all previously banned contractions to allow them to potentially happen. We will keep iterating until there are no more contractions being done.

Finally, we also have to update the stored normal vector of faces that were changed.

Initialization

1. Remove small triangles from the mesh, as described in Removing small triangles section.

2. Construct an R-Tree (Guttman [1984]) that stores boundary edges of the mesh. We will later update this R-Tree dynamically with new and modified edges.

Iteration For the iteration step we will require a priority queue with support for deletion. In this queue we will store the shortest possible contraction for each vertex in the form of a vertex-edge pair. All the contractions globally will be ordered by their distance.

1. For every boundary vertex in the mesh we find its closest edge and insert the contraction into the priority queue.

2. While there are valid contractions in the queue we:

   (a) Attempt to do the shortest contraction, checking that it is valid using the approach we described previously. Invalid contractions will get added to the *ban list* and we will find a replacement contraction for the mentioned vertex. The newly found contraction gets added back into the queue.

   (b) If the contraction succeeds we will have some edges marked as modified. These might be both edges the position of which has changed, or edges that are no longer a boundary. For all such marked edges we need to invalidate all their corresponding contractions and find a replacement contraction for their vertices.

2.5.5 Result

In the end of this step we will have a mesh where all the internal gaps have been connected and small defects fixed.

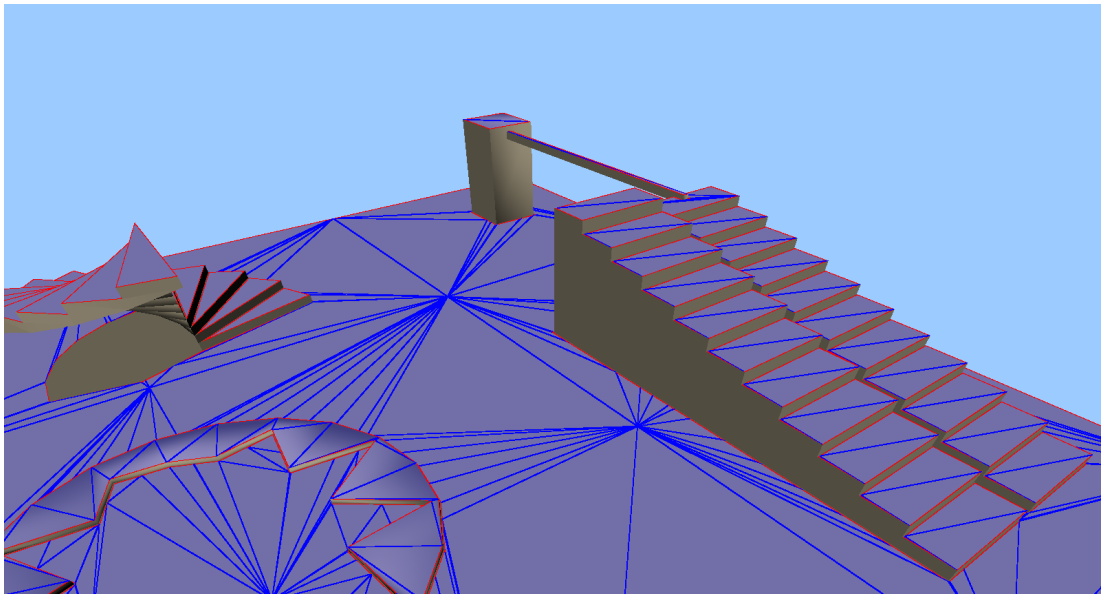


Figure 2.30: Mesh after gap closing

2.6 Unnecessary vertex removal

After gap closing we may a large amount of geometry that increases the mesh complexity without adding any detail. Examples could be internal vertices of flat faces or consecutive vertices on the boundaries. In this step, we will try to remove all such unnecessary features without sacrificing accuracy in the next parts of the algorithm

2.6.1 Decimation

The process of removing features from a mesh is called decimation. OpenMesh library provides a mesh decimation framework that works by applying a series of *edge collapses* to the mesh based on a priority metric. The choice of a priority metric has a great influence on the outcome of the decimation. Apart from setting the order in which edges are collapsed, it also allows us to prevent some edges from being collapsed at all. By customizing the metric, we can focus the decimation only on the redundant parts of the geometry.

In the context of OpenMesh, we have the option of customizing the decimator and its metric by a set of *Decimating Modules*. Every decimating module has the option to prevent an edge collapse from happening and can potentially do some extra work before the collapse has happened. One of the decimating modules can be set to control the overall order of collapses by assigning each of them a priority value.

In the rest of the section we will describe a set of modules that we used to achieve the desired decimation behaviour for this step.

Modules

Normal deviation module This is a module implemented by OpenMesh. Its purpose is to track a change in a size of a cone of normal vectors of surrounding faces. An edge collapse can only happen if it does not increase the size of the cone over a set threshold.

By providing a low threshold of 1°, we allow only collapses within a relatively flat neighbourhood. We also use the magnitude of change of the cone size as a decimation priority, but the effect is likely negligible.

Normal flip module This is another module implemented by OpenMesh, that prevents faces from ever being flipped by an edge collapse. However, in our testing this has not always proved sufficient and we had to extend this by adding a custom module. Our custom module guarantees that all face normals in the mesh will always point upwards.

Boundary preserving module This module prevents edge collapses from changing the boundary of the mesh. Its operation is based on the following rules:

- Collapses from a boundary vertex inwards (into non-boundary one) are prohibited. Collapsing inner vertices outwards is allowed.

- Collapses from a boundary vertex to another boundary vertex are only allowed if they share a boundary edge. This prevents a collapse of two different boundary lines.
- Collapses along a boundary edge can happen only if the vertices are collinear and only in a way that will not move the edge. (See figure 2.31)

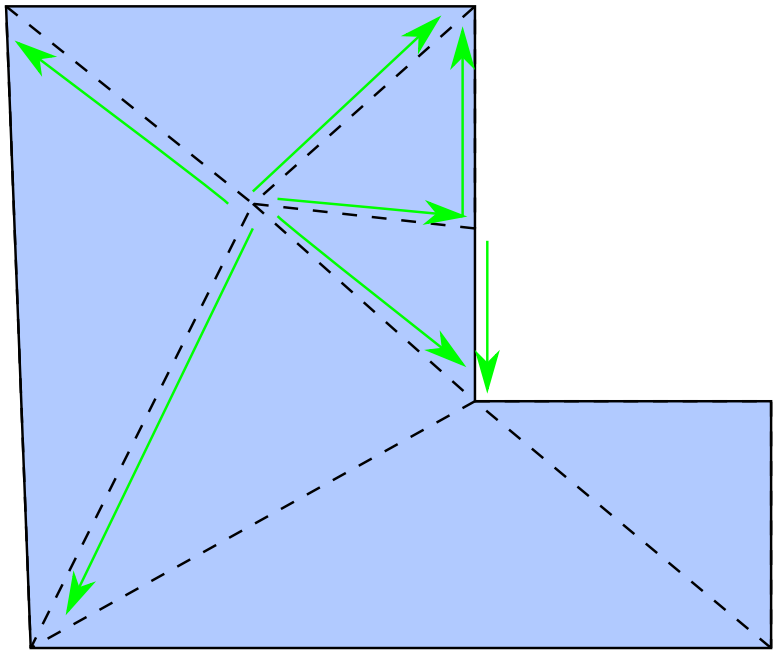


Figure 2.31: Contractions allowed by the boundary preserving module are shown in green.

Edge type preserving module This module blocks contractions of collinear boundary points on a line with two different edge types. Additionally, it preserves the edge type of boundaries during inwards contractions, similarly to how was done during gap closing. (Figure 2.32)

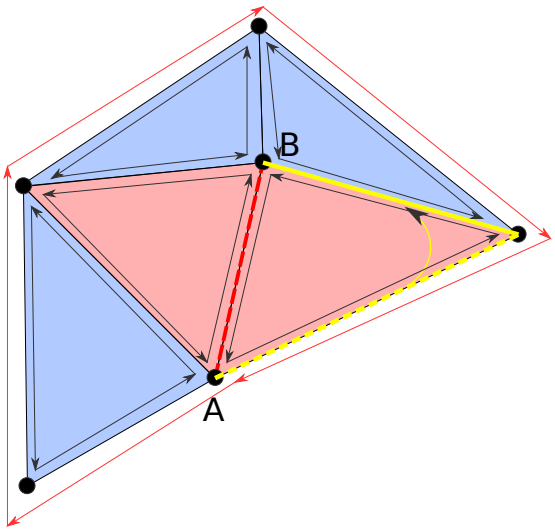


Figure 2.32: Collapse of A $\rightarrow$ B will remove the red triangles. The edge type preserving module will copy the edge type of the dashed yellow edge inwards, so it is not lost.

2.6.2 Result

The combination of the previously mentioned decimation modules results in removal of some of the unnecessary vertices. Our goal in this step is to stay conservative with our removal. We still need the features of the geometry to stay precise.

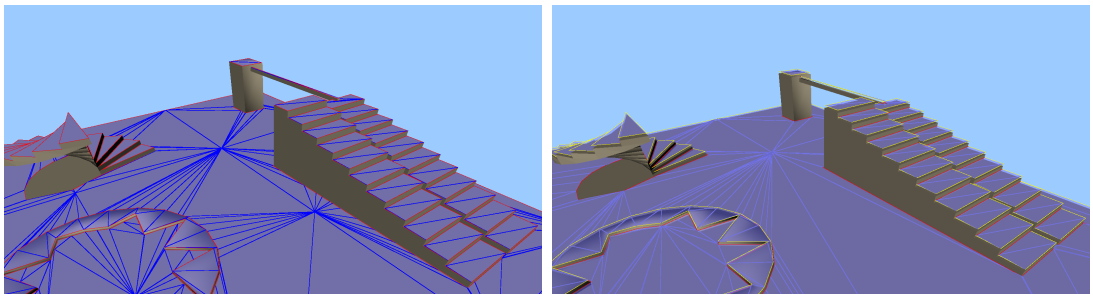


Figure 2.33: Mesh before and after removing unnecessary vertices.

2.7 Adding ramps

Throughout this section we describe the process of connecting different parts of the navmesh via ramps.

In the Cutting section, we were able to distinguish several types of edges, that are described in Edge classification paragraph. The most important ones to us are the *open edges*, these are the ones that are not being blocked by any geometry, be it walls or obstacles above.

From these open edges, we will place ramp quad that will then connect to the rest of the navmesh below. We will keep only the parts of the ramp quad that are supported by existing navmesh. If there are any holes under the ramp, then a matching part of the ramp will be missing.

After we are done, the ramps will become a regular triangles of the navmesh and will undergo the same cutting step against input geometry as the other parts of navmesh did before.

Ramp angle Ideal ramp placement is not an easy task. If we try to place a ramp too close, we will end up with unnecessary sharp turns in the geometry. If we try to put the ramps too far, we might not be able to connect to the navmesh below. Even the shape of the bottom part of a navmesh will influence the ideal placement of a ramp. Figures 2.34 and 2.35 illustrate two problematic cases with different ramp placement.

