Abstract

In crowd simulation, a navigation mesh of an environment is typically generated to allow efficient path planning. Some methods for navigation mesh generation require as input a representation of the walkable environment, that is, the parts of the environment that a human would be able to walk across. In a previous Master’s thesis, Polak developed a filtering pipeline that extracts such a walkable environment from a 3D environment [12]. A shortcoming of this method is that the result often contains gaps; for instance, stairs will yield a series of free-floating steps, with no information indicating that it is possible to walk from one step to the next.

In this thesis, we provide an algorithm extending the existing pipeline that tries to solve this. We detect gaps, up to a given size, between pairs of boundary edges of the walkable environment, and fill them with a quad. For connections between different cycles of boundary edges, we make a heuristic choice based on projected distance when there are multiple destinations for a given segment of a boundary edge. For connections between boundary edges in the same cycle, we project all the quads onto the ground plane and take their union, lifting the result back to 3D to fill the gap.

We compare our algorithm to two voxel-based methods of navigation mesh generation: Recast [10] and NEOGEN [11]. We find that our method gives more accurate results in many environments: it retains the exact representation of the walkable environment, semantically separates the gaps from the walkable areas, and requires no tweaking of parameters to obtain optimal results. However, there are some open problems that still need to be solved for our method to work fully automatically in practice, such as dealing with obstacles and the interaction of more than two cycles of boundary edges.
Figure 1: An example of undesirable gaps in the output of the pipeline from reference [12]. (a) shows the input of the pipeline. (b) shows in red the parts that get rejected because either the slope is too high or the vertical clearance is too low. (c) shows the output of the pipeline in blue and the rejected parts in transparent gray; there is no longer any information indicating the ability to walk from one step to the next. (d) shows the output of our algorithm, which adds the parts shown in green, indicating that traversal is possible. The resulting surfaces can be projected onto the ground plane without overlap, allowing the result to be put in a single layer.

1 Introduction

In crowd simulation, agents (in our case virtual humans) are expected to be able to navigate complex 3D environments. To facilitate this, a navigation mesh is typically generated. This mesh contains a representation of the areas that an agent may walk across, and is usually designed to allow efficient path planning for large numbers of agents. Some methods of navigation mesh generation (such as Recast [10] and NEOGEN [11]) take the full 3D environment as input, while others (such as the explicit corridor map (ECM) [15]) need a 2D or multi-layered representation as input. A method for generating multi-layered environments exists [8], but rather than working on the full 3D environment, it needs to be given only the walkable surfaces as input. For an overview of existing methods of navigation mesh generations and a detailed explanation of their requirements, see reference [17].

As such, for methods of navigation mesh generation that take an MLE as input, we need a way to extract the walkable surfaces from an arbitrary 3D environment. One such method was presented in the master’s thesis of Polak [12]. In this work, the walkable areas of an environment are extracted by means of a configurable filtering pipeline. This pipeline identifies those areas of the environment that have enough vertical clearance and a low enough slope for an agent to stand on. However, the result may contain gaps: parts of the mesh that are close together, but not (directly) connected to each other, even though an agent should be able to step between these parts of the mesh. Consider the example in Figure 1: the vertical parts of the stairs have been removed, leaving a set of free-floating steps, with no information indicating that it is possible to walk from one step to the next. The gaps may also be present in the input; see Figure 2 for an example. In this environment, there are many small holes in the floor, such as might be present in a metal grate. The pipeline rejects everything except the top, but this leaves all the holes in the output, even though they are small enough to be walked across.

In this thesis, we give an algorithm that extends Polak’s pipeline such that it can fill these gaps. A major difference with existing methods is that our algorithm makes the distinction between the existing geometry (those areas a person can stand on) and the geometry added to indicate the possibility of traversing a gap. The geometry of the input is not modified; this is important, as path planning algorithms and animation systems may handle the traversal of gaps differently from walking on a normal surface. For instance, a path planning algorithm may give a higher cost to paths traversing many gaps, using such routes only when they are significantly shorter than any possible alternatives, allowing the behaviour of the agents to more closely mimic that of real humans. Additionally, the animation system may place the feet of a virtual human such that they do not stand on a gap, as doing so might break the immersion in a game.

This thesis is structured as follows. In Section 2, we discuss existing methods for extracting the walkable parts of 3D environments, and look at existing methods that deal with gaps in polygonal meshes from a variety of perspectives, such as mesh repair and jumping agents. In Section 3, we give a summary of the pipeline described in reference [12] and give some definitions and notational conventions.
that are used in later sections. Section 4 defines the problem we are trying to solve more formally. In
Section 5, we describe our algorithm for detecting and filling gaps in walkable environments. Section
6 describes the experiments we performed to test the capabilities of our algorithm and compare it to
existing methods; the results are described and interpreted in Section 7. In Section 8, we conclude that
that our method gives more correct and semantically useful results than existing methods, but that
there are some limitations that still need to be overcome, such as dealing with obstacles and interactions
between more than two edges.

2 Related work

There are several areas of research that deal with the detection and filling of gaps in 3D environments.
In this section we summarise how existing methods of determining the walkable area (and/or building
a navigation mesh) deal with the presence of gaps in the input. We also consider the problem from the
perspective of mesh repair, in which the requirements on the result tend to be slightly different than
ours. Finally, we consider techniques that allow path planning for agents that can jump.

2.1 Walkable areas

Several automated methods for finding the walkable areas of a polygonal 3D environment exist. They
can be subdivided into two groups: volumetric and surface-based methods.

2.1.1 Volumetric approaches

Volumetric approaches work on a voxelised representation of the environment. The main advantage
of such methods is that by simply seeing if anything is contained inside a voxel, it circumvents many
complicated and sometimes ambiguous cases involving degeneracies, overlaps, intersections and gaps. A
disadvantage is that these methods tend to be sensitive to the size of the voxels: small voxels will cause
increased computation times on environments with large dimensions, whereas large voxels may result in
a lack of detail.

One widely used volumetric method for determining the walkable area is Recast [10], by Mononen. It
constructs a voxelised representation of the environment, then filters this on slope and vertical clearance
to determine the voxels an agent can stand on. After calculating a distance transform of the voxel grid,
a watershed algorithm is applied to partition the voxels into simple regions (meaning no overlaps and no
holes). The contours of these regions are then traced, simplified and triangulated. The resulting set of
triangles is used as the navigation mesh.

Recast deals with gaps robustly, but the result may be sensitive to the exact alignment of the voxel
grid due to the voxelisation. As such, two identical gaps may be handled differently simply because of
their location in the environment. There are also conflicting interests when it comes to voxel size: for
robustly handling gaps up to a certain size, we may want quite a coarse grid, but finding a detailed
representation of the walkable environment requires a much finer grid.
Oliva and Pelechano present a more advanced volumetric method, called NEOGEN [11]. It starts by voxelising the environment in a manner similar to Recast. This voxel representation is used to find regions with the required minimum vertical clearance. The contours of these regions are then refined, after which a detailed depth map of each region is calculated from a top-down perspective. The walkable area is then extracted from this render by introducing obstacles on pixels for which the difference in depth with a neighbour is higher than the maximal vertical step size. As the resolution of the render is much higher than that of the voxel grid, this allows for a more detailed representation of the walkable area around obstacles.

NEOGEN takes care to handle vertical gaps well, but any gap of which the horizontal distance is large enough to show up in the render will not be filled. As such, it will not work well for situations in which there are sizeable horizontal gaps, such as grates or wooden bridges that model the gaps between the floorboards. Furthermore, the authors report a voxel grid size limit of 128x128x128, which may cause unwanted results on large environments, as the voxel size will need to be increased, reducing the amount of detail in the result. This limitation could be overcome by dividing large environments into multiple sections of limited size, then stitching the results together, but the authors give no specific details as to how this might be achieved.

2.1.2 Surface-based approaches

Surface-based methods work directly on the polygonal representation of the environment, and as such have no problems with limited resolution or a lack of detail. The downside is that these methods need to deal with the full complexity and possible ambiguity of each environment. An example of a surface-based method is Lamarche’s TopoPlan [9]. It employs a prismatic spatial subdivision to find all triangles that overlap a given point in the ground plane quickly, thus facilitating the computation of regions with minimal vertical clearance. However, the method only deals with gaps that are strictly vertical. Like NEOGEN, this allows it to work with certain kinds of steps and stairs, but it does not handle horizontal gaps. TopoPlan also requires its input to contain no intersections between triangles, meaning it may require a preprocessing step to resolve any intersections.

2.2 Mesh repair

Mesh repair is concerned with the detection and reparation of errors in polygonal meshes. Depending on the source of a mesh, different errors may be present. For instance, gaps typically appear in CAD systems [4], where surfaces are modelled as sets of patches. The conversion of such patches to polygons may introduce gaps along the boundaries of disjoint patches. Gaps also often appear in models obtained by scanning a physical object: self-occlusion may prevent scanning of certain parts of the mesh, giving rise to holes in the final mesh.

The objectives and requirements of mesh repair are typically quite different from our own. For one, mesh repair may not restrict itself to the creation of new polygons, but may also alter the input geometry, which is undesirable for our application, as we wish to maintain a clear distinction between the walkable surfaces and traversable gaps. Moreover, the goal will often be to create a closed mesh, having a clearly defined inside and outside, which is clearly not a property we want for our application. A good overview of mesh repair techniques, including common sources of errors, is given by Attene et al. [4]. Here we provide an overview of some algorithms specifically for the closing of gaps.

Barequet and Kumar [5] provide an algorithm that deals with several types of common mesh defects, including gaps and holes. Their approach is similar to ours as it considers pairs of boundary edges, but it resolves gaps by merging close boundary edges, which we cannot allow in our application: we want to keep a clear distinction between the parts of an environment an agent can stand on, and those that it can step across. It also relies on user input to resolve ambiguous cases, which would become tedious for very large inputs. Borodin et al. [7] provide a method based on a vertex-edge contraction operator, which also modifies the input geometry, which, again, we cannot allow.

Barequet and Sharir [6] give an algorithm that most closely resembles our own. They discretise the boundary edges into segments of fixed length, and then let each segment vote on the segments that are within a user-specified distance. These votes are then converted into suggestions for matches between parts of the boundaries. From all of these suggestions a non-overlapping subset is chosen heuristically, after which the area between each matched partial boundary is filled through stitching. This method does
not modify the original geometry, but it assumes a close match in the shape of disconnected boundaries, which is not necessarily present in our application.

2.3 Jumping agents

There are obvious similarities between agents that can step over gaps and those that can jump. Several systems for planning paths of jumping agents exist; a good overview can be found in Van de Kerkhof’s master’s thesis [14]. These techniques typically employ a form of sampling: a large set of points on the walkable areas are chosen in some fashion, and the places an agent can jump to from those points are calculated. The resulting possibilities are stored as jump links, and the path planning algorithm is extended to use these.