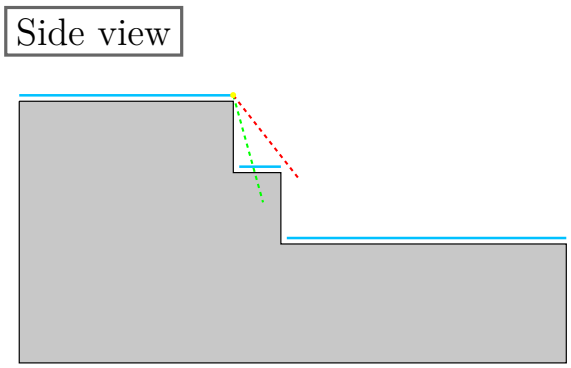


Figure 2.34: Ramps with shallow angle (red dashed) might miss navmesh below.

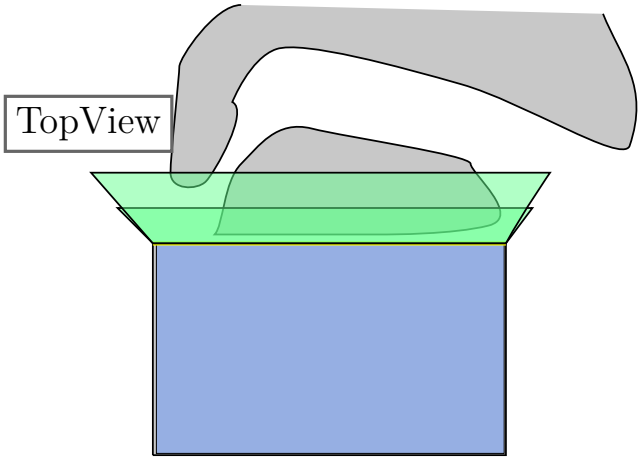


Figure 2.35: Ideal ramp placement would require considering the rest of the topology. The shorter ramp would have a longer shared edge, but it would not connect to the farther part of navmesh

To err on the side of caution, we decided to place the ramps endpoints as close as possible (but respecting the max steepness limit) in order to never miss the navmesh on even short stairs. We will try to fix the sharp turns later during Ramp flattening and simplification. This ramp placement produced the best results for us, but we recognize that a better solution might exist.

2.7.1 Preparation

Before we begin placing ramps, we will do some pre-processing on the mesh. We introduce two additional properties that we will store in the faces of the mesh:

- *Plateau id* This property is used to identify triangles that belong to the same flat area on top of a stair. Multiple ramp start points (open edges) might belong to the same plateau. We will later use this property to avoid cutting parts of navmesh by their own ramps
- *Ramp id* This property identifies triangles of the same ramp.

And one property that we will store on the open boundary edges:

- *Aligned normal* Vector orthogonal to the edge, that is also aligned to the adjacent face. We calculate it as a cross product of the edge direction and the normal of the inner triangle. (Figure 2.36)

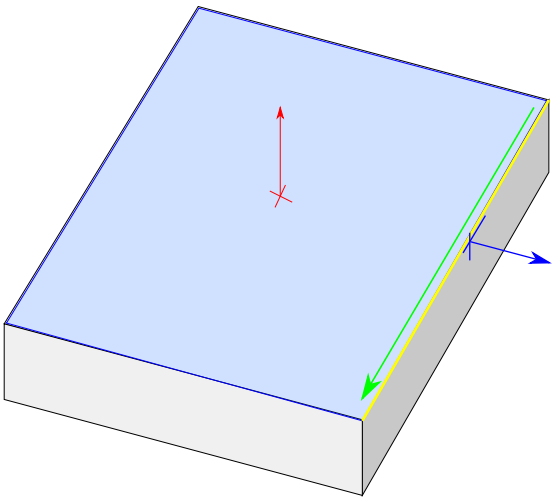


Figure 2.36: Aligned normal (blue)

Initializing plateau id We iterate over every open boundary halfedge, and check its inner face. If the face has no *plateau id* yet, we assign it a new one, and also assign the same *plateau id* recursively to all its neighbouring with similar normal vector.

Preparing edges We filter away any open edges that are too short for a ramp, or where we were not able to compute a valid aligned normal. This can sometimes happen for very small faces which will be removed during the next simplification step.

2.7.2 Adding ramp quads

Once we have all the properties ready, we can continue by adding the actual ramp. The ramp starts as a quad which we will later shape based on its intersection with the rest of the navmesh.

Common aligned normal We would like ramps coming from consecutive edges to share their side edges so we can close any holes. To guarantee that, both the start and end points of neighbouring ramps must be the same.

Let us introduce a concept of a *common aligned normal* that we define for a vertex between two open edges. The *common aligned normal* is set to the average direction of the aligned normals of the two neighbouring edges, scaled by a factor used to compensate for a change in the angle over the vertex.

Let n_1 and n_2 be *aligned normals* of two consecutive halfedges and α an angle between them. We define their *common aligned normal* n_c as follows:

$$n_c = \frac{n_1 + n_2}{\|n_1 + n_2\|} \cdot \frac{1}{\cos\frac{\alpha}{2}}$$

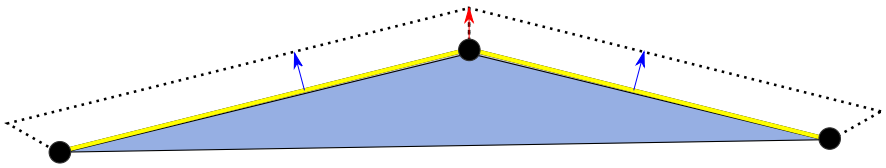


Figure 2.37: Common aligned normal (red). Future ramps are dotted.

Scaling of the *common aligned normal* is important to guarantee that the four points of a ramp will remain a valid quad with the same ramp angle at all points along the edge.

However, if we do not limit the scaling, the ramp endpoints could be placed too far during sharp turns. (Figure 2.38)

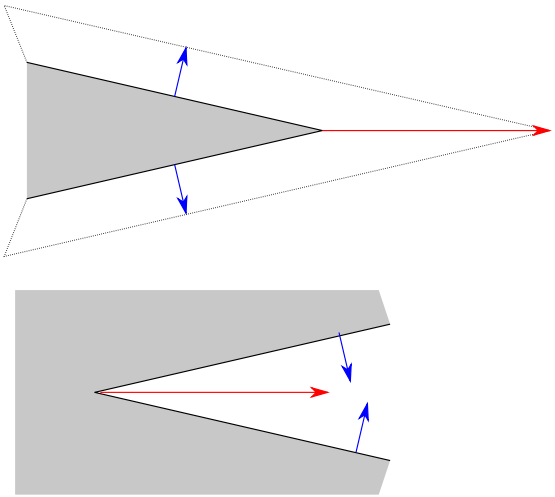


Figure 2.38: Common aligned normal (red) without limit.

We will set a limit on the maximum distance the *common aligned normal* can stretch from the ramp. In cases where it is either too far away for convex turns, or past midpoint for concave turns we will use the regular aligned normal and keep the ramps edge disjoint.

Ramp start points We will place the ramp start points in the vertices of the edge, offset slightly in the direction of the common aligned normal. This offset will reduce the chance of us accidental clipping part of the ramp by the underlying geometry during cutting.

After we are done placing the ramp quad and successfully intersect it with other navmesh geometry, we will set the positions of the original vertices of the ramp edge to the offset ramp start points.

Ramp end points Our plan is to create as steep ramps as possible and use the maximum possible height. We are not worried about a ramps clipping existing navmesh, because we will be cutting them later.

We have two variables exposed in the Algorithm settings, one controls the maximum slope of a ramp, the other controls the maximum height of a single step. From these two, we can find the ramp distance d in the horizontal direction. Figure 2.39 illustrates their relationship.

$$d = \frac{\text{max step height}}{\tan(\text{max stair slope})}$$

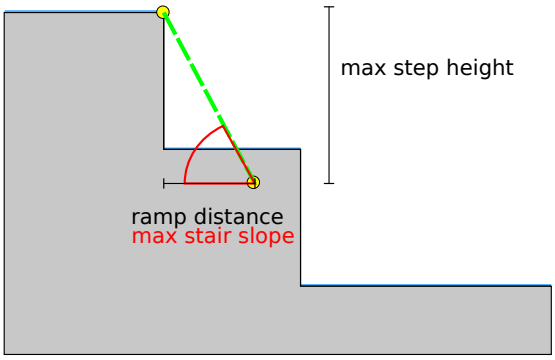


Figure 2.39: Ramp parameters.

Let h be the max step height from Algorithm settings, $\vec{up}$ the world up vector and d the ramp distance calculated in the previous step. For a ramp start vertex at position r_s with *common aligned normal* $\vec{n_s}$ we calculate the position of the ramp end point r_e in the following way:

$$r_e = r_s + d \cdot \vec{n_s} - h \cdot \vec{up}$$

Intersecting the ramp quad By defining two start and two end positions of the ramp, we have created a quad that lies within a single plane. In the Cutting section we described how we cut away parts of the navmesh by intersecting wedges created by the world geometry.

Now we will use the same approach, this time cutting away parts of the ramp quad by wedges created from the **navmesh geometry**. The goal is to shorten the ramp so it does not go through the bottom part of the navmesh.

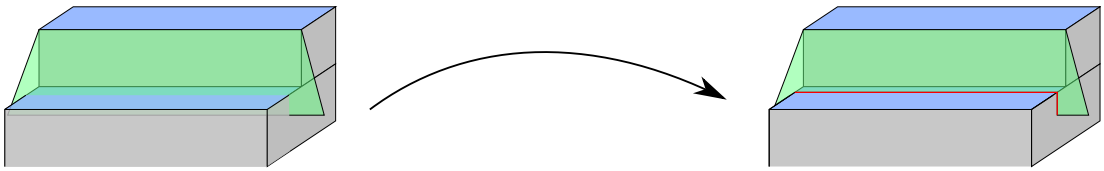


Figure 2.40: Ramp quad cutting by navmesh

However, before we triangulate the result of the cutting step, we will do some additional processing that will throw away parts of the ramp that are not being connected the rest of the navmesh.

Finding supported sections If we take a better look at the shape of a ramp polygon after cutting, we can notice that it has a characteristic shape. Every flat section of a navmesh that we cut by has left a sort of an island surrounded by almost vertical edges.

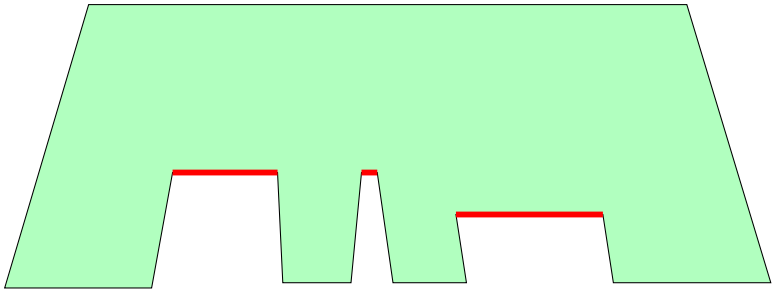


Figure 2.41: Cuts from navmesh have created plateaus (red) surrounded by close-to-vertical edges.

By finding such plateaus, we can identify the bottom parts of a ramp quad that are being supported by navmesh from below.

Finding supported ramp sections:

1. Begin at the bottom left vertex
2. Continue edge by edge in counter-clockwise direction, looking for the start of a supported section. An edge is a part of a supported ramp section if the following is true:
   - Is not almost vertical. (The exact threshold depends on max. allowed slope)
   - Is going in opposite direction of the ramp edge. We need to check this because there might be some imprecisions and we are trying to keep the resulting polygon *simple*.
   - Is not close to the bottom ramp quad edge.
3. Connect edges of a supported ramp section together into groups.

Note, some of the edges in a ramp polygon might be too small to reliably detect their orientation. Such edges can be skipped to keep the grouped sections connected.

Creating supported ramps Once we have identified the bottom boundaries of supported ramp sections, we can break down the original ramp quad into individual supported ramps.

For each supported ramp section we will project the left and rightmost points of the boundary to the ramp edge, to find two additional points for its polygon. We will use these new polygons instead of the original cutting result for the triangulation. In order not to create too thin ramps we can ignore any ramp sections that are too short.

Additionally, to help us better connect neighbouring ramps in the future, we can slightly nudge the projected points of the first and last supported group towards the start vertices of the ramp quad (Figure 2.42).

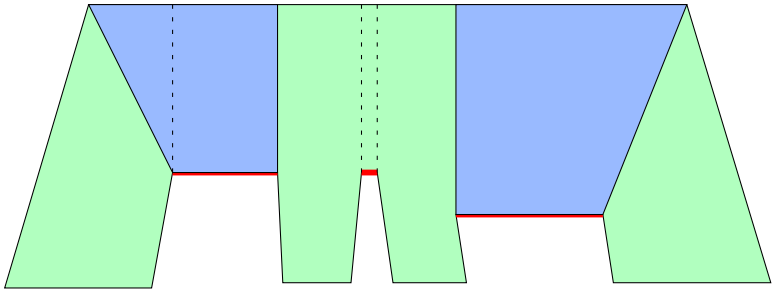


Figure 2.42: Turning supported ramp sections (red) into ramp polygons (blue).

Finally, when we triangulate this polygons into triangles and add them back to the mesh, we will also update their *ramp id* property to the ID of this ramp.

2.7.3 Cutting and gap closing

We have cut the ramp to proper size, but the bottom part of the navmesh is still going past the ramp boundary and would not always connect after gap closing.

The solution to this problem is the same as before – cutting. We could cut the triangles of navmesh by extruding the ramp triangles downward into wedges. However, there is a more efficient approach. Shape of every ramp is already similar to a wedge. And we are also interested only in clipping relatively small parts of the navmesh, there is no need to deal with vertical clearance.

Cutting by ramp wedges We will cut every triangle of the old navmesh by a horizontal ramp wedge for each ramp (Figure 2.43 shows how to create such wedges.). At the same time, we avoid cutting any triangles that represent ramps. They can be identified by the *ramp id* property.

Additionally, we avoid cutting triangles of a plateau (based on *plateau id*) by any ramps that originated from one their edges. Since these plateaus are flat, none of the triangles will be under any of these ramps. This is an additional precaution to avoid accidentally disconnecting our ramps.

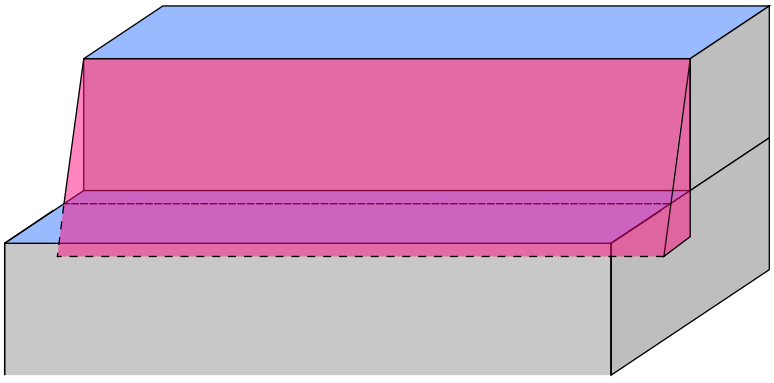


Figure 2.43: Cutting navmesh by a horizontal ramp wedge (pink).

Cutting against original geometry Until now, we have completely ignored the input geometry when placing ramps. It is possible that there could be an obstacle above a ramp that would not provide a sufficient vertical clearance to allow

passage. To fix such cases, we will run another cutting step. This time only cutting triangles of the ramps by wedges of extruded triangles of the original geometry. This is the same step as the rest of the navmesh underwent in Cutting section. However, for any triangles we cut, we must preserve the *ramp id property*. We will later use it during ramp flattening.

Finally, we will run again the Gap closing procedure on the whole mesh to connect all the cut parts back together.

2.7.4 Results

This section showcases some parts of the navmesh after adding ramps. Red edges are boundaries and blue edges are properly connected inner edges.

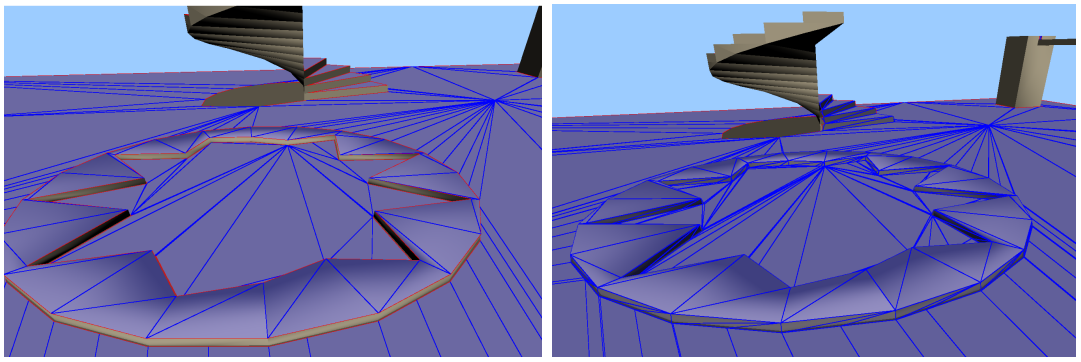


Figure 2.44: Mesh before and after adding ramps. Neighbouring ramps connect properly.

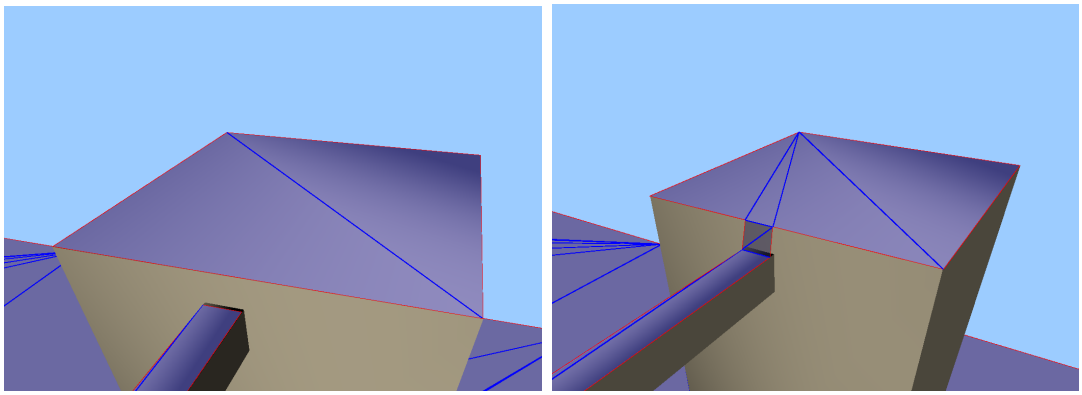


Figure 2.45: Only the supported part of a ramp is kept.

2.8 Ramp flattening and simplification

By adding ramps, we have connected previously disconnect parts of the navmesh. But the topology is yet far from ideal. Specifically, staircases would be made of multiple ramps and flat parts, which are unnecessary details for path planning. In ideal case, the whole staircase would be covered by a single ramp.

In this section we will try to flatten the staircases and simplify them together into a single ramp.

2.8.1 Ramp flattening

We will attempt to flatten ramps by doing a series of edge contractions within the faces of ramp geometry. To implement this, we will reuse the OpenMesh decimator that we described in Decimation section previously. We will use a decimation module that prevents flipping of normals (Normal flip module) and a ramp flattening module that we define below:

Ramp flattening module The module is implemented using the following rules:

- Allow only contraction of edges of ramp faces. (Inner or outer)
- Prevent contractions between boundary vertices that are not contracting a boundary edge. This prevents contractions from connecting two different boundaries together.
- Allow only contractions going upwards. This forces the same direction of contractions for all edges of a ramp and makes it more likely that it will be simplified into a single quad.

Additionally, we use an error quadrics module to set the ordering of contractions. We limit the max error to the maximum step height. This module will be better described in the following Simplification section.

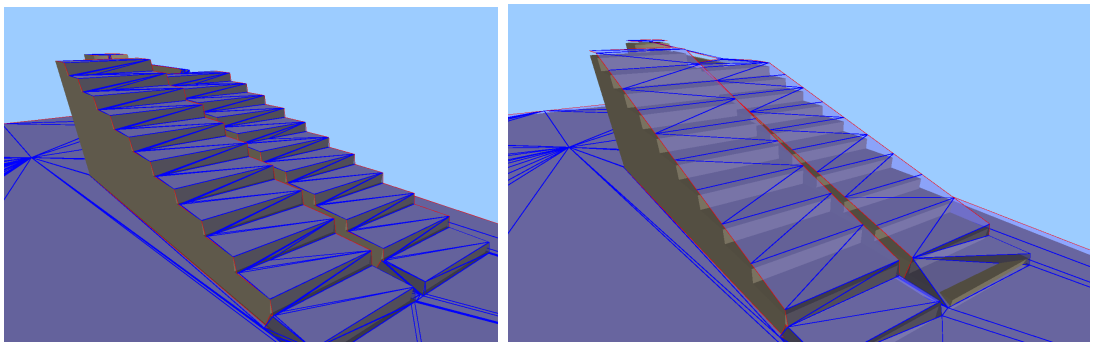


Figure 2.46: Flattening ramps

2.8.2 Simplification

After flattening, the ramps have a much better shape, but their topology is still too cluttered. In a previous section called Unnecessary vertex removal, we simplified internal geometry to remove such details. But the settings we used during that step were too conservative. Now that we have a final shape of the navmesh almost done, we don't need to preserve all the internal detail.

We will run a simplifying pass on the geometry, again using the Decimation algorithm. This time, we don't need to limit the simplification only to the ramps. We will use decimation modules to preserve boundaries (Boundary preserving module) and orientation of faces (Normal flip module).