For our application, such an approach might work well for finding the gaps that an agent can step over, but the agent would not be able to deal with small holes, such as those that might appear in a metal grid: the sampling would have to be very dense to adequately cover the surface, and the potential jumps would likely end inside another hole. Furthermore, these techniques take into account the trajectory an agent follows through the air when jumping. This is not necessary for our application, and may give undesired results on purely vertical gaps. An advantage of a method based on explicitly stored jump links is that it does not suffer from many of the ambiguous cases that surface-based methods encounter: we can simply store multiple destinations from a single jump point. However, this comes at the cost of an increase in the complexity of the path planning algorithm. Finally, the sample-based nature means the results can be arbitrarily bad when the samples are chosen poorly, similarly to how voxel-based methods may miss gaps because of poor alignment with the voxel grid.

2.4 Conclusion

In conclusion, there are many different approaches to finding and filling gaps in polygonal meshes. Volumetric methods compute a voxelisation of the mesh, which inherently closes gaps that fall in the same voxel. Mesh repair offers many different methods for closing gaps, but many of them modify the input geometry, and all assume that the aim is to obtain a closed mesh, which is not desirable for our application. Finally, path planning for jumping agents is usually achieved by augmenting the navigation mesh with jump links, which are sampled at random. This satisfies our need for a clear distinction between the areas an agent can walk on and those he can step across, but existing methods will have difficulty with small holes in the walkable surface. They also take into account the trajectory an agent travels when jumping, which is not required for our application.

Our contribution in this thesis is the following. We provide a method for the detection and filling of gaps that does not alter the input geometry, handles both holes and gaps, and is specifically designed with walking agents in mind. This allows us to semantically annotate the resulting polygons as being part of the walkable geometry or part of a gap or hole, which may then be used by path planning and animation systems during simulation.

3 Preliminaries

This work continues directly from Polak’s master’s thesis [12], which defines a filtering pipeline that can be used to find those parts of a polygonal environment with low enough slope and enough vertical clearance for an agent to stand on. As such, we largely use the same definitions and assumptions. For convenience and clarity, we will include the most important ones here. Note that some definitions have been slightly modified from the original, as there were minor issues with the correctness of the formulations.

3.1 Geometry

We will first give some general notational conventions concerning the way we represent geometry. We work in three dimensional Euclidean space, with points \( p = (p_x, p_y, p_z) \in \mathbb{R}^3 \). We take the XY-plane as the ground plane, with the positive Z-axis pointing up. We take the XYZ-plane as the ground plane, with the positive Z-axis pointing up. We work on polygonal meshes containing only triangles, where each triangle \( T \) is a list of vertices \( \{v_1, v_2, v_3\} \), which we assume to be given in
counter-clockwise order. Each vertex corresponds to a point in $\mathbb{R}^3$. The normal of a triangle is the unit vector perpendicular to each edge and can be calculated as $\text{normal}(T) = \frac{(v_2-v_1) \times (v_3-v_1)}{|(v_2-v_1) \times (v_3-v_1)|}$. We define the slope of a triangle to be the angle between the triangle and the ground plane, calculated as $\text{slope}(T) = \cos^{-1}(\text{normal}(T) \cdot Z)$, where $Z$ is the unit vector pointing up, i.e. $Z = (0,0,1)$.

### 3.2 Walkable environments

A 3D walkable environment $E$ consists of a set of walkable triangles $\mathcal{P}(E)$ and a set of obstacle lines segments $\mathcal{L}(E)$, which is a subset of the edges of the walkable triangles. For an environment to be walkable, we need that no triangle in $\mathcal{P}(E)$ exceeds a given maximum slope $\alpha_{\text{max}}$, and that each point in the interior of a triangle has sufficient vertical clearance. We need the obstacle line segments to represent obstacles that are perfectly vertical: their projection on the ground plane has no area, so we cannot represent the obstacle by cutting away the part of the mesh that does not have enough vertical clearance. We then arrive at the following definition of a walkable environment:

**Definition 1** ([12]). Given a maximum slope $0 \leq \alpha_{\text{max}} < \pi$ in radians and a minimum vertical clearance $v_{\text{min}} > 0$, an environment $E$ is walkable with respect to an environment $F$, if for every triangle $P \in \mathcal{P}(E)$:

- $\text{slope}(P) \leq \alpha_{\text{max}}$.

  - For every point $p \in P$, one of the following holds:
    - $\forall Q \in \mathcal{P}(E) \cup \mathcal{P}(F), q \in Q : (Q \neq P \land q_x = p_x \land q_y = p_y) \rightarrow (q_z - p_z \geq v_{\text{min}} \lor q_z < p_z)$. 
    - $\exists L \in \mathcal{L}(E) : p \in L$.

That is, each triangle has a slope no more than $\alpha_{\text{max}}$, and each point on its surface lies on an obstacle segment or has no other points directly above it within $v_{\text{min}}$.

To enable agents to properly navigate the environment, the triangles of the environment must also be connected:

**Definition 2** ([12]). A walkable environment $E$ is connected if every two triangles $P, Q \in \mathcal{P}(E)$ that contain more than one of the same points on their boundary that are not on an obstacle edge are topological neighbours, i.e. they share an edge.

Additionally, we want that the walkable environment completely describes all the walkable parts of the input. We define this constraint in the following definitions:

**Definition 3** ([12]). An environment $E$ is contained within an environment $F$, if $\forall P \in \mathcal{P}(E), p \in \text{interior}(P), \exists Q \in \mathcal{P}(F) : p \in \text{interior}(Q)$.

**Definition 4** ([12]). An environment $E$ is a walkable area of an environment $F$, if $E$ is both walkable with respect to $F$ and contained within $F$.

**Definition 5** ([12]). An environment $E$ is the complete walkable area of an environment $F$, if no point could be added to the interior of a triangle in $\mathcal{P}(E)$ or removed from a line segment in $\mathcal{L}(E)$, while keeping $E$ a walkable area of $F$.

We assume that the input to our own algorithm is such a complete walkable area. Examples of complete walkable areas and how they relate to the input can be seen in Figures 1(c) and 2(c).

### 3.3 Filtering pipeline

The algorithm to extract a complete walkable area is designed as a configurable sequence of filters. Each filter aims to solve a specific error in the input, enforce a specific constraint placed upon complete walkable areas, or prepare the result for the generation of a navigation mesh. The available filters are as follows [12]:

- Degeneracy filter: removes any degenerate polygons (that is, polygons with no area) in the environment.
Figure 3: Illustrations of our additional definitions. (a) shows boundary edges in red, and other edges in black. (b) shows three connected components in different colours. Note that the blue and green triangles are in separate components, as they only share one point. (c) shows two connected components which are also coplanar regions, as all triangles in the component have the same normal vector. (d) shows the associated plane of the red edge in transparent green. This plane contains the red boundary edge and is perpendicular to the ground plane.

- Duplication filter: merges any vertices in the environment that are closer to each other than a given threshold.
- Intersection filter: resolves intersecting polygons by decomposing them.
- Slope filter: removes those polygons that exceed the maximum slope.
- Vertical clearance filter: removes the (parts of) polygons that do not have enough vertical clearance.
- Area filter: removes connected components that have too little surface area.
- Simplification filter: simplifies the environment through retriangulation.

We will be adding a gap filter to this pipeline, which will serve to connect regions that are close enough together that an agent will have no trouble stepping from one to the other. Due to the requirements on its input, the gap filter needs to be placed after the slope and vertical clearance filter. It is also advisable to not precede it with an invocation of the area filter, as this may remove components from the scene that the gap filter would have connected to a larger whole. A typical order would be: slope filter, vertical clearance filter, degeneracy filter, gap filter, area filter, layer subdivision filter.

3.4 Additional definitions

In addition to the definitions and assumptions we take from the previous work, we define several terms to allow unambiguous discussion of the notion of gaps. The concepts are illustrated in Figure 3.

**Definition 6.** An edge $e = \{v_1, v_2\}$ is a boundary edge when it is part of only one triangle.

**Definition 7.** A set of triangles $C \subseteq P(E)$ is a connected component when the corresponding set of vertices in the dual graph $G$ of $P(E)$ forms a connected component.

**Definition 8.** A connected component $C$ is a coplanar region when $\forall T_1, T_2 \in C : \text{normal}(T_1) = \text{normal}(T_2)$

**Definition 9.** The associated plane of a boundary edge $e = \{v_1, v_2\}$ is the plane through $v_1$ and $v_2$ perpendicular to the ground plane, with the triangle that $e$ is a part of on the negative side. In point-normal notation, it is the plane formed by any point on $e$ (we arbitrarily take $v_1$), and the normal calculated as $n = \frac{(v_2 - v_1) \times Z}{||v_2 - v_1||Z||}$, where $Z$ is the vector pointing straight up.
4 Problem statement

To give an algorithm for the detection and filling of gaps, we must first define what a gap is. Informally speaking, we say a gap is a region where parts of the mesh are close to each other, but not directly connected. A formal definition is, however, more complex, as we will show in this section. We make two important assumptions:

Assumption 1. We want to fill gaps up to some maximum distance $d_{\text{max}} < \frac{v_{\text{min}}}{2}$.

Assumption 2. The maximum slope $\alpha_{\text{max}}$ is less than $\frac{\pi}{2} - \arcsin\left(\frac{d_{\text{max}}}{v_{\text{min}}}\right)$ radians.

The first assumption will prevent ambiguities when an edge has multiple edges it can connect to, as it ensures that all edges within $d_{\text{max}}$ of the same edge have no overlaps when projected onto the ground plane. The second assumption is required for the proof of Theorem 1 below. Note that neither restriction is problematic in practice: $d_{\text{max}}$ will be on the order of 0.3 metres (large enough to cover typical stairs and steps, but not so large as to cause difficulty for a typical person), while $v_{\text{min}}$ will be about 2 metres, roughly the height of a person. $\alpha_{\text{max}}$ will also normally be at most 45$^\circ$ (large enough to handle stairs modelled as ramps), much less than the maximum restriction of 60$^\circ$ that our assumption imposes.

We have two further assumptions about the previous filter steps that have already been applied when we perform the gap filter:

Assumption 3. The slope- and vertical clearance filters have already been applied.

Assumption 4. There are no degenerate triangles in the scene.

These assumptions ensure that the environment has the properties necessary for the gap filter to run successfully.

We would like to only consider boundary edges when filling gaps, ignoring the interior of each connected component. The primary reason for this is to avoid the creation of singular edges (edges incident to more than two polygons), which would occur when we fill a gap between a boundary edge and the interior of a connected component. Not all methods of navigation mesh generation can handle such geometry (e.g. the ECM method [15]), so we would like to avoid such situations. Fortunately, we can prove that up to a certain distance to the interior of a connected component, there must also be a boundary edge in the same component that has a distance of at most $d_{\text{max}}$.