To prioritize the best contractions, we will make use of error quadrics by Garland and Heckbert [1997]. This prioritization method is implemented by OpenMesh in the quadric module. It works by storing an accumulated error value from previous contractions in a form of quadric surface defined in each vertex. The benefit of using quadrics is that combining their error value can be easily done by summing the matrices that define the quadric.

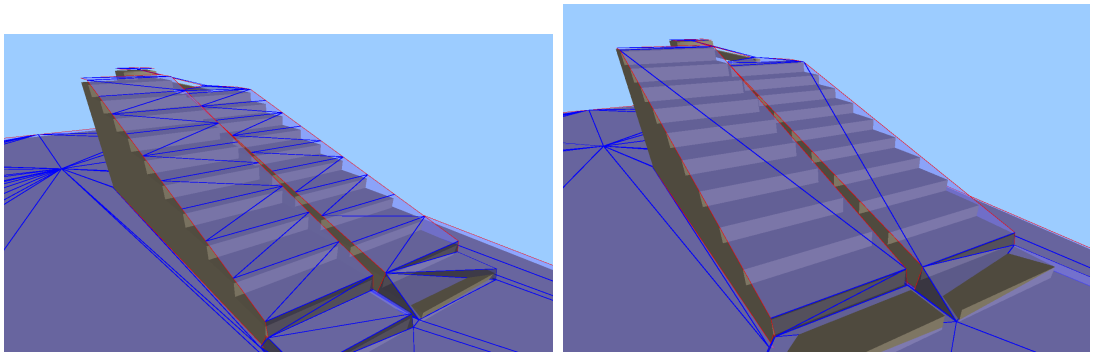


Figure 2.47: Simplification of ramps

2.9 Wall clearance

The navmesh after applying the previous simplification step of the algorithm can already be used for pathfinding. While finding a simple path for a point-agent can be done by A* algorithm Hart et al. [1968], we can also find a path for an agent of arbitrary size using an algorithm devised by Kallmann [2010].

However, when pathfinding for agents of different sizes is not necessary, it is beneficial to have boundaries of the navmesh itself already respect the size of an agent. This can be done by cutting away parts of the navmesh that are close to obstacles in order to create clearance.

2.9.1 Implementation

The boundaries of our navmesh closely follow the walls or ledges of the world geometry. Assuming that the agent can be represented by a cylinder with a fixed height and radius, we can use a 2D capsule to represent areas which are too close to a wall for our agent to walk in.

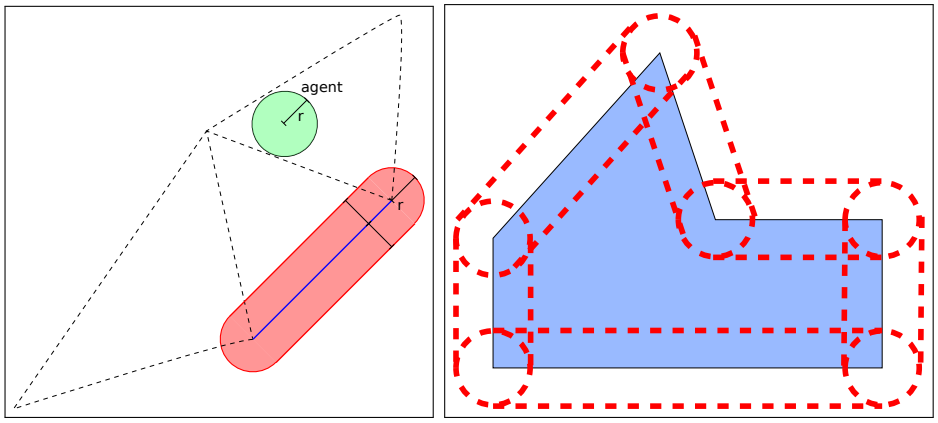


Figure 2.48: Capsule used to represent areas too close to a wall.

In 3D, the situation is slightly more complicated. Because there is no guarantee that the navmesh can map to a 2D surface, we cannot use just a 2D capsule to work in the navmesh space.

We will instead convert these capsules into polyhedrons, and use a similar approach as we did in Cutting to remove blocked areas.

Shape of the extruded polyhedron The shape of the polyhedron is based on the position of the edge and actor radius. However, we cannot use just the direction vector of the edge as one dimension of the capsule – the slope of the edge would skew the radius, and if we looked at the capsule from top-down view, it would not match the desired 2D shape. Figure 2.49 illustrates such problem.

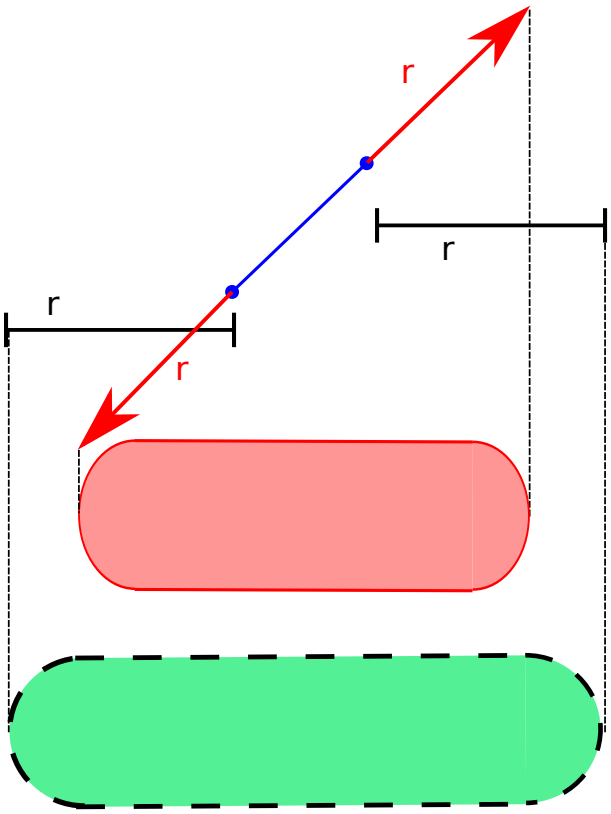


Figure 2.49: Side view of a ”capsule” shape in 3D.

To avoid this, we will use just the horizontal component as the direction for one side. Additionally, we will extrude this shape vertically by the *maximum step height* from the Algorithm settings. The resulting polyhedron of a blocked area is shown in the figure 2.50. To simplify cutting, we will use the convex hull of this polyhedron to make the intersections easier.

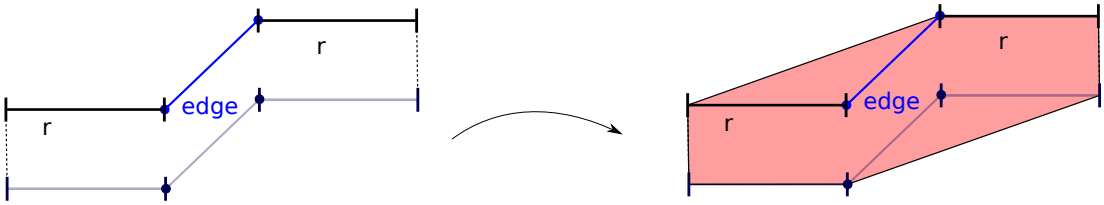


Figure 2.50: Side view of blocked area around edge (left), and the polyhedron used for cutting (right).

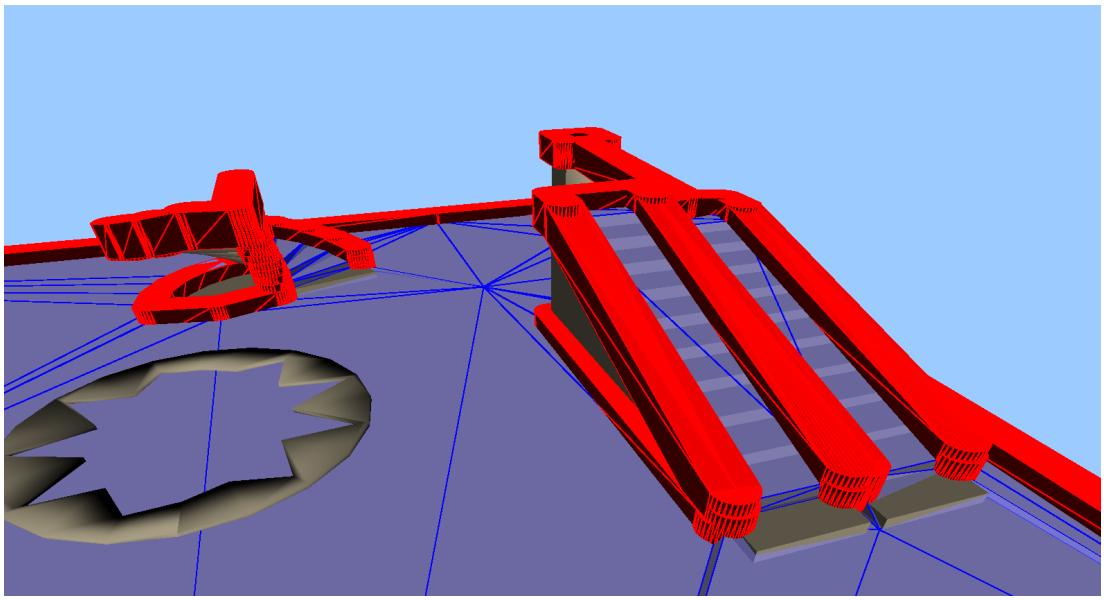


Figure 2.51: Polyhedrons representing the wall clearance shown in the navmesh.

Cutting Once we have created the polyhedron, we will proceed by cutting the navmesh triangles like in Cutting section. The major difference is that instead of intersecting planes against a wedge shape, we now have to intersect against the shape of the polyhedron.

Polyhedron-plane intersection To reuse as much as possible from previous work and to take advantage of the fact that our polyhedron is convex (figure 2.50), we will break down the intersection into two steps:

1. Calculate intersections of all triangles that make the sides of our polyhedron against the plane. We will store the results as a collection of points.

2. Calculate a convex hull of these points in the plane. The result is a polygon that represents an intersection of the polyhedron and the plane.

With the intersection polygons calculated, we will proceed by subtracting them from the navmesh triangles. Once we are done, we triangulate the remainder. These steps are the same as in the Cutting part of our algorithm.

Finalization In the end, we run our Gap closing algorithm again to connect the remaining navmesh together. The cutting may have detached parts of the mesh away, some of which are now too small to be useful for pathfinding.

We can calculate the combined area of the navigable surface of each part by summing the surface areas of their triangles together. Then we remove any such parts, where the combined area is below a threshold.