Theorem 1. If the distance between an edge $e_1$ and a coplanar region $C$ with slope $\alpha_{\text{max}}$, having closest points $p_1$ on $e_1$ and $p_2$ on $C$, is $d \leq d_{\text{max}}$, and $p_2$ is not on the negative side of the associated plane of $e_1$, then there must be at least one boundary edge $e_2$ in $C$ that has a distance of no more than $d' = \frac{d}{\sin(\pi - \alpha_{\text{max}})}$ to $e_1$.

Proof. We derive an upper bound on the distance between $e_1$ and $e_2$. If the point on $C$ closest to $e_1$ is on a boundary edge, we are done; otherwise, it must be in the interior of $C$. See Figure 4; it is given that the distance between $p_1$ and $p_2$ is $d$. We look at the intersection of $C$ with the vertical plane through $p_1$ and $p_2$. This will be a collection of line segments, call it $S$. We consider the associated plane of $e_1$, and call the point on $S$ closest to it and not on its negative side $p_3$. Note that there must be a point on $S$ closer to the associated plane of $e_1$ than $p_2$: our constraint on vertical clearance ensures that all points directly below $p_1$ must either be on a boundary edge of $C$, or outside of $C$. This is because the distance between $p_1$ and a point directly below it in the same plane as $C$ has distance $\frac{d}{\sin(\pi - \alpha_{\text{max}})} < \frac{d_{\text{max}}}{\sin\left(\frac{\pi}{2} - \arcsin\left(\frac{d_{\text{max}}}{v_{\text{min}}}\right)\right)} = v_{\text{min}}$. This means that $p_3$ must be on a boundary edge $e_2$ of $C$. If $p_3$ is directly below $p_1$, we obtain $|p_1 - p_3| = \frac{d}{\sin\left(\frac{\pi}{2} - \alpha_{\text{max}}\right)} = d'$. If $p_3$ is not directly below $p_1$, $|p_1 - p_3|$ must be smaller than $d'$, as $p_3$ cannot be behind $p_2$. Combining both cases, we obtain $|p_1 - p_3| \leq d'$.

The theorem excludes those cases where $p_2$ is on the negative side of the associated plane of $e_1$, as the proof does not hold in those situations. However, as we will see later on, we do not consider connections to the negative side of the associated plane, as the semantic interpretation would be that an agent bridges a gap by stepping back from the edge of the walkable area. As such, this extra condition does not restrict us in practice. Furthermore, regions will not always be coplanar. However, the same theorem easily holds
Figure 4: A coplanar region $C$ with maximum slope and a boundary edge $e_1$ that is close to it. $p_1$ and $p_2$ are the points at which $C$ and $e_1$ are closest to each other. The dashed line $S$ represents the intersection of $C$ with the vertical plane through $p_1$ and $p_2$. $p_3$ is the lowest point on this intersection that is not on the negative side of the associated plane of $e_1$, and must be on a boundary edge, in this case $e_2$, because of the constraint on vertical clearance. Figure (a) shows the full situation, whereas Figure (b) shows a side view, in which the horizontal dashed line is parallel to the ground plane.
for a connected region in which not all triangles have slope $\alpha_{max}$, as $p_3$ can never be moved further away from $p_1$ by decreasing the slope of the triangles.

When the distance of a boundary edge $e_1$ to a connected component $C$ is larger than $d_{max} * \sin(\frac{\pi}{2} - \alpha_{max})$, it is possible that no boundary edge in $C$ has distance at most $d_{max}$ to $e_1$. As such, if we only consider gaps between boundary edges, we may ignore gaps where a boundary edge is closer than $d_{max}$ to the interior of a connected component. However, as the region of distances where this can occur is relatively small (e.g. between 0.7 and 1 times $d_{max}$ when $\alpha_{max} = 45^\circ$), and as the complexity of the algorithm would increase significantly if we also consider the interior of the triangles, we still choose to detect gaps only between pairs of boundary edges.

To find gaps up to a maximum distance of $d_{max}$, we must first define what we mean by distance between boundary edges. We want to connect those parts of two boundary edges that are within $d_{max}$ of each other. We might then define the gap $G$ we want to fill between two boundary edges $e_1$ and $e_2$ as follows:

$$G = \{ p \mid \exists x, y : \text{OnSegment}(x, e_1) \land \text{OnSegment}(y, e_2) \land \text{OnSegment}(p, \{x, y\}) \land |x - y| \leq d_{max} \} \quad (1)$$

where OnSegment($a, \{b, c\}$) is true iff the point $a$ lies on the line segment between points $b$ and $c$.

Unfortunately, the resulting shape is not necessarily linear, meaning we cannot represent it with triangles. Therefore, we use the representation seen in Figure 5. The gap between boundary edges $e_1$ and $e_2$ is filled by a gap filler $G'$, which is defined by two connections $c_1$ and $c_2$, which are line segments with their endpoints on the two boundary edges. Connections must have a length of at most $d_{max}$, which implies that $G' \subseteq G$. We can choose any two connections with length at most $d_{max}$, but we would like the gap filler to be as wide as possible. In this context, we define the width of a gap as follows:

**Definition 10.** If the gap filler $G'$ between boundary edges $e_1$ and $e_2$ is given by connections $c_1 = \{p_1, p_2\}$ and $c_2 = \{p_3, p_4\}$, with $p_1$ and $p_3$ on $e_1$ and $p_2$ and $p_4$ on $e_2$, then the width of $G'$ is defined as $\text{width}(G') = \min(|p_1 - p_3|, |p_2 - p_4|)$.

The reason we want the gap fillers to be as wide as possible is that in a simulation framework using the ECM method [16], the width of a corridor determines the maximum width an agent can have and still pass through it. As such, maximising the width of our gap fillers maximises the width of the largest agent that can pass over it. In addition, the symmetry guarantees that the gap between $e_1$ and $e_2$ is always the same as the gap between $e_2$ and $e_1$.

For many cases (as shown in Section 5.2), we have multiple ways of connecting a gap that are of maximum width. When the width is limited by the length of the shorter edge, we make the other side as wide as possible. In other cases, we maximise the width by making $|p_1 - p_3|$ and $|p_2 - p_4|$ equal and as large as possible.

We need one additional property for our definition of a gap between boundary edges to make sense: the resulting mesh has to be orientable. This means that the side on which we can walk should be consistent – it cannot change when we move to an adjacent triangle. This can be violated when we connect a
boundary edge to one that is “behind” it when viewed from the triangle it is a part of. More formally, for the gap between $e_1 = \{v_1, v_2\}$ and $e_2 = \{v_3, v_4\}$, defined by the line segments $c_1 = \{p_1, p_2\}$ and $c_2 = \{p_3, p_4\}$, we need that $p_1$ and $p_2$ are not on the negative side of the vertical plane through $v_3$ and $v_4$, and that $p_3$ and $p_4$ are not on the negative side of the vertical plane through $v_1$ and $v_2$. This is further explained and illustrated in Section 5.2.3.

It is important to note that the presence of a gap filler does not imply a person can step directly between any two points on its boundary: if a gap filler is very wide, many such pairs of points may be much further than $d_{\text{max}}$ apart. Such cases will need to be handled during simulation by the path planning algorithm or the animation system. For instance, the path planning algorithm may give a higher cost to sections of the path running over a gap filler, or the animation system may place one foot of the agent on one side of the gap and the other foot on the other side when travelling along a long, narrow gap.

5 Gap filter

Our gap filter consists of four major steps:

1. Find cycles of boundary edges.
2. Detect gaps for each pair of boundary edge cycles.
3. Use the detected gaps to connect cycles.
4. Use the detected gaps to fill holes.

In this section we will describe each of these steps in detail. Note that filling a hole means connecting a cycle of boundary edges with itself. For time complexity, we always take $n$ to equal the number of triangles and $m$ to equal the number of boundary edges in the input. Note that in the worst case, $m \in O(n)$, but we typically expect the number of boundary edges to be lower.

5.1 Finding cycles of boundary edges

The first step is simple: we iterate over all triangles in the input to find a boundary edge. When we find one, we look at all triangles that contain the boundary edge’s second vertex, looking for the next boundary edge. If there is more than one, we take the boundary edge most counterclockwise to the previous edge; we choose this option because we do not consider triangles that share only one point to be adjacent. We continue until we reach the edge we started with, giving us a cycle of boundary edges. Once we have found a complete cycle, we continue iterating over all triangles in the input until we find another boundary edge. By marking all the edges that we visit, we can ensure that we find each cycle only once. The time complexity of finding the cycles is $O(n)$.

5.2 Detecting gaps

In this step we detect gaps between pairs of boundary edges. For each boundary edge $e_1 = \{v_1, v_2\}$ we determine its axis-aligned bounding box and expand it by $d_{\text{max}}$ along all positive and negative axes. This allows us to query an R-tree containing all the boundary edges in the environment to efficiently find boundary edges that may be within $d_{\text{max}}$ of the edge. For each boundary edge $e_2 = \{v_3, v_4\}$ in the query result, we attempt to make a connection with $e_1$. As there can be up to $O(m)$ boundary edges that intersect the bounding box, we consider $O(m)$ pairs of boundary edges. Over all boundary edges we need to consider a total of at most $O(m^2)$ pairs of boundary edges. Connecting a pair of edges consists of three steps, each of which we will detail below:

1. Split edges at closest points.
2. Find connection between pairs of split edges.
3. Limit result to non-negative side of associated plane.

As each of these steps takes a constant amount of time, detecting all gaps has a worst-case time complexity of $O(m^2)$. 


5.2.1 Split edges

We first find the pair of points on $e_1$ and $e_2$ with minimum distance to each other; call these points $c_1$ and $c_2$, with $c_1$ on $e_1$ and $c_2$ on $e_2$. If $|c_1 - c_2| > d_{\text{max}}$, we know that there exist no points on $e_1$ and $e_2$ with distance at most $d_{\text{max}}$, so the gap is too large, and we discard it. Similarly, unless the edges are parallel, if $|c_1 - c_2| = d_{\text{max}}$, there is only one pair of points with distance at most $d_{\text{max}}$: the gap is discarded, as the resulting connection would have no area. Otherwise, we use the closest points to split the edges into $e_{11} = \{v_1, c_1\}$, $e_{12} = \{c_1, v_2\}$, $e_{21} = \{c_2, v_3\}$ and $e_{22} = \{v_2, v_4\}$. Because our edges are always counterclockwise with respect to the polygon they are a part of, we know that we must make one connection between $e_{11}$ and $e_{22}$ and another between $e_{12}$ and $e_{21}$.

Performing this split makes it easier to find points on the edges with distance at most $d_{\text{max}}$, as we know that the pairs of split edges are closest at two vertices ($c_1$ and $c_2$). If we parameterise each split edge from the closest point (e.g. $e_{11}(t) = c_1 + t \cdot (v_1 - c_1)$, with $t \in [0, 1]$), we know that due to the linearity of the edges, the distance between the respective pairs must be non-decreasing; that is, if we increase $t$, we do not decrease $|e_{11}(t) - e_{22}(t)|$ and $|e_{12}(t) - e_{21}(t)|$.

This approach does not work when $e_1$ and $e_2$ are parallel: in this case, we have no unique closest points $c_1$ and $c_2$. Instead, we try to make the vertex-edge connection described below for each vertex of $e_1$ and $e_2$.

5.2.2 Connect pairs of split edges

We assume without loss of generality that we are connecting $e_{11}$ and $e_{22}$. We first simply check if $|v_1 - v_4| \leq d_{\text{max}}$: if this is true, we connect $v_1$ to $v_4$. Otherwise, as we want to maximise the width of the connection, we find the points we want to be equally far from $c_1$ and $c_2$. We achieve this by making both edges equal in length: assuming $c_{11}$ is the shorter edge, we cut off part of the end of $e_{22}$; call this shortened edge $e_{22}'$. We can calculate it as $e_{22}' = \{v_4', c_2\}$, with $v_4' = c_2 + |v_1 - c_1| \cdot \frac{v_4 - c_2}{|v_4 - c_2|}$; an example can be seen in Figure 6. This makes $|e_{11}(t) - e_{11}(0)| = |e_{22}'(t) - e_{22}'(0)|$, which ensures that we maximise the width of the connection when we find a value of $t$ for which $|e_{11}(t) - e_{22}'(t)| = d_{\text{max}}$.

However, we will later have to specifically handle the case where the maximum width is limited by the length of one or both edges.

We connect each pair of split edges by finding the value of $t$ for which their distance is exactly $d_{\text{max}}$, which means solving $|e_{11}(t) - e_{22}'(t)| = d_{\text{max}}$ for $t$. If we take $d(t) = e_{11}(t) - e_{22}'(t)$, this is equivalent to solving the equation $(d(t) \cdot d(t)) - d_{\text{max}}^2 = 0$ for $t$. This is a quadratic equation of the form $at^2 + bt + c$, which is easily solved with the well-known formula $t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$, with $D = b^2 - 4ac$.

We obtain two $t$-values at which the supporting lines of $e_{11}$ and $e_{22}$ are at distance $d_{\text{max}}$. As we know that the edges are not parallel and have their closest points at distance $d_{\text{max}}$, we know that the larger of the two $t$-values must always be greater than zero and the smaller less than zero. We want the larger value, as the other one is never on either edge. Note, however, that it is possible for the larger value of $t$ to be larger than one, meaning that the point at which the supporting lines have distance $d_{\text{max}}$ lies beyond $v_1$.

In the case where the points associated with the larger $t$-value lie beyond $v_1$, we connect the vertex of the shorter edge to the point furthest along the longer edge with distance exactly $d_{\text{max}}$: call this point $p_1$. See Figure 8 for an example. Assuming without loss of generality that the vertex we are connecting is $v_1$, we can find this point in the following manner. First, we project $v_1$ onto the supporting line $l_1$ of $e_{22}$; call this projected point $p_2$. Since we know that $|v_1 - p_1| = d_{\text{max}}$ and $\langle v_1, p_1, p_2 \rangle$ is a right-angled triangle, we know that $|p_1 - p_2| = \sqrt{d_{\text{max}}^2 - |v_1 - p_2|^2}$. As there are two points on $l_1$ that lie at this distance from $p_2$, we have two possible locations for $p_1$; we want the one that lies in the direction of $v_4$, so we obtain $p_1 = p_2 + \sqrt{d_{\text{max}}^2 - |v_1 - p_2|^2} \cdot \frac{v_4 - p_2}{|v_4 - p_2|}$. If $p_1$ lies on $e_2$, we make the connection between $p_1$ and $v_1$; otherwise there is no point on $e_2$ that is within distance $d_{\text{max}}$ from $v_1$. Note that $p_1$ cannot lie beyond $v_4$, as we know the distance between $v_1$ and $v_4$ is larger than $d_{\text{max}}$.

Note that it is possible that the other point at distance $\sqrt{d_{\text{max}}^2 - |v_1 - p_2|^2}$ from $p_2$ is on $e_2$ when $p_1$ is not. However, we would never be attempting a vertex-edge connection in this case, as it would mean that $e_{22}$ is in fact the shorter edge.
Figure 6: Edge $e_{11} = \{v_1, c_1\}$ is shorter than $e_{22} = \{v_4, c_2\}$, so the latter is cut to $e_{22}' = \{v_4', c_2\}$, with $|e_{11}| = |e_{22}'|$. 

Figure 7: The different cases of the gap detection algorithm on the same pair of edges, but for increasing values of $d_{\text{max}}$, seen here as the length of connection $c$. In (a), the vertices of $c$ are on $e_{11}$ and $e_{22}'$, so this is the final connection. In (b), one vertex is outside of $e_{11}$, but the other is still on $e_{22}$: this case is resolved by the vertex-edge connection seen in Figure 8. In (c), the vertex is outside $e_{22}$, which is resolved in the same way as case (b).

Figure 8: A connection between $v_1$ and $e_{22}$. $p_2$ is the projection of $v_1$ onto the supporting line $l_1$ of $e_{22}$. As we want $|c| = d_{\text{max}}$, we can calculate the location of $p_1$ from the right-angled triangle $\langle v_1, p_1, p_2 \rangle$. 

13
5.2.3 Restrict by associated plane

After we have found two connections using the method described above, we need to further refine the result. Specifically, we do not want any connection to go “backwards” from either edge it is connected to. More formally, we do not want to connect to any point on the negative side of the associated plane of either edge; see Figure 9 for an example. This is because it would make no semantic sense: an agent cannot traverse a gap by stepping backwards. Furthermore, in non-coplanar 3D situations such as the one seen in Figure 10, it could mean that the resulting polygons could no longer be oriented consistently. If an endpoint of a connection is on the wrong side of the associated plane, we replace it with the intersection of the associated plane and the edge the point is on. If no intersection exists, we discard the gap filler.

5.3 Connecting cycles

For the connections made between distinct cycles of boundary edges, we employ a heuristic. This heuristic is based on three principles:

1. Any section of an edge can be connected to only one other edge, as otherwise singular edges are introduced.
2. A connection to an edge that is nearby when projected onto the ground plane is preferable to one that is far away.
3. It is important that chains of close edges are generally handled well, meaning they are connected by one continuous sequence of gap fillers.

As such, our heuristic has three main components. It first uses vertical planes along the bisectors of adjacent edges to limit every detected gap, as seen in Figure 11. Then, it builds an interval map of gaps for each edge, telling us which gaps are connected to each part of every boundary edge. Finally, for each of these segments, it selects the gap with the best heuristic score.

The restriction by planes is done in the same way as was done before with the associated plane: we test if a connection’s vertex is on the negative side of the plane, and if so, replace it by the intersection
of the plane with the edge. If no intersection exists, we discard the gap. The remaining gaps are used to build interval maps of fillers along each edge, as seen in Figure 12.

The interval maps associate a list of fillers with each part of an edge. For each interval, we heuristically choose one of the fillers using a score based on the squared projected distance between the edges in that interval. The score is calculated as follows. Assume that we want to know the score on the interval \([t_1, t_2]\) of edge \(e_1 = \{v_1, v_2\}\) for the filler connecting it to edge \(e_2 = \{v_3, v_4\}\), with the filler having points \(p_1\) and \(p_2\) on \(e_1\) and \(p_3\) and \(p_4\) on \(e_2\). We first calculate the points on \(e_1\) corresponding to the interval as \(i_1 = e_1(t_1)\) and \(i_2 = e_1(t_2)\). We then convert the interval on \(e_1\) to an interval on the filler, which we calculate as \([t_1', t_2']\), with \(t_1' = \frac{|i_1 - p_2|}{|p_3 - p_2|}\) and \(t_2' = \frac{|i_2 - p_1|}{|p_3 - p_1|}\). We obtain the corresponding part of the filler on \(e_2\) by taking \(i_3 = p_4 + t_1' \times (p_3 - p_4)\) and \(i_4 = p_4 + t_2' \times (p_3 - p_4)\). Note that \(p_3\) and \(p_4\) are reversed with respect to the usual counterclockwise order; this is because \(t = 0\) on one end of the filler corresponds with \(t = 1\) on the other. The score is then calculated as the sum of the squared shortest distances of \(i_3\) and \(i_4\) to the line segment \(\{i_1, i_2\}\) when projected onto the ground plane.

As an example, consider the interval map on edge \(e\) in Figure 12(a). The intervals \([0, 0.5]\) and \([0.75, 1]\) are associated with a single filler each, so no choice needs to be made. For the interval \([0.5, 0.75]\), we need to choose between the filler connecting \(e\) to \(a\) and the one connecting \(e\) to \(b\). We first convert the interval on the edge \(e\) to the interval of the filler that is connected to this part. For the filler with \(a\) this gives the interval \([0, 0.5]\), and for \(b\) it yields \([\frac{2}{3}, 1]\). We then consider the part of the filler corresponding to this interval, as shown in Figure 12(b). The heuristic score for \(a\) is then the sum of the squared distances of \(a_1\) and \(a_2\) to the line segment \(\{i_1, i_2\}\) when projected onto the ground plane. The score for \(b\) is obtained in the same way by using points \(b_1\) and \(b_2\).

This heuristic score is desirable for several reasons. First, it favours connecting edges that are close together when projected onto the ground plane. We prefer this over the closest edge in 3D because it prevents us from connecting “over” or “under” other components, as illustrated in Figure 13. Second, by taking the sum of distances of the endpoints rather than the shortest distance between the line segments on each edge, we favour segments that are close together over a large range, rather than just at one point, as seen in Figure 14.

After taking the gap filler with the lowest score for each interval of each edge, we need to find the mutual intervals between pairs of edges. We define a mutual interval as follows:

**Definition 11.** An interval \([a, b]\) on edge \(e_1\) with a desired connection to edge \(e_2\) is a mutual interval iff edge \(e_2\) has a desired connection to edge \(e_1\) on the interval \([1 - b, 1 - a]\).

The mutual intervals are easily found by taking each interval \([a, b]\) on each edge \(e_1\) with a desired connection to \(e_2\) and intersecting each interval on \(e_2\) that has a desired connection with \(e_1\) with \([1 - b, 1 - a]\). Each non-empty intersection gives us a mutual interval. The last step is then to take each mutual interval and connect the edges with two triangles on that interval.

As each boundary edge can be connected to at most \(m - 1\) other boundary edges, construction of all interval maps takes at most \(O(m^2 \log m)\) time. As each interval map has a number of intervals equal to the number of elements inserted, finding the mutual intervals takes \(O(m^2)\) time per boundary edge in the
Figure 12: The interval map for an edge $e$ with connections to edges $a$ and $b$. (a) shows the two gaps that were detected and the interval map associated with edge $e$. (b) illustrates the parts of the fillers that are used to calculate the heuristic score. As the score of the filler with edge $b$ is better in the interval $[0.5, 0.75]$, assuming edges $a$ and $b$ have no other fillers connected to them, we obtain the final result seen in (c).