2.9.2 Results

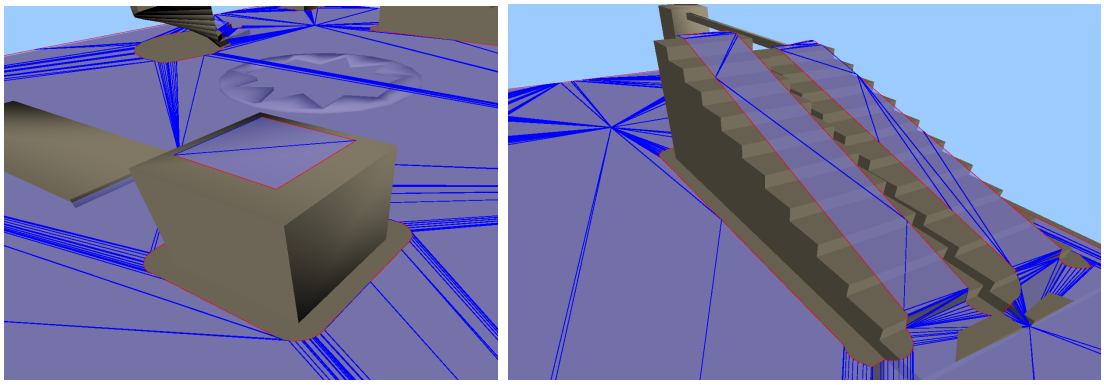


Figure 2.52: Navmesh with wall clearance 1

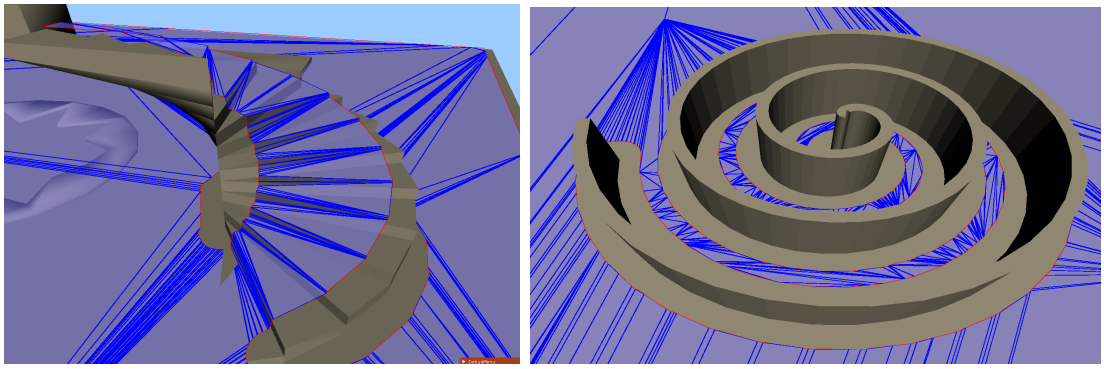


Figure 2.53: Navmesh with wall clearance 2

3. Evaluation of results

In this chapter we will evaluate the results of our algorithm and compare it to an algorithm that uses voxelization during its work.

3.1 Results of our algorithm

Our goal was to create an algorithm that is able to preserve precision and detail of the input geometry. This is necessary in order to guarantee that parts of the geometry that are connected via thin pathways will also remain connected in the navmesh. Failure to do so would make any pathfinding algorithm unable to find some paths.

The algorithm that we introduced in this thesis does not do a voxelization step to simplify its input, but instead works on the raw geometry directly. This allows it to keep the required precision and avoid discarding important features of the mesh.

Specifically, navmeshes generated by our algorithm reliably include paths through thin doorways, low ceilings and other areas where the room for movement is limited, but are not blocked yet.

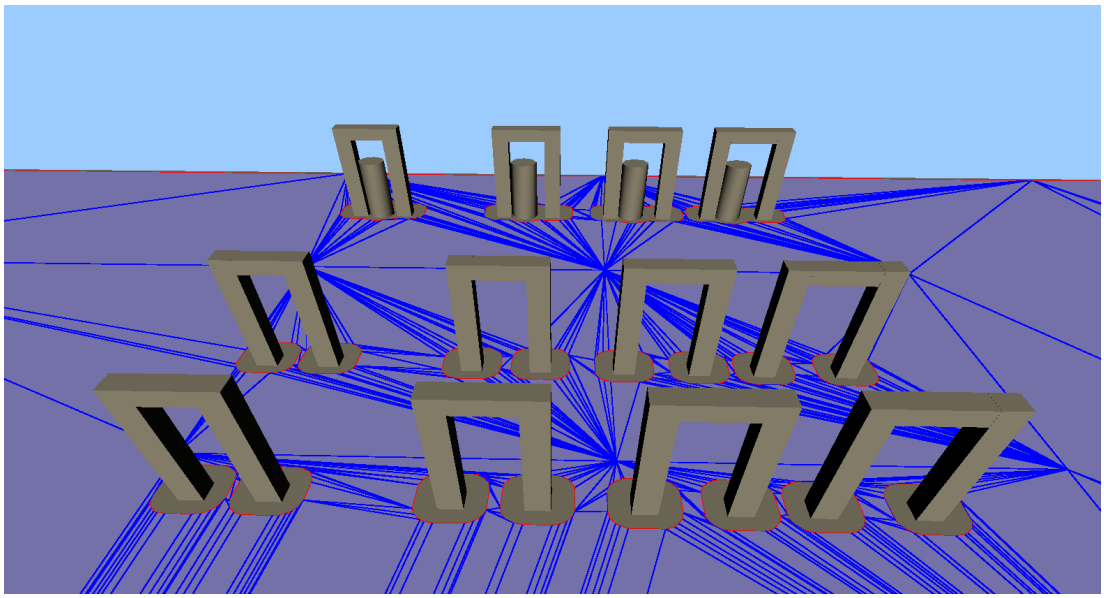


Figure 3.1: Navmesh generated by our algorithm. The input geometry includes doorways of different sizes. For reference, the agent size is being represented by the cylinder in the back.

While our main goal was to preserve important detail, we also try to simplify the navmesh geometry in cases where the detail is not required.

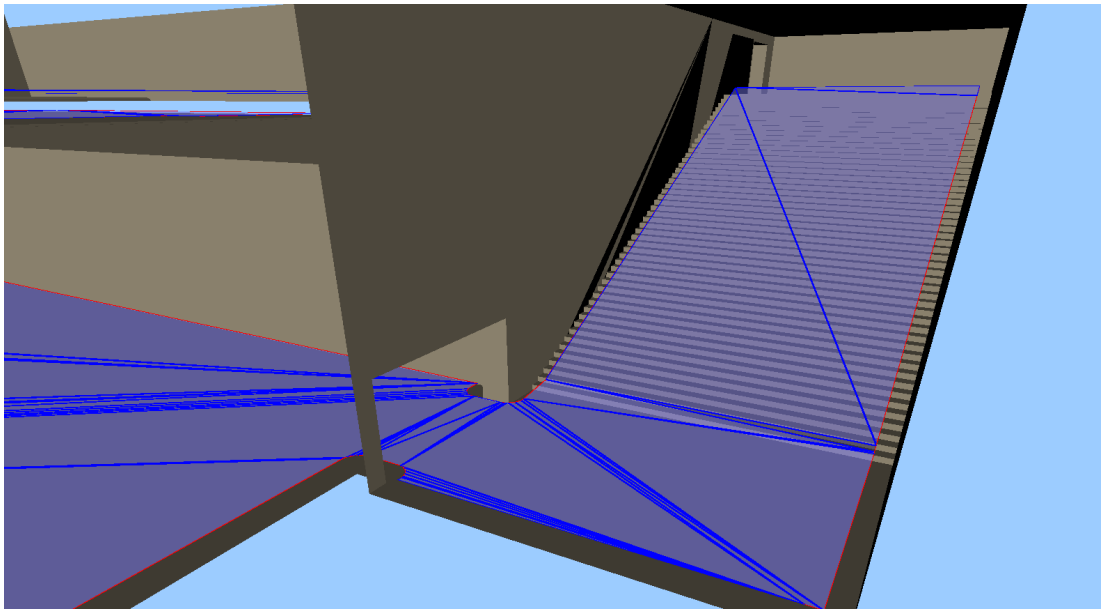


Figure 3.2: Navmesh generated by our algorithm. The floor topology was simplified and the stairs have been merged into a single ramp.

While not as as fast as some algorithms that use voxelization, our algorithm is still able to create a navmesh for a large detailed area in seconds.

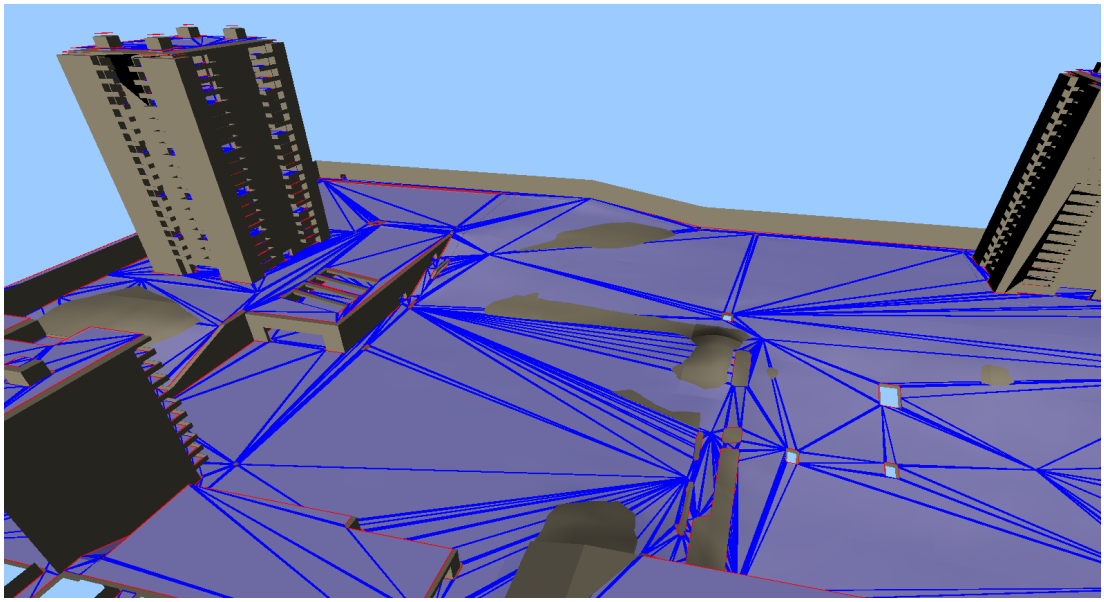


Figure 3.3: Navmesh generated by our algorithm. The level includes multiple internal areas and a large outdoor area.

3.2 Comparison with existing works

In this section, we will compare the results of our algorithm with Recast (Mononen [2009]), which is a state of the art algorithm for navmesh generation that uses voxelization.

Both algorithms will use the same settings for the agent parameters:

- Agent height: 1.8 m
- Agent radius: 0.3 m
- Max slope: 45°
- Max step height = 0.4 m

Not all of the other navmesh generation settings can be unified, as both algorithms use different techniques to generate the navmesh. To provide a fair comparison, we will therefore do two comparisons of Recast. One will be done with the default settings, the other will be done with the cell-size or other relevant setting tweaked to a level where its results can be compared to ours.

We will measure the time it takes to generate a navmesh as an average of 5 runs and the number of triangles in the mesh.

3.2.1 Detail dependent mesh

This is custom mesh designed to stress parts of the navmesh generation algorithm that require detail. The mesh will be available as *competition.obj* along the demo application of our algorithm as an attachment of this thesis.