Figure 13: A situation motivating our choice for using distance when projected onto the ground plane in the heuristic. The input is shown in (a). The distance between the lower two components is the smallest, so a heuristic using the 3D distance would make the choice seen in (b). This is undesirable, as it leaves the upper component unconnected. When projected onto the ground plane, the distance between the upper and lower components is much smaller, so the heuristic makes the choices seen in (c). The lower components are still connected indirectly via the upper component, making this choice preferable.

worst case, for a total worst case complexity of $O(m^3)$. This could be optimised by not searching through the entire interval map when finding each mutual interval, but as the number of intervals is typically much lower than $O(m)$, we have not implemented this. The calculation of the heuristic scores also takes $O(m^3)$ time in the worst case: we have $O(m)$ boundary edges with at most $O(m)$ intervals containing at most $O(m)$ elements each. This gives the entire heuristic a worst-case complexity of $O(m^3)$.

5.4 Filling holes

Holes are filled using the gap fillers detected between boundary edges in the same boundary edge cycle. For now, we only handle holes that have a boundary that is a simple polygon when projected onto the ground plane. When this is the case, we also project all the gap fillers between the cycle and itself onto the ground plane, giving us a set of 2D polygons. To prevent connecting the same interval on an edge twice, we disregard the parts of the gap fillers that connect to an interval already used by the heuristic. We then take the set-theoretic union of these polygons, yielding a new set of polygons where each polygon may contain holes. We intersect this set with the polygon representing the projected boundary edge cycle, to ensure we only take the parts inside this boundary into account. Finally, we triangulate each polygon in the resulting set and use those triangles to fill the hole. As we may be taking the union of up to $O(m^2)$ gap fillers, we obtain a worst-case time complexity of $O(m^2 \log m)$.

The resulting set of polygons may contain new vertices, either on the original boundary of the cycle, or on the boundary of the hole in the polygon obtained after taking the union of the gap fillers. As
Figure 14: A situation motivating our choice for using the sum of the squared distances of the vertices to the segment in the heuristic. The shortest distance between $e$ and $a$ is equal to that between $e$ and $b$, but we would prefer the connection with $a$. By using the sum of the squared distances of $v_1$ and $v_2$ to $a$ and $b$, we achieve this.

Figure 15: A hole before and after taking the union of the individual gaps. The blue polygon is the original input, the green polygons are the detected gaps; overlaps are represented by the dark areas. (a) shows the individual gaps detected between the edges in the boundary cycle. (b) shows the resulting union of these gaps, before triangulation.

these new vertices were created in the 2D space of the ground plane, we need to derive a $Z$-coordinate for them. For the vertices on the boundary of the cycle, this is straightforward: we just interpolate the $Z$-coordinates of the vertices of the edge it is on. For the other vertices, we take a weighted average of the $Z$-coordinate of all vertices on the boundary of the cycle, where each vertex has a weight equal to the reciprocal of the projected squared distance to the point of which we are calculating the $Z$-coordinate. This way we favour having a $Z$-coordinate similar to vertices that are close by when projected onto the ground plane.

5.5 Limitations

Our algorithm has several limitations. It does not currently handle holes which have a non-simple boundary when projected onto the ground plane, as our approach of taking the union of the projected gap fillers does not work in such cases. This could be overcome by using a different mapping to a two-dimensional plane, for instance by employing techniques from mesh parameterisation [13]. Any mapping algorithm would need to fulfill two requirements for our application: it needs to be a two-way mapping function (i.e., we need to be able to map new points created in the 2D space to the original 3D space), and the result should be a simple 2D polygon.

Another limitation is that the method ignores obstacles: it is possible that a gap filler will connect two edges through a wall. We try to avoid this by not connecting two edges that are both obstacle line segments, but this does not cover all cases. We cannot simply test the gap filler for intersection with the environment, as there are also cases where such intersections are of no concern, for instance when the intersection is only on a small part of the gap filler: we would not want to throw away the entire gap filler.
Figure 16: In this side view, if we would apply vertical clearance to the gap filler between $e_1$ and $e_2$ (represented by the dotted line), it would be rejected, as the original geometry (represented by the dashed line) is above it.

![Diagram](image1.png)

Figure 17: A case illustrating a limitation of our heuristic approach. In Figure (a), heuristic connections made from the red edge cannot be outside the planes denoted by the dashed lines. This limits any connections to the green region, meaning it cannot connect to the other component. The result of our algorithm is shown in Figure (b): the concave area is filled as a hole, and the two regions are connected by two gap fillers, but this leaves a triangular hole in the middle.

because of a one-millimetre-deep intersection. For a similar reason, we cannot simply apply the vertical clearance rule to our gap fillers: the gap filler may be (partially) underneath the original geometry, which could result in large parts of gap fillers being erroneously rejected, as illustrated in Figure 16.

A third limitation may occur with strongly concave sections of boundary edges that are not part of a hole. In these cases, restricting the gap fillers by the vertical planes along the bisectors of adjacent edges may cause the rejection of many of them when those bisectors intersect before the edge the gap filler is connected to, as seen in Figure 17. This concave part may be filled separately as a hole, but this still leaves a gap not covered by gap fillers.

The final limitation is that we do not backtrack on our heuristic choice when we do not find a mutual interval, which may cause parts of edges to be left unused when a connection would have been possible. We chose not to implement a strategy for backtracking when the option with the highest heuristic score is unavailable out of performance considerations, as it would give the algorithm exponential complexity in the worst case. These last two limitations may be overcome considering not single pairs of boundary edges at a time, but chains of boundary edges along which the connection between two cycles is continuous. This is similar to a part of the mesh repair method proposed by Barequet and Sharir [6].

6 Experiments

We evaluate our algorithm by looking at several metrics:

- **Number of components:** as our algorithm’s purpose is to connect parts of the walkable areas to each other, we consider the number of connected components in the environment before and after applying our algorithm. The difference between these two numbers can give an indication of how well the algorithm manages to connect previously unconnected regions.
- **Agent width:** we have also tried to make the resulting environment traversable by as wide an agent as possible. To evaluate this aspect, we attempt to find a path through the environment between 100,000 pairs of random points for varying sizes of agents. The success rate of finding paths can give some insight into the size of agents that would be able to comfortably navigate the environment.

- **Performance:** we are of course interested in the computational performance of our algorithm, so we measure the time it takes to apply our algorithm to each environment.

In addition to these metrics, we perform a manual visual inspection of the results for each environment, to look for possible defects or errors. We compare the results obtained with our algorithm to those obtained by Recast [10] and NEOGEN [11].

### 6.1 Test environments

Our test environments can be divided into two categories: artificial environments that we designed specifically to highlight certain aspects of the algorithm, and real-world environments that one might actually be interested in performing crowd simulation on. We provide a short description of each environment here.

#### 6.1.1 Artificial environments

**Concave boundary**
- Number of vertices: 25
- Number of triangles: 21
- Description: A boundary with a concave section, opposite a straight boundary.
- Expected result: This illustrates a limitation of our heuristic: when the boundaries of two components are poorly matched, we may discard some of the gap fillers when we make them chainable. The straight parts of the boundary should be connected to each other, while the concave section will be connected to itself, leaving a hole, even though the distance between the concave part and the other boundary is less than $d_{max}$.

**Grate**
- Number of vertices: 3740
- Number of triangles: 8316
- Description: A flat beam with a regular grid of holes, meant to simulate a metal grate used as a floor.
- Expected result: All the holes in the grid should be completely filled.
Holes
Number of vertices 399
Number of triangles 431
Description A horizontal plane with holes of different shapes and sizes.
Expected result As the value of $d_{\text{max}}$ increases, we expect each hole to shrink from the corners and edges, until they are filled completely.

Platform stairs
Number of vertices 72
Number of triangles 108
Description Floating platforms arranged in a grid, gradually increasing in height to create stairs.
Expected result Each platform is connected to the platforms in each of the cardinal directions, when they exist. Residual holes will be present at the intersections of the spaces between the platforms, as our method cannot resolve the four-way interaction at these points.

Platforms
Number of vertices 128
Number of triangles 192
Description Several floating platforms at differing distances and angles, representing different kinds of steps. Contains a mix of steps that are intersecting, exactly aligned and horizontally separated.
Expected result Each step should be connected to the one above and below.

Platforms in hole
Number of vertices 48
Number of triangles 80
Description Platforms floating inside a hole in a larger platform.
Expected result The platforms should be connected to each other and to the boundary of the hole. However, no matter how large we set $d_{\text{max}}$, we should never fill the entire free space of the hole, illustrating the limitations of our heuristic.
**Spiral hole**
Number of vertices 166
Number of triangles 166
Description A spiralling hole of gradually increasing width.
Expected result For a given value of $d_{\text{max}}$, the hole is either filled completely, or there comes a point in the spiral where the gap is too wide, at which point part of the hole will be left open.

**Step size**
Number of vertices 238
Number of triangles 308
Description Platforms of different shapes set at increasing distances from each other, illustrating how the connections change as the distance increases.
Expected result All but the last step should be connected, but depending on the geometry, the gap fillers may become less wide.

**Uneven steps**
Number of vertices 20
Number of triangles 10
Description The situation seen previously in Figure 13.
Expected result The lower components should be connected to the higher component rather than to each other, as the heuristic score considers the distance when projected onto the ground plane.

### 6.1.2 Real-world environments

**Amphitheatre**
Number of vertices 1263
Number of triangles 2553
Source 3D Warehouse
Description A small amphitheatre with a building. We removed the doors and had to fix some topology errors in a part of the seating.
Expected result The result should be one component, connecting each row of seats to the ramp in the centre and, depending on the size of $d_{\text{max}}$, to the rows above and below them.

1https://3dwarehouse.sketchup.com/model.html?id=519449fcf72233ba3fdb4378aaa154e
<table>
<thead>
<tr>
<th>Environment</th>
<th>Number of vertices</th>
<th>Number of triangles</th>
<th>Description</th>
<th>Expected result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>House</strong></td>
<td>446</td>
<td>820</td>
<td>The house environment from [12]. We include this as a reference test case, as it has no apparent holes or sources of gaps.</td>
<td>The lower part of the side of the ramp should be connected to the ground plane, allowing agents to step onto it from the side.</td>
</tr>
<tr>
<td><strong>Recast nav test</strong></td>
<td>884</td>
<td>1612</td>
<td>A test environment from Recast [10], containing many different cases of steps, holes and terrain.</td>
<td>The steps of the stairs should be connected to those above and below, while the corners of the three square holes should be filled, as they are too big to be filled completely. Some of the pillars may also be connected to surfaces close by.</td>
</tr>
<tr>
<td><strong>Skeleton building</strong></td>
<td>752</td>
<td>1292</td>
<td>The skeleton of a multi-storey building. We removed the foundation and the rebar which were modelled inside the concrete to reduce the triangle count, which was over 1 million in the original model.</td>
<td>The steps of the stairs should be connected to those above and below, while the corners of the holes through which the stairs reach the floor above should be filled. The result should be a single connected component.</td>
</tr>
<tr>
<td><strong>Stairs</strong></td>
<td>131</td>
<td>260</td>
<td>A set of stairs taken from a much larger environment, which unfortunately was too large to process.</td>
<td>Each step should be connected to the one above and below it, resulting in a single connected component.</td>
</tr>
</tbody>
</table>

2https://3dwarehouse.sketchup.com/model.html?id=u08d4d9be-bf3a-4ce9-9aa6-2626cb3036f
3https://3dwarehouse.sketchup.com/model.html?id=ub8ab230-3eb3-425e-9161-3db1b365a7e
Street
Number of vertices 248
Number of triangles 412
Description An intersection of two roads, with pavements and buildings on each side.
Expected result Each ridge of the pavement should be connected to the road seamlessly along its entire length. There are sixteen buildings with flat roofs, so the result should contain 33 components: one for the street, and two for each building (interior and roof).