In the following text, we will look into individual tested cases in more detail.

Case 1: Rooms and doorways

This testcase has two rooms and several doorways that constrict a path through them. The default settings of Recast are not sufficient to preserve the required detail in the navmesh. To keep all the doorways traversable we had to reduce the cell size and it had a drastic effect on the run time of the algorithm.

Algorithm	Time spent	Triangles
Recast (default)	130 ms	240, but insufficient detail
Recast (cellSize = 0.04)	15 328 ms	702
Our algorithm	2 999 ms	1 395

Although our algorithm was faster than Recast in producing the detailed mesh, our mesh also had almost twice as many triangles. This suggests that we should reduce the restrictions we place on our decimation algorithm when we simplify the mesh.

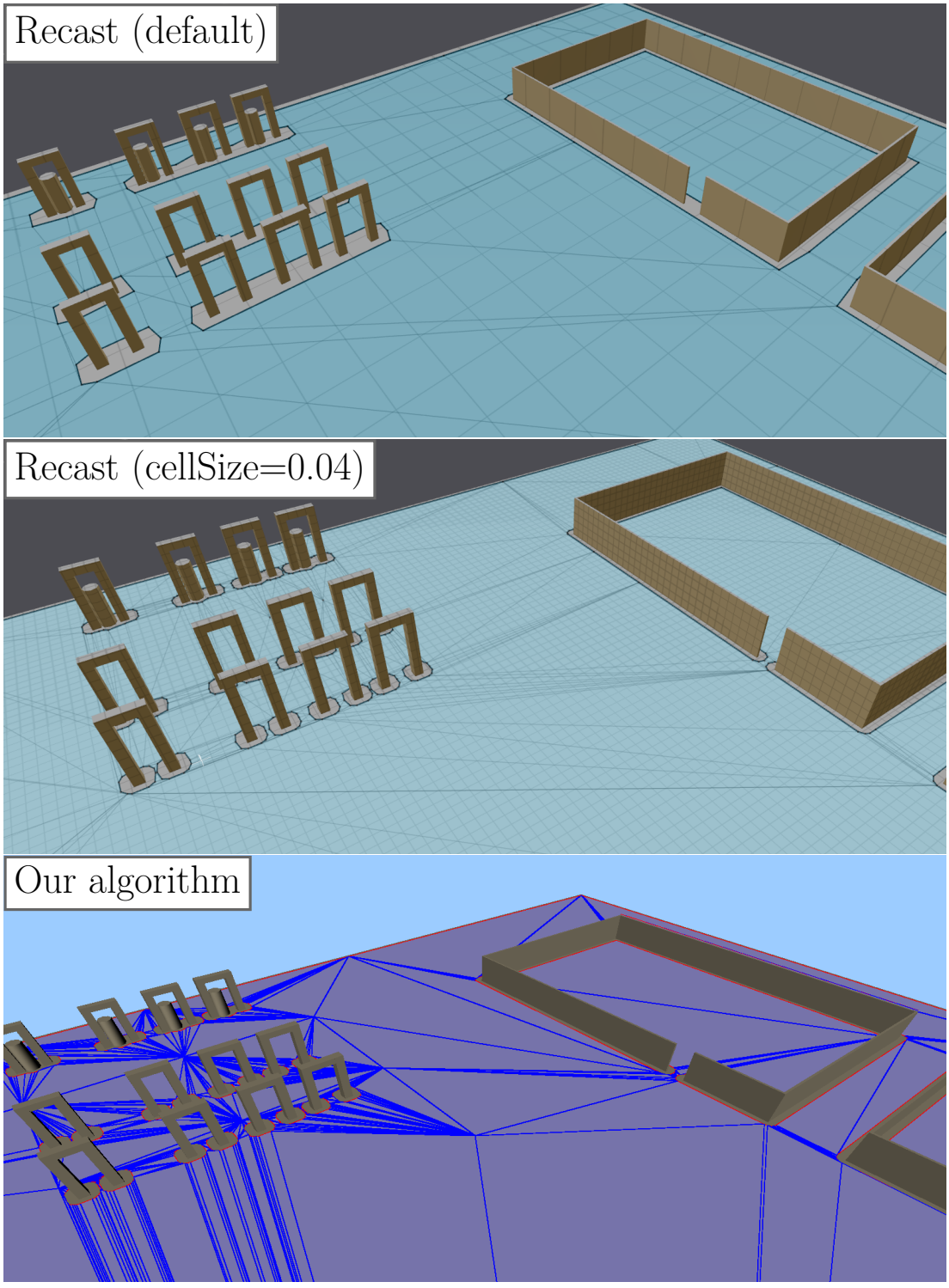


Figure 3.4: Comparison of navmeshes around rooms & doorways.

Case 2: Spiral maze

This case presents a simple circular maze, that simulates a situation where a path might run through a thin alleyway.

Algorithm	Time spent	Triangles
Recast (default)	130 ms	240, but insufficient detail
Recast (cellSize = 0.05)	8 047 ms	595†
Our algorithm	2 999 ms	1 395

† Still too conservative, navmesh was cut short.

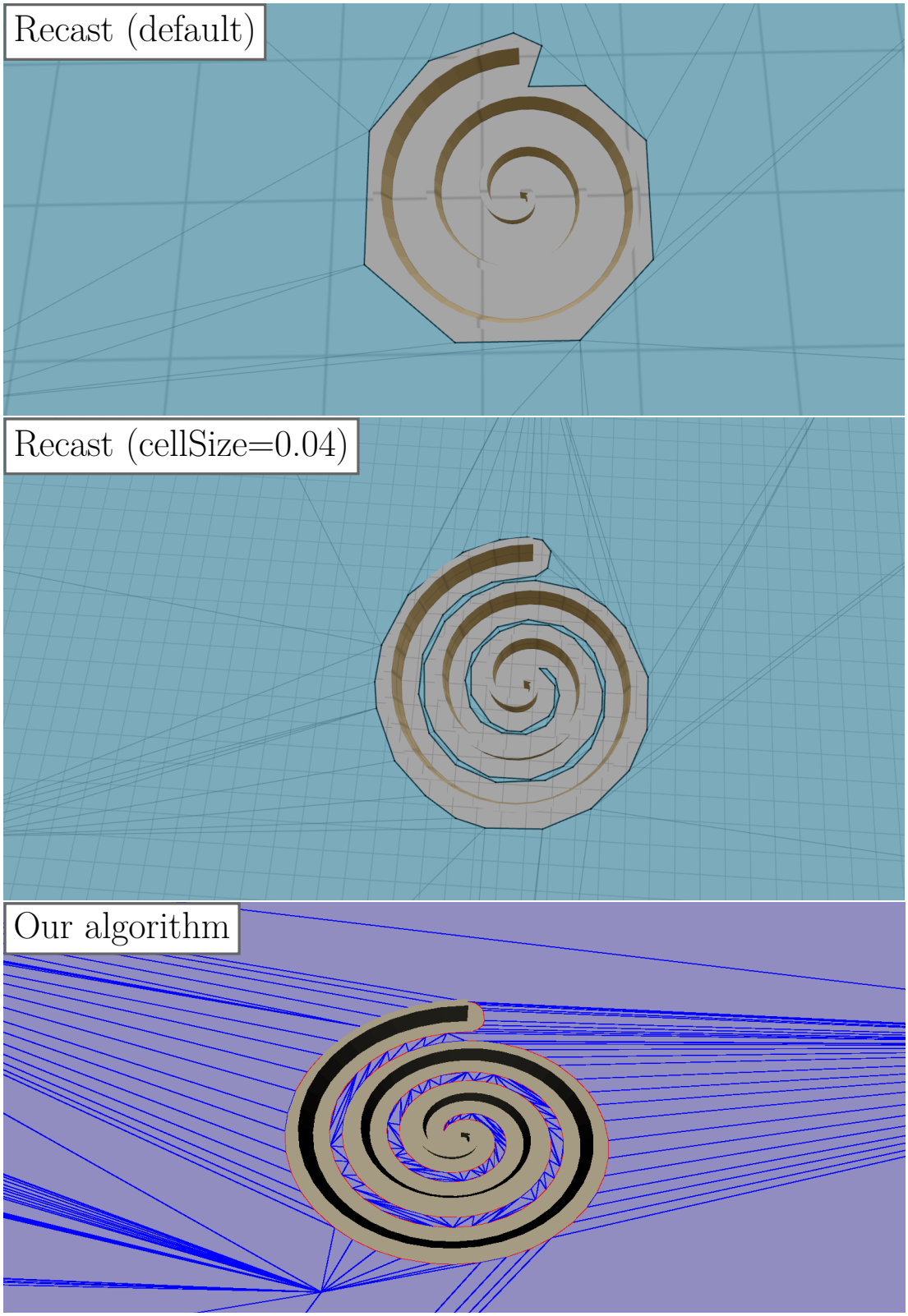


Figure 3.5: Comparison of navmeshes within a spiral. Notice that none of the Recast navmeshes fully reach into the centre.

Case 3: Overhangs

This testcase presents a ramp that creates an overhang above a path. A detailed navmesh should not overcompensate for the overhang and restrict the agent in movement any more than necessary. For voxelization algorithms, this presents an extra challenge. To have the same level of detail as our algorithm, they must reduce both the cell size and the cell height.

Algorithm	**Time spent**	**Triangles**
Recast (default)	130 ms	240, but insufficient detail
Recast (custom*)	44 487 ms	795†
Our algorithm	2 999 ms	1 395

* cellSize = 0.03, cellHeight = 0.01.
† Still too conservative, navmesh was cut short.

To preserve a comparable detail in the Recast navmesh, we had to push the cell size and cell height setting extremely low. Even then, Recast's algorithm was still too conservative and its navmesh was a few centimetres away from the real boundary. Further reduction of the cell dimensions was not possible as the algorithm ran out of available memory.

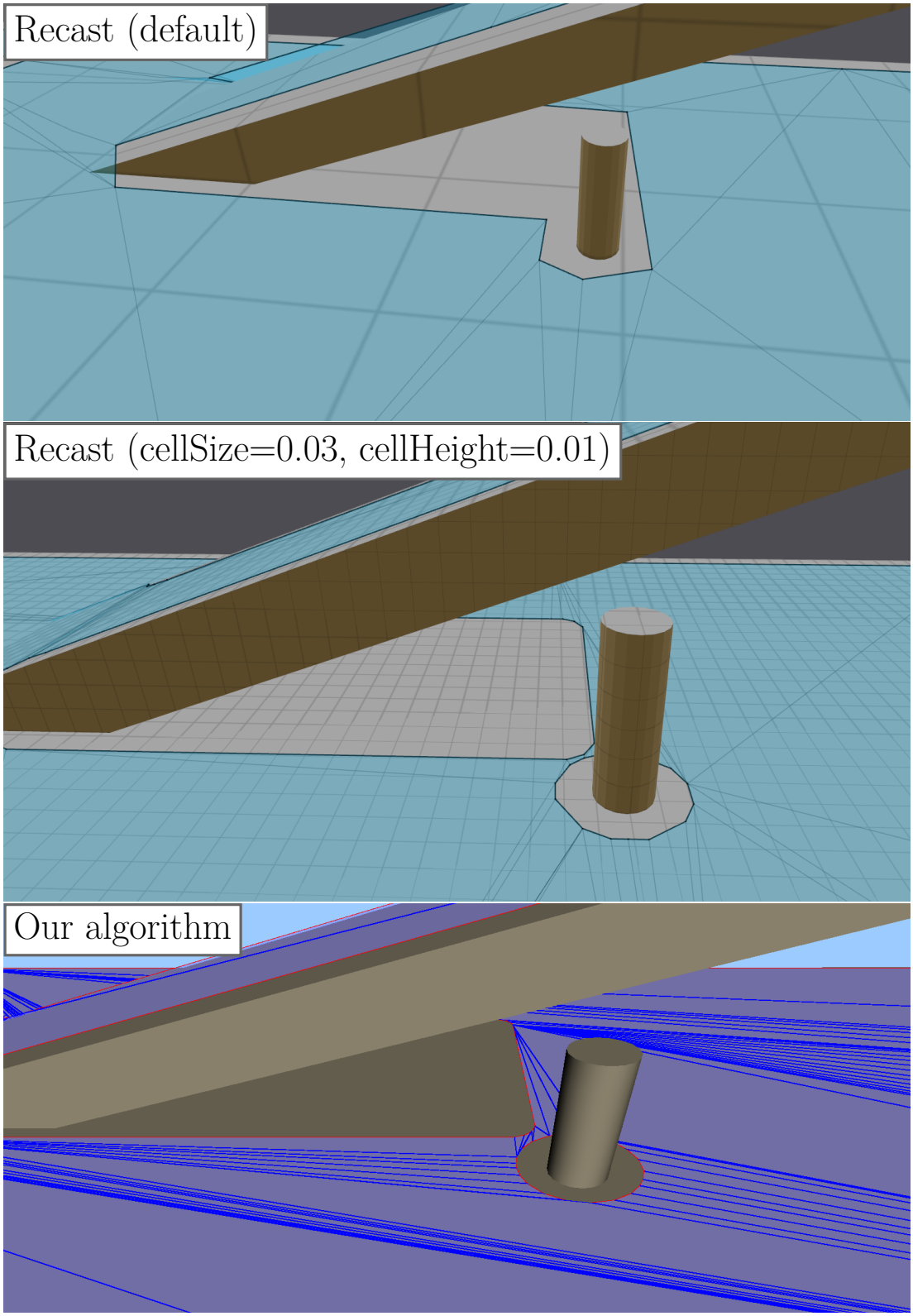


Figure 3.6: Comparison of navmeshes around overhangs. The cylinder represents the size of the agent and is positioned in such a way that it would fit under the ramp if walking straight ahead.

Summary

While it is possible to create a detailed navmesh using algorithms based on voxelization, in many cases we have to tweak parameters to different values based on the specific scene. Additionally, reducing voxel cell dimensions has a detrimental effect on performance and can sometimes result in the algorithm running out of resources.

Our algorithm works on the input geometry directly and is able to produce detailed navmeshes without running into such issues.

3.3 Problematic cases

One disadvantage of working directly with geometry directly is that our algorithm is more sensitive to its input. A bad result during cutting and gap closing can cause a disconnect in navmesh, which might be further enlarged after applying the wall avoidance step.

One known source of defects during cutting in our implementation are bugs in the polygon set operations in Boost library.

Boost geometry bugs A polygon difference calculation between two valid polygons using the Boost geometry library might return an empty result even in cases where the result should be some non-empty polygon. This error is related to the positions of the points of the two polygons and not to the loss of numerical accuracy. Thus it can't be avoided by filtering away polygons with small size.

The impact of these bugs for our Cutting implementation is that some triangles might be lost entirely. We try to avoid this by rejecting set operations with obviously wrong results. However, cuts skipped in such way might leave out geometry that will later complicate gap closing.

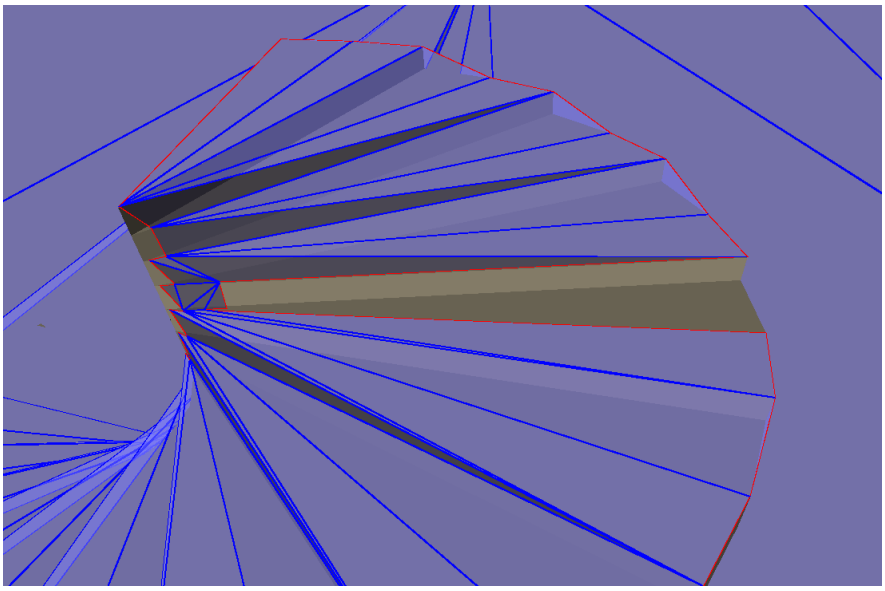


Figure 3.7: Triangle of a navmesh is missing because of failed set operation during cutting.

The authors of Boost geometry library are aware of the issues and some of the similar ones have already been fixed. But the process is still ongoing.

Other problematic cases Our algorithm is not always able to resolve clusters of many small intersecting triangles. Some of the small triangles might be missed during cutting and will prevent stitching of the local geometry together.

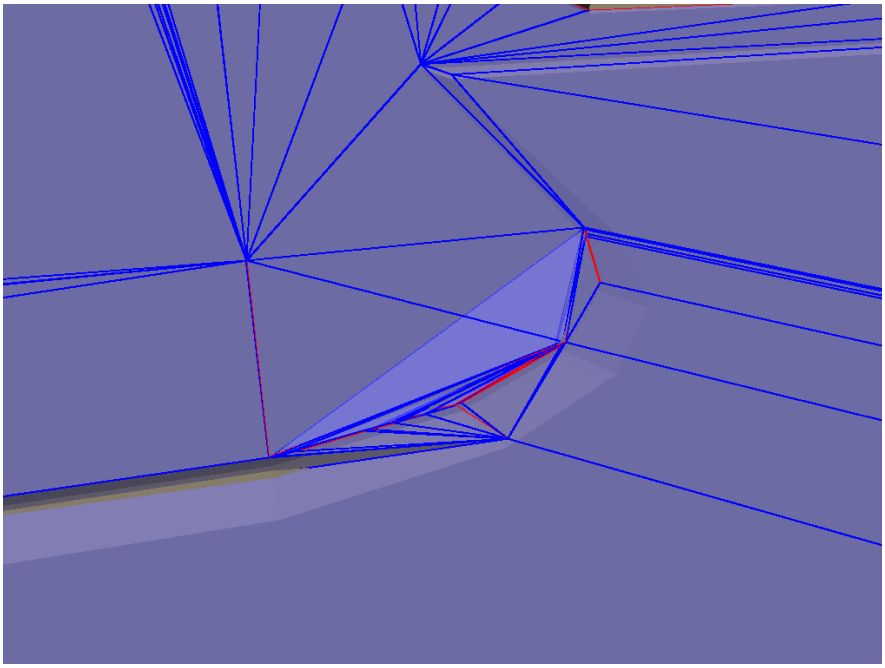


Figure 3.8: Cluster of small intersecting triangles that survived cutting.

4. Future work

In this chapter we describe our suggestions about how to further improve our navmesh generation algorithm and its implementation.

4.0.1 Ramp improvements

Better ramp placement In our Adding ramps section we chose to place ramps as steep as possible in order not to miss any stairs below. However, a gap created by thick walls during cutting might cause such ramps to miss the navmesh below.

A better placement of ramps would find a good ramp angle while taking the bottom navmesh geometry into account.

As we have shown in figure 2.35, finding the best angle for a ramp is a hard problem. But finding a ramp that is guaranteed to connect to at least some part of the navmesh would likely be a good enough solution.

Extra ramps for sharp corners to avoid miter effect In our Adding ramps section we avoided holes in the navmesh by making ramps of consecutive edges share their edges when possible. We achieved this by defining a *common aligned normal* vector in every ramp start point.

In the corners between individual ramps, this has caused the bottom vertex of the ramp to extend forward. The sharper the angle, the longer the extension would have to be.

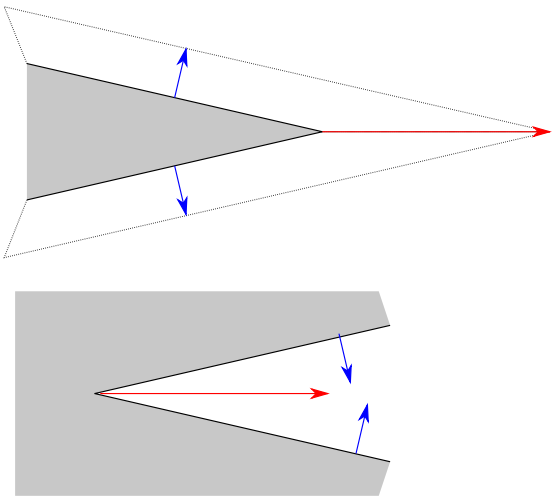


Figure 4.1: Extending ramp endpoints. The distance of the bottom ramp edge is based on the sharpness of the turn.

In order not to extend the ramp into infinity, we have put a limit on the maximum distance. For turns that would cross this limit, we would not use the shared normal and keep the ramp edges split. This could result in a hole in the mesh.

Solution A better solution to this problem would be to insert one extra ramp at the position of the vertex. The new ramp would have a shape of a triangle that would serve as a bridge between the two ramps of the edges.

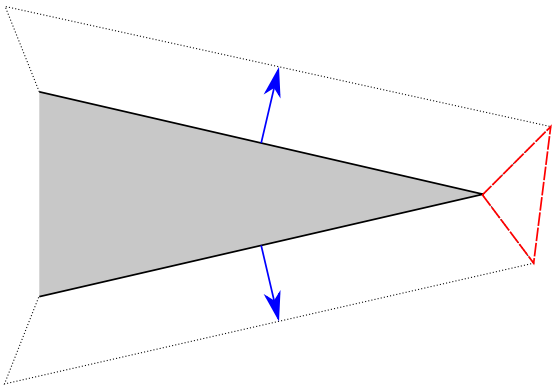


Figure 4.2: Adding a new triangular ramp to avoid extending shared edge into infinity.

This solution would allow us to keep the ramps connected even in very sharp turns.