Terrace house
Number of vertices 2400
Number of triangles 6155
Source 3D Warehouse
Description A two-storey house. We removed the doors and windows to allow unrestricted passage between the different areas, as well as the cars in the parking space and the toilet bowl and kitchen sink.
Expected result As this building includes stairs up to the second floor, we expect the result to contain only one main connected component, with possibly one more for the roof. This requires all the steps of the stairs to be connected, as well as some doorsteps separating the rooms.

Town house
Number of vertices 1476
Number of triangles 2676
Source 3D Warehouse
Description Another two-storey house on a road. We once again removed the doors and windows. We also removed the roof beams to decrease the time needed to apply the vertical clearance filter. Finally, we added a large rectangle underneath the entire scene, as the ground was not modelled. Note that the building lacks stairs to the second floor.
Expected result All the steps should be connected on either side; these include the stairs leading up the the porch, the tiles in the yard and the steps from the road on to the pavement. As there are no stairs leading to the second floor, the result should have two connected components: the ground floor, connected to the yard and road, and the second floor. Depending on the value of $\alpha_{max}$, several more components may be introduced by walkable surfaces on the roof.

4https://3dwarehouse.sketchup.com/model.html?id=ud50c65de-58f8-4018-95cd-8b524733e14c
5https://3dwarehouse.sketchup.com/model.html?id=ufceb8906-a322-426a-8cea-f0e60120447d
6.2 Implementation

We implemented our algorithm by expanding the filtering pipeline from [12]; as such, it was implemented in C++ and makes extensive use of the Computational Geometry Algorithms Library (CGAL) [3]. In particular, we use the exact computation provided by the CORE library, which is shipped with CGAL. This means we should never have to worry about errors introduced by the limited precision of floating-point numbers, but this comes at the cost of occasionally staggering memory consumption. To help alleviate this, we do not use exact computation for the building of intervals maps and for the calculation of scores. The interval maps are constructed with a fixed-point data type, having 32 integer and 480 fractional bits, which gives more than enough precision for this stage of the computation. The scores simply use 64-bit floating point values, as a high degree of accuracy is not particularly important in these calculations: the worst that can happen is that two scores which are very close in value have their order reversed.

Our measurements concerning the success rate of planning paths for agents of different widths were obtained in the ECM crowd simulation framework [16]. We first converted the results for our algorithm and Recast to a multi-layered environment using the PEEL library [8]. This required some manual adjustments in cases where this method generated connections between layers that spanned more than one edge, as this introduces a point obstacle in the ECM, which would influence our measurements.

6.3 Testing environment

Our tests were performed on a machine running Windows 7, with an Intel Core i7-2600K processor (4 cores, 8 threads, clocked at 4.3 GHz) and 16 GB of DDR3 RAM. Our program was compiled for 64-bit with Visual Studio 2015, using compiler version 19.00.24213.1. Our program uses CGAL version 4.9 and Boost version 1.63 for the R-tree [1] and the interval maps [2].

6.4 Pipeline settings

For all our tests, we use the following order of pipeline filters:

DeduplicationFilter 0.0001
DegeneracyFilter
RoundingFilter 0.001
DuplicationFilter
DegeneracyFilter
SlopeFilter 45
VerticalClearanceFilter 1.8
DuplicationFilter 0.0001
DegeneracyFilter
SimplificationFilter
RoundingFilter 0.0001
DuplicationFilter
DegeneracyFilter
SlopeFilter 45
ComponentCount
GapFilter 0.3
DuplicationFilter 0.001
DegeneracyFilter
ComponentCount

We first remove any duplicate vertices and degenerate triangles from the input, the round every vertex to the nearest millimetre. The rounding serves no other purpose than to increase the speed of subsequent filter steps. We then apply the restrictions on slope and vertical clearance, after which we simplify the remaining geometry through retriangulation. We then round again, to a resolution of 100 micrometres; we use a smaller number here in an attempt to prevent reintroduction of vertical overlap. This extra rounding step has proven vital to the performance of the gap filter: without it, our algorithm...
has to operate on the vertices created by the vertical clearance filter, which can have quite a complex representation in the exact number type we use, causing a large increase in memory usage. Indeed, without this step, the algorithm could not complete without needing more than the available 16 GB of memory for many of our larger environments.

After rounding, we need to filter for triangles exceeding the maximum slope again, as the rounding step may have caused some triangles to become vertical, which our implementation of the gap filter cannot handle (more specifically, it does not handle vertical boundary edges). We count the number of connected components present in the environment before and after the gap filter. Note that there are many invocations of the duplication- and degeneracy filter. These are necessary after the rounding-, vertical clearance- and gap filters, as each of these potentially introduces multiple vertices with the same coordinates. The value of $d_{\text{max}}$ is 0.3 for all our experiments, except when noted otherwise in the results.

6.5 Recast and NEOGEN

We used a slightly modified version of Recast [10], used in a comparative study of navigation meshes by Van Toll et al. [17]. This version allows the program to be run from the commandline, and facilitates exporting the results to a file. To make as fair a comparison as possible, we used the smallest possible voxel size that gave reasonable results, using a default value of 0.1 metres. In addition, we set the climb height to the value of $d_{\text{max}}$ for each environment, took 45° as the maximum slope, and used an agent height of 1.8 metres and a radius of 0. We also had to disable the three available filtering steps (“low hanging obstacles”, “ledge spans” and “walkable low height spans”) in most environments, as these caused no result to be given in small environments, and tended to remove steps from stairs when they only occupied a single voxel row. Our other settings mimicked those from [17]: we use watershed partitioning, the minimum region size is set to 0, the merged region size to 10000, maximum edge length to 500, maximum edge error to 1.3, vertices per polygon to 6, a detail sample distance of 6 and a detail sample error of 1.

For NEOGEN [11], the source code is not publicly available. However, with the permission of developer Ramon Oliva, we used the compiled version used the same comparative study mentioned above [17]. Unfortunately, this version does not allow us to tweak any of the parameter settings. We are therefore forced to use the settings from [17]: a 0.2 metre voxel size and a 60° maximum slope. The implementation does not take vertical clearance into account. We cannot use these settings for the other experiments, as our algorithm requires the vertical clearance rule to be enforced.

7 Results

In this section, we will discuss the results for each of the test environments. For all images, we render the walkable area in blue, gaps in green, and the input in transparent grey. We also use the same camera angle for the different results of the same environment, to make a fair comparison. Lastly, the images with a top-down perspective are rendered orthographically to prevent perspective distortions near the edges. Table 1 contains some statistics about the input to our algorithm and the steps it performs. Table 2 contains the time taken for each step of our algorithm on each environment.

7.1 Artificial environments

7.1.1 Concave boundary

The results for this environment can be seen in Figure 18. Our algorithm succeeds in connecting the two components to each other, but we are left with a residual hole, even though the distance between the apex of the concave part and the other component are within $d_{\text{max}}$ of each other. This is because the planes along the bisectors of adjacent edges intersect before reaching the other component, as previously illustrated for a simpler case in Figure 17.

One point of interest is that the result of our algorithm is asymmetric. This is because both edges of the concave part closest to the other component yielded a gap filler connecting to the same part of the other side. This gives identical scores to both options, at which point the choice made depends on the order in which we examine them. As part of one of the edges in the concave part is now already in use, it cannot be used to fill the hole, causing asymmetry in that part as well.
For Recast, we had to reduce the voxel size to 0.2 to obtain results. At this voxel size, the gap is still completely closed, but the generated navigation mesh is larger than and floats above the original input. NEOGEN cannot handle the horizontal gap, and returns only one connected component by design, such that the result contains only one of the two regions in the input.

7.1.2 Grate

The results for this environment can be seen in Figure 19. Our algorithm fills all the holes in the grate, as expected. Due to the large numbers of boundary edges located close to each other, this environment has the largest number of candidate edge pairs, and the highest gap detection time to match. The filling of the holes is very fast despite the large number present in the environment; this is because the result only connects existing vertices, without introducing any new ones, which is where most of the computation time in other environments is spent when filling holes.

Recast generates a result without any holes, but it is floating some distance above the original geometry. NEOGEN does not fill the holes: the original detail is retained, but the result has only narrow walkable areas.

7.1.3 Holes

The results for this environment can be seen in Figure 20. As expected, the holes are progressively more filled as $d_{\text{max}}$ increases, expanding from the places where the boundaries are closest together, until they are all completely filled when $d_{\text{max}} = 1$. The time needed to run our algorithm is dominated by the step in which holes are filled, but the time needed decreases as $d_{\text{max}}$ increases. This is because, as more holes are filled completely, fewer new vertices are introduced in the environment.

We also note that the time needed to detect the gaps increases with the value of $d_{\text{max}}$, which is to be expected, as this increases the number of candidate edge pairs. However, it is curious that the time needed to calculate the interval maps also increases, as there is not a single pair of edges that is connected heuristically in this environment. This indicates an inefficiency in our implementation of the
Figure 20: The results for the holes environment. The left column contains the results from our algorithm, while the right column contains the results obtained with Recast. From top to bottom, the values of $d_{\text{max}}$ and the voxel size are 0.1, 0.2, 0.3, 0.5 and 1.0 metres. The bottom-most image is the result obtained with NEOGEN.
building of the interval maps.

Recast also progressively fills the holes more as the voxel size increases, but the way in which this happens is less predictable, and the shape of the remaining holes is often only loosely related to the shape of the hole in the input. At a voxel size of 0.5, all the holes are filled, but there is a part of the input not covered by the generated mesh near the bottom. Increasing the voxel size to 1.0 ensures coverage of the entire area, but makes the result protrude over the edges of the input. NEOGEN does not handle holes, but it also does not simply return the input: the group of small holes close together has turned into a single hole in the result.

7.1.4 Platform stairs

The results for this environment can be seen in Figure 21. The result from our algorithm is exactly as expected, but there are residual holes, in this case because the heuristic score favours the directly adjacent platforms over those diagonally adjacent. Furthermore, the four-way interaction at each hole is something our algorithm cannot fully resolve in its current form, as we only look at single pairs of boundary edges. Around 90% of the running time is spent building the interval maps.