4.0.2 Simplification improvements

To avoid losing important detail, we have been fairly restrictive in the way that we applied mesh decimation to our navmesh. Comparison with other navmesh generation algorithms has shown that our navmeshes have more triangles than others. Higher number of faces in the navmesh has a negative impact on pathfinding performance.

To further reduce geometric complexity of our navmesh, we need to lessen the restrictions we use during the last decimation step.

Collapse of boundaries inwards Our decimation process keeps mesh boundaries mostly intact and removes only collinear points. Keeping boundaries fully precise even after running wall clearance is likely unnecessary. As long as we don't move any part of the navmesh into a place that is not traversable or disconnect two paths, a less detailed navmesh might be sufficient.

One way how to reduce the complexity of boundaries is to use a line simplification algorithm by Douglas and Peucker [1973]. Hershberger and Snoeyink [2000] further improve the Douglas-Peucker algorithm by using path hulls to speed up the process.

However, any boundary simplification should not change the navmesh in a way that it would allow paths through blocked areas. To guarantee that, we should only simplify the boundaries inwards and never increase the covered area.

4.0.3 Experiment with clustering

Our algorithm can sometimes produce invalid results for groups of small intersecting geometry. One way to prevent that could be by using a vertex clustering algorithm similar to a one by Rossignac and Borrel [1993].

We would divide all the vertices of a scene into a uniform grid, and for each grid we would keep just one representative vertex. This clustering would simplify small features while keeping enough accuracy in the larger ones.

Conclusion

In this thesis we described and implemented a new navigation mesh generation algorithm. Compared to other algorithms that use voxelization, our algorithm is better suited for preserving important features of the geometry in navmesh. The algorithm consists of four main parts which we will now summarize.

In the cutting stage of the algorithm, we removed parts of the navmesh where there was insufficient vertical clearance. We did it by extending the world geometry downwards and removing any intersections with navmesh using polygon set operations.

Then we applied a gap closing algorithm to connect triangles of the mesh together and close any holes. For this step, we defined a set of rules to keep the topology of our mesh valid. We also implemented two contraction operations that were used to join the mesh.

We connected different parts of the navmesh together using ramps. The shape of our ramps closely matches the shape of the geometry below and no unsupported ramps were created. Consecutive ramps share their boundaries to keep the mesh connected.

Finally, to compensate for an agent's radius, we cut away areas of the navmesh to add clearance from walls.

To conclude, we compared the results of our algorithm to Recast, an open-source navmesh generation library. We have shown specific examples in which our algorithm produced better results. We have also identified areas for improvement, which we described in the Future work chapter.

Bibliography

Yariv Barkan, Antonin Guttman, Michael Stonebraker, Melinda Green, Paul Brook, Greg Douglas, and Gero Mueller. N-dimensional rtree implementation in c++, 2020. URL `https://github.com/nushoin/RTree`.

Boost. Boost C++ Libraries. `https://www.boost.org/`, 2021. Accessed 2021-07-05.

Pavel Borodin, Marcin Novotni, and Reinhard Klein. Progressive gap closing for mesh repairing. 08 2002. doi: 10.1007/978-1-4471-0103-1_13.

David H Douglas and Thomas K Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, 1973. doi: 10.3138/FM57-6770-U75U-7727. URL `https://doi.org/10.3138/FM57-6770-U75U-7727`.

Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, page 209–216, USA, 1997. ACM Press/Addison-Wesley Publishing Co. ISBN 0897918967. doi: 10.1145/258734.258849. URL `https://doi.org/10.1145/258734.258849`.

A. Gueziec, G. Taubin, F. Lazarus, and B. Hom. Cutting and stitching: converting sets of polygons to manifold surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 7(2):136–151, 2001. doi: 10.1109/2945.928166.

Antonin Guttman. R-trees a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, SIGMOD '84, page 48, 1984. doi: 10.1145/602259.602266. URL `http://www-db.deis.unibo.it/courses/SI-LS/papers/Gut84.pdf`.

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. doi: 10.1109/TSSC.1968.300136.

John Hershberger and Jack Snoeyink. Computing minimum length paths of a given homotopy class. *Computational Geometry*, 4(2):63–97, 1994. ISSN 0925-7721. doi: https://doi.org/10.1016/0925-7721(94)90010-8. URL `https://www.sciencedirect.com/science/article/pii/0925772194900108`.

John Hershberger and Jack Snoeyink. Speeding up the douglas-peucker line-simplification algorithm. *5th Intl Symp on Spatial Data Handling*, 11 2000.

S. Hertel and K. Mehlhorn. *Fast triangulation of simple polygons.*, volume 158 LNCS of *Lecture Notes in Computer Science*. Springer Verlag, Universität des Saarlandes, 1983. ISBN 9783540126898.

Marcelo Kallmann. Shortest paths with arbitrary clearance from navigation meshes. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, page 159–168, Goslar, DEU, 2010. Eurographics Association.

Leif Kobbelt, Stefan Bischoff, Mario Botsch, and Stefan Steinberg. Openmesh: A generic and efficient polygon mesh data structure, 2002. URL `http://www.graphics.rwth-aachen.de/media/papers/openmesh1.pdf`.

Mapbox. Fast, header-only polygon triangulation, 2021. URL `https://github.com/mapbox/earcut.hpp`. Accessed 2021-07-05.

Mikko Mononen. Recast & detour. `https://github.com/recastnavigation/recastnavigation`, 2009. Accessed 2021-07-01.

Mikko Mononen. Automatic navmesh generation via watershed partitioning. `https://digestingduck.blogspot.com/2010/02/slides-from-past.html`, 2010. Accessed 2021-07-07.

Tomas Möller. A fast triangle-triangle intersection test. *Journal of Graphics Tools*, 2(2):25–30, 1997. doi: 10.1080/10867651.1997.10487472. URL `https://doi.org/10.1080/10867651.1997.10487472`.

Jarek Rossignac and Paul Borrel. Multi-resolution 3d approximation for rendering complex scenes. pages 455–465, 01 1993. doi: 10.1007/978-3-642-78114-8_29.

Greg Snook. Simplified 3D movement and pathfinding using navigation meshes. In Mark DeLoura, editor, *Game Programming Gems*, pages 288–304. Charles River Media, 2000. ISBN 1584500492.

Paul Tozour. Building a near-optimal navigation mesh. In Steve Rabin, editor, *AI Game Programming Wisdom*, pages 171–185. Charles River Media, 2002.

List of Figures

A. Attachments

A.1 Implementation demo

We have implemented our navmesh generation algorithm inside a demo application that can be used to visualize its results. It is available as an executable file for Windows 10 located at `demo_application\libnavmesh_renderer.exe` in the digital attachments.

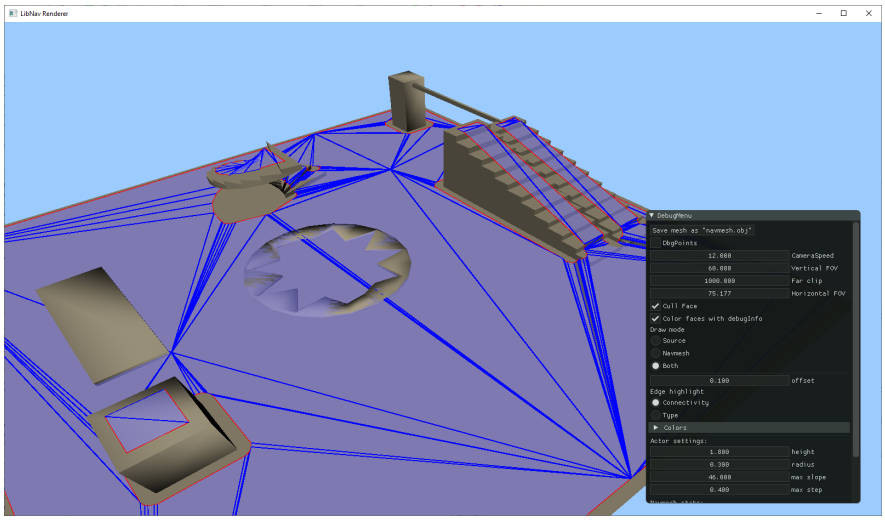


Figure A.1: Implementation of our algorithm in the attached demo.

A.1.1 System requirements

We recommend these minimum system requirements to run the demo application:

1. Operating system: Windows 10, 64 bit
2. CPU: x64, multi-core, 2 GHz or higher.
3. GPU: Must support OpenGL 4.3.
4. Additional requirements: Microsoft Visual C++ Redistributable 2019[1]

[1] `https://aka.ms/vs/16/release/vc_redist.x64.exe`

A.1.2 User manual

The demo application allows to generate and preview navmeshes for geometry stored in an `.OBJ` file. We provide a few example files inside the `demo_application/meshes` folder.

Usage

1. Start the application.
2. Drag and drop a mesh in `.OBJ` onto the window of the application
3. Wait until the navmesh is generated.
4. The Debug Window allows you to switch between the view of the input geometry, navmesh or both.
5. The navmesh can be exported as an `.OBJ` by pressing the "Save mesh as..." button in the Debug Window.

Controls

- Use WASD keys to control camera movement. Holding shift increases the speed of the movement.
- Click and hold the right mouse button to rotate the camera.
- Press G to generate navmesh again.
- Press L to turn the camera towards the geometry.

A.1.3 Source code

Source code of the demo application is located in the `source_code` directory

The demo application can be built from scratch using CMake[2] to generate a Visual Studio solution.

Prerequisites

- Microsoft Visual Studio 2017. Newer versions might work, but will require using newer version of Cinder library. Available at `https://visualstudio.microsoft.com/`.
- CMake 3.15 or newer. Available at `https://cmake.org/`
- Git. Available at `https://git-scm.com/`
- Boost libraries version 1.75 or newer. Available at `https://www.boost.org/`
- OpenMesh libraries version 8.0 or newer. Available at `https://openmesh.org/`

[2] `https://cmake.org/`

- Cinder library source code,
  revision `ae926dfd7e8368a655a95251f99bfd9d048523f7`.
  Available at `https://github.com/cinder/Cinder`

Steps

1. Use git to clone the Cinder library source code into the `/lib/Cinder` directory. We recommend using this specific revision:
   `ae926dfd7e8368a655a95251f99bfd9d048523f7`.
2. Set your environment variables:
   (a) `BOOST_ROOT` to point to the root directory of the Boost library.
   (b) `OPENMESH_ROOT` to point to the root directory of the OpenMesh library.
3. Use CMake to generate Visual Studio solution files for the project.
4. Build the Visual Studio solution

Third party libraries

Apart from the libraries we mentioned in the build prerequisites section, we use some additional open-source libraries whose source code we include directly among the source files. We provide a list of these libraries below. Their license terms are included next to their files.

1. Earcut [3]
2. OpenGL Mathematics (GLM) [4]
3. GoogleTest [5]
4. RTree [6]

[3] `https://github.com/mapbox/earcut.hpp`
[4] `https://github.com/g-truc/glm`
[5] `https://github.com/google/googletest`
[6] `https://github.com/nushoin/RTree`