Recast gives only one square covering the entire input. There are no holes left, but the step-like nature of the environment has been reduced to a ramp. We had to set the voxel size to 0.3 metres for this environment, otherwise the rows of platforms would not be connected to each other. NEOGEN gives an empty output for this environment.

7.1.5 Platforms

The results for this environment can be seen in Figure 22. Our algorithm performs as expected, connecting each step to the one above and below. At the two bigger platforms, the first steps are also
connected towards the sides, indicating the possibility of stepping onto each set of stairs from the sides. In the left set of stairs, at the point where it makes a right angle, one step is connected to the one two steps above it, indicating that it is possible to cut that corner of the stairs. About two-thirds of the time is spent building the interval maps, and about one-third is spent calculating the heuristic scores.

Recast handles all the steps well, resulting in one large connected area, but there is a lot of overshoot near the curved stairs, and the shape is not followed particularly well there, because it is not aligned to the axes of the voxel grid. NEOGEN gives an empty output for this environment.

7.1.6 Platforms in hole
The results for this environment can be seen in Figure 23. The result produced by our algorithm for this environment is strongly asymmetric despite the symmetric input. As with the concave boundary environment, this is due to arbitrary tie-breaks when two gap fillers have the same heuristic score. The result also has some residual holes, which are due to the interaction between more than two boundaries, as was the case in the platform stairs environment. The result has two components, as the inside of the hole, below the floating platforms, is also partly walkable.

We used a voxel size of 0.2 metres for Recast in this experiment, as using the default size of 0.1 metres gave a result only covering the outer rim and the bottom of the hole, but not the platforms. With a voxel size of 0.2 metres, Recast completely covers the hole with the platforms, with no holes left over. However, the bottom of the hole is not included in the result, even though it is walkable. NEOGEN does not handle horizontal gaps or holes, and returns only a single component, giving just the outer rim of the environment.

7.1.7 Spiral hole
The results for this environment can be seen in Figure 24. The hole gets progressively more filled as $d_{\text{max}}$ increases. When $d_{\text{max}} = 0.2$, there is an obvious hole left over in the part that was filled (the same happens when $d_{\text{max}} = 0.1$, but it is less easily noticed). However, this is simply because in that part of the spiral, the distance between both sides is larger than 0.2 metres: the distance between both sides of the spiral is not strictly increasing as it spirals outwards, due to the approximation by line segments. Our algorithm is very slow on this environment, due to the large number of gap fillers that need to be merged when filling the hole.

Recast fills the entire hole from a voxel size of 0.2 metres, but the circular shape of the input is barely reflected in the result. For a voxel size of 0.1, we are left with three holes, which are not necessarily in predictable locations. The central hole is probably caused because the total area of the input in those voxels is too small for them to be tagged as walkable. NEOGEN does not handle holes and simply returns a closely matched approximation of the input.

7.1.8 Step size
The results for this environment can be seen in Figure 25. There is only one obvious defect in the solution our algorithm provides to this environment: there are residual holes between the shapes in the top row.
Figure 24: The results for the spiral hole environment. The left column contains the results from our algorithm, while the right column contains the results obtained with Recast. From top to bottom, the values of $d_{\text{max}}$ and the voxel size are 0.1, 0.2 and 0.3 metres. The bottom-most image is the result obtained with NEOGEN. Note that with the Recast results, we render the transparent part on top for clarity, but the navigation mesh is actually floating a small distance above the input.
These occur because the limitation by planes along the bisectors of adjacent edges; see Figure 26. Each edge can only connect to points in the space limited on each side by the bisector with the adjacent edge. The middle edge will prefer a connection with the edge directly opposite, as it is closer than the edge on either side. As the opposite side has the exact same situation, we are left with two holes despite the proximity of the edges. It is worth noting that this would not be a problem if there was a ground plane at most \( d_{\text{max}} \) below the shapes: in that case, each edge would connect straight down to the ground plane. However, a situation where this is not the case could occur in, for instance, a set of tall poles placed close together, on top of which agents can walk.

For this environment, most of the running time is spent filling the hole on the outer boundary of the top-left most region. The two squares below it are not considered one boundary, as they touch in only one point, so this is not considered a hole by our algorithm.

For Recast, we had to set the voxel size to 0.2 metres to obtain a result in which the steps were connected. All the gaps between regions are completely filled, but those that are connected are all reduced to rectangles, regardless of the shape of the input. The shape of the regions that are too far away to be connected are also a poor match for the shapes in the input. NEOGEN does not handle horizontal gaps and returns only the top-left component.

### 7.1.9 Uneven steps

The results for this environment can be seen in Figure 27. Our algorithm performs as expected on this environment, preferring the vertical connections despite the horizontal regions being closer together. Our algorithm takes about 155 milliseconds to compute this solution, which is not particularly fast, considering the small size of the input. It is especially slow when compared to the house environment, which, despite containing over ten times more boundary edges, takes roughly 170 milliseconds to process. The difference is in the building of the interval maps: the house environment does not contain a single edge which has more than one edge it can connect to, whereas the uneven steps environment has to build
the interval maps for the lower two regions.

Recast covers all three regions in the input, but the steps are reduced to two ramps. NEOGEN gives an empty output for this environment.

7.2 Real-world environments

7.2.1 Amphitheatre

The results for this environment can be seen in Figure 28. For this environment we had to set $d_{max}$ to 0.6, as otherwise not all the rows of seats would be connected. The results are mostly as expected. The rows of seats are connected to each other and to the ramp running down the middle. The base of each column is also connected to the ground. There are two cases of edges being connected through a wall, both inside the roof of the building (one case can be seen in the figure between the two pillars). There are four main components in the result: one containing the floor and the seating, one on the upper rim of the seats, and one on each side of the roof; the remaining six components are all minor parts of the (inside of the) roof and walls. Most of the computation time is spent on the building of the interval maps by far: over 95% of the time is spent on this step.

Recast has no issues with connections through walls, but the shape of the result is a very poor match for the shape of the input. There are some triangles in the result with a very steep slope despite the flatness of the input in the same area (such as the one seen running down the centre of the stage). NEOGEN puts the floor and the seating into the same component, as is desired (however, due to a technical limitation, this is not drawn as such: the connections between layers are not included when exporting the result). However, note that the method fails to identify the walls and pillars as obstacles.

7.2.2 House

The results for this environment can be seen in Figure 29. Our algorithm does almost nothing in this environment, which is as we expected. The only connections made are those cutting off corners in the walls; the expected connection at the bottom of the ramp is missing, as those edges are part of a hole with a non-simple boundary when projected onto the ground plane, and are therefore skipped.

Recast has some problems with this environment: there is a hole in one of the corners of the ramp, where the geometry only touches in two points. Furthermore, the step onto the side of the ramp is much higher than the 0.3 metres we allowed in the settings. This is likely because the step size is calculated on top of the voxel representation, which has already raised the floor level considerably. There are also two parts of the roof marked as walkable, despite the high slope of the input geometry. NEOGEN does not include the ground plane in the result, but somehow puts the top of the wall and the ramp in one component.

7.2.3 Recast nav test

The results for this environment can be seen in Figure 30. Any amount of rounding after applying the vertical clearance filter reintroduced vertical overlap in almost all the steps, so we could not round
Figure 28: The results for the amphitheatre environment.
before applying our algorithm. This meant our implementation was unable to find a solution within the available amount of memory.

For Recast, we enabled the three filtering steps, as this gave better results. Note that in the left set of stairs, one of the vertices in the navigation mesh is placed on the floor below the stairs, rather than on it. NEOGEN once again places two seemingly unrelated areas into the same component, and ignores many of the obstacles on the ground floor.

### 7.2.4 Skeleton building

The results for this environment can be seen in Figure 31. Our algorithm connects all the steps of the stairs exactly as expected. The result contains many extra components: small areas inside the pillars which have enough vertical clearance, but also some areas inside the pillars that are closely below each floor. It appears that a bug in the implementation of the vertical clearance filter prevents these parts from being removed. Our algorithm is relatively fast on this environment, requiring only 7.4 seconds to compute a solution. Most of this time is spent detecting the gaps and building the interval maps.

For Recast, we enabled the three filtering steps, as this gave better results. There is still one instance of a vertex in the navigation mesh being given the height of a different floor. All the stairs have also been turned into ramps, losing any information about step size and distance. For reasons unknown to us, NEOGEN fails to include the stairs and the bottom floor in the result. The fact that the other four floors are all in the result indicates that NEOGEN considers them to be part of the same component, but the output does not reflect this.

### 7.2.5 Stairs

The results for this environment can be seen in Figure 32. On this environment, our algorithm gives exactly the result we expected: each step is connected to the one above and below, making the whole surface one connected component. There is one small extra component on the inside of structure supporting the upper part, which has enough vertical clearance to not be rejected. At 704 milliseconds, our
Figure 31: The results for the skeleton building environment.
algorithm is relatively fast on this environment, with most time once again being spent on the building of the interval maps.

For Recast, we enabled the filtering steps, as this gave better results. Recast successfully connects the lower and upper platforms via the stairs, but the stairs have been replaced by ramps. The bottom of the ramp also starts earlier on the left side than it does on the right. NEOGEN is unable to process the stairs correctly, giving only the lower platform as the result. Note that it fails to identify the column as an obstacle, and that it disregards the vertical clearance with the underside of the stairs.

7.2.6 Street

The results for this environment can be seen in Figure 33. Our algorithm neatly connects the pavement with the road along the whole length of the gap between them. The computation time is once again completely dominated by the building of the interval maps, with about 99% of the time being spent on that step.

Recast offers a solution quite similar to our own, but the distinction between the pavement and the street has been lost. We did, however, have to increase the voxel size to 0.2 metres, as the number of voxels was too great when subdividing the environment voxels of 0.1 metres in size. NEOGEN gives an empty output for this environment.

7.2.7 Terrace house

The results for this environment can be seen in Figure 34. The result given by our algorithm has two main components: the interior of the house, containing both the ground- and first floor, and the roof. There are many extra components, most of them on the insides of walls. There are some cases of parts of the floor being connected through the wall, and in some cases to the inside of the wall. The building of the interval maps accounts for about two-thirds of the 58 seconds it takes for our algorithm to produce this solution, with another 24% being spent on the calculation of the heuristic scores, indicating that there are many edges with multiple gap fillers to choose from.

Recast has no problem separating the rooms from the inside of the walls, but loses all information about steps in the environment: the stairs become a ramp, and the thresholds between the rooms are lost. There is also a walkable area on the railing of the stairs which is much wider than the input geometry.

NEOGEN was unable to compute a solution within several hours, at which point we terminated the application. The program most likely got stuck in an infinite loop.

7.2.8 Town house

The results for this environment can be seen in Figure 35. Our algorithm succeeds in connecting all the steps to the floor or each other, resulting in three main connected components: outside and the ground floor, the first floor, and the roof. The first floor is erroneously connected to some of the lower parts of the roof through the walls. There are many extra components, mostly in the walls and pillars around the yard. Despite the larger number of boundary edges in this environment, it is processed much faster than the terrace house environment, indicating that this one has fewer gap fillers per edge on average.
Figure 33: The results for the street environment. NEOGEN gave no output here.
Figure 34: The results for the terrace house environment. NEOGEN gave no output here.
Figure 35: The results for the town house environment. NEOGEN gave no output here.
<table>
<thead>
<tr>
<th>Environment</th>
<th>Triangles</th>
<th>Boundary edges</th>
<th>Cycles</th>
<th>Pairs</th>
<th>Candidates</th>
<th>Gap fillers</th>
<th>Heuristic Holes</th>
<th>Components in</th>
<th>Components out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concave boundary</td>
<td>21</td>
<td>25</td>
<td>2</td>
<td>516</td>
<td>240</td>
<td>136</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Grate</td>
<td>1262</td>
<td>844</td>
<td>211</td>
<td>357012</td>
<td>22854</td>
<td>5030</td>
<td>0</td>
<td>210</td>
<td>1</td>
</tr>
<tr>
<td>Holes ($d_{max} = 0.1$)</td>
<td>341</td>
<td>309</td>
<td>18</td>
<td>56240</td>
<td>2498</td>
<td>984</td>
<td>0</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>Holes ($d_{max} = 0.2$)</td>
<td>341</td>
<td>309</td>
<td>18</td>
<td>56240</td>
<td>5037</td>
<td>1985</td>
<td>0</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>Holes ($d_{max} = 0.3$)</td>
<td>341</td>
<td>309</td>
<td>18</td>
<td>56240</td>
<td>7303</td>
<td>2853</td>
<td>0</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>Holes ($d_{max} = 0.5$)</td>
<td>341</td>
<td>309</td>
<td>18</td>
<td>56240</td>
<td>12398</td>
<td>5114</td>
<td>0</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>Holes ($d_{max} = 1.0$)</td>
<td>341</td>
<td>309</td>
<td>18</td>
<td>56240</td>
<td>18013</td>
<td>7455</td>
<td>0</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>Platform stairs</td>
<td>18</td>
<td>36</td>
<td>9</td>
<td>684</td>
<td>262</td>
<td>44</td>
<td>28</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>Platforms</td>
<td>42</td>
<td>80</td>
<td>19</td>
<td>3296</td>
<td>467</td>
<td>64</td>
<td>39</td>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td>Platforms in hole</td>
<td>34</td>
<td>36</td>
<td>9</td>
<td>684</td>
<td>114</td>
<td>32</td>
<td>21</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Spiral hole ($d_{max} = 0.1$)</td>
<td>166</td>
<td>166</td>
<td>2</td>
<td>23102</td>
<td>1828</td>
<td>320</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Spiral hole ($d_{max} = 0.2$)</td>
<td>166</td>
<td>166</td>
<td>2</td>
<td>23102</td>
<td>3651</td>
<td>738</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Spiral hole ($d_{max} = 0.3$)</td>
<td>166</td>
<td>166</td>
<td>2</td>
<td>23102</td>
<td>5280</td>
<td>1154</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Step size</td>
<td>306</td>
<td>120</td>
<td>19</td>
<td>7596</td>
<td>518</td>
<td>74</td>
<td>24</td>
<td>1</td>
<td>19</td>
</tr>
<tr>
<td>Uneven steps</td>
<td>6</td>
<td>12</td>
<td>3</td>
<td>84</td>
<td>62</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 1: Statistics about the input to our algorithm and the steps performed. The table contains the number of triangles, boundary edges and boundary edge cycles in the input to our algorithm. It also lists the theoretical maximum number of edge pairs that need to be considered (i.e. when $d_{max} = \infty$), the number of candidate pairs retrieved from the R-tree, the number of gap fillers created (and thus the number of edge pairs between which a gap was detected), the number of edge pairs connected heuristically and the number of holes filled. In addition, it lists the number of connected components in the environment before and after the application of our algorithm.

Recast once again has no problems with connections through walls, but loses all the steps again. The front part of the lower roof are at inconsistent heights due to the protruding beams below it. The ramps over the two stairs are also significantly less wide than the passages in the original geometry. NEOGEN gives an empty output for this environment.

### 7.3 Pathfinding results

The results for our pathfinding experiments with agents of different widths can be seen in Table 3. We only performed these test in environments of sufficient size. Unfortunately, we cannot include results from NEOGEN: it gave no results for most of the environments, and its output only contains the cells of the navigation mesh, but no information about which cells are connected to each other, and where. This means that even in the amphitheatre environment, on which NEOGEN gave a reasonable output, we cannot compute an ECM of the result, as we have no information about which parts of the mesh connect to which.

We first note that the values are nearly identical between our method and Recast, indicating that the width of the corridors are mostly the same for each method. We also note that the values tend to slightly increase for small agent sizes when their radius increases. This is because, as the agent radius increases, there are fewer components with free positions, thus decreasing the number of free position pairs for which a path cannot exist. As such, we do not see such an increase of the values in the street environment, as this environment contains no small components.

In the amphitheatre environment, we would expect to see a drop in the numbers when the agent size
<table>
<thead>
<tr>
<th>Environment</th>
<th>Cycle detection</th>
<th>Gap detection</th>
<th>Interval maps</th>
<th>Heuristic scores</th>
<th>Mutual intervals</th>
<th>Filling holes</th>
<th>Retrigration</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concave boundary</td>
<td>1.4</td>
<td>41</td>
<td>9163</td>
<td>11</td>
<td>13</td>
<td>728</td>
<td>7.1</td>
<td>9966</td>
</tr>
<tr>
<td>Grate</td>
<td>65</td>
<td>9589</td>
<td>279</td>
<td>&lt;0.1</td>
<td>1.1</td>
<td>61852</td>
<td>33</td>
<td>10254</td>
</tr>
<tr>
<td>Holes ($d_{max} = 0.1$)</td>
<td>51</td>
<td>599</td>
<td>0.8</td>
<td>&lt;0.1</td>
<td>1.1</td>
<td>61852</td>
<td>33</td>
<td>62538</td>
</tr>
<tr>
<td>Holes ($d_{max} = 0.2$)</td>
<td>52</td>
<td>909</td>
<td>13</td>
<td>&lt;0.1</td>
<td>1.3</td>
<td>46747</td>
<td>15</td>
<td>47739</td>
</tr>
<tr>
<td>Holes ($d_{max} = 0.3$)</td>
<td>57</td>
<td>958</td>
<td>800</td>
<td>&lt;0.1</td>
<td>1.2</td>
<td>48630</td>
<td>11</td>
<td>50459</td>
</tr>
<tr>
<td>Holes ($d_{max} = 0.5$)</td>
<td>54</td>
<td>999</td>
<td>525</td>
<td>&lt;0.1</td>
<td>1.1</td>
<td>46960</td>
<td>7.3</td>
<td>48547</td>
</tr>
<tr>
<td>Holes ($d_{max} = 1.0$)</td>
<td>52</td>
<td>1397</td>
<td>1107</td>
<td>&lt;0.1</td>
<td>1.3</td>
<td>7906</td>
<td>1.1</td>
<td>10466</td>
</tr>
<tr>
<td>Platform stairs</td>
<td>1.5</td>
<td>78</td>
<td>2019</td>
<td>50</td>
<td>65</td>
<td>1.0</td>
<td>14</td>
<td>2230</td>
</tr>
<tr>
<td>Platforms</td>
<td>7.0</td>
<td>259</td>
<td>10759</td>
<td>5032</td>
<td>242</td>
<td>1.0</td>
<td>47</td>
<td>16348</td>
</tr>
<tr>
<td>Platforms in hole</td>
<td>1.8</td>
<td>43</td>
<td>439</td>
<td>68</td>
<td>68</td>
<td>88</td>
<td>16</td>
<td>727</td>
</tr>
<tr>
<td>Spiral hole ($d_{max} = 0.1$)</td>
<td>5.8</td>
<td>2193</td>
<td>&lt;0.1</td>
<td>&lt;0.1</td>
<td>1.0</td>
<td>237776</td>
<td>35</td>
<td>240012</td>
</tr>
<tr>
<td>Spiral hole ($d_{max} = 0.2$)</td>
<td>5.8</td>
<td>4049</td>
<td>&lt;0.1</td>
<td>&lt;0.1</td>
<td>1.0</td>
<td>540492</td>
<td>19</td>
<td>544568</td>
</tr>
<tr>
<td>Spiral hole ($d_{max} = 0.3$)</td>
<td>5.5</td>
<td>3139</td>
<td>&lt;0.1</td>
<td>&lt;0.1</td>
<td>1.0</td>
<td>553330</td>
<td>1.0</td>
<td>556478</td>
</tr>
<tr>
<td>Step size</td>
<td>19</td>
<td>5026</td>
<td>2622</td>
<td>47</td>
<td>66</td>
<td>75404</td>
<td>366</td>
<td>83554</td>
</tr>
<tr>
<td>Uneven steps</td>
<td>1.2</td>
<td>14</td>
<td>126</td>
<td>5.0</td>
<td>6.0</td>
<td>0.8</td>
<td>0.9</td>
<td>155</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Environment</th>
<th>Amphitheatre</th>
<th>House</th>
<th>Skeleton building</th>
<th>Stairs</th>
<th>Street</th>
<th>Terrace house</th>
<th>Town house</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concave boundary</td>
<td>112</td>
<td>31</td>
<td>43</td>
<td>7.0</td>
<td>15</td>
<td>99</td>
<td>108</td>
</tr>
<tr>
<td>Grate</td>
<td>3799</td>
<td>109</td>
<td>1459</td>
<td>114</td>
<td>215</td>
<td>5908</td>
<td>6213</td>
</tr>
<tr>
<td>Holes ($d_{max} = 0.1$)</td>
<td>160515</td>
<td>&lt;0.1</td>
<td>5355</td>
<td>513</td>
<td>4254</td>
<td>37344</td>
<td>6879</td>
</tr>
<tr>
<td>Holes ($d_{max} = 0.2$)</td>
<td>3269</td>
<td>&lt;0.1</td>
<td>188</td>
<td>28</td>
<td>82</td>
<td>13678</td>
<td>1434</td>
</tr>
<tr>
<td>Holes ($d_{max} = 0.3$)</td>
<td>899</td>
<td>&lt;0.1</td>
<td>247</td>
<td>35</td>
<td>118</td>
<td>313</td>
<td>498</td>
</tr>
<tr>
<td>Step size</td>
<td>148</td>
<td>21</td>
<td>74</td>
<td>36</td>
<td>6.5</td>
<td>481</td>
<td>74</td>
</tr>
<tr>
<td>Uneven steps</td>
<td>137</td>
<td>6.5</td>
<td>7405</td>
<td>4.6</td>
<td>2.5</td>
<td>55</td>
<td>76</td>
</tr>
</tbody>
</table>

Table 2: The performance measurements of our algorithm for each of the steps that is performed, in milliseconds. The measurements were obtained from a single run of our program.

<table>
<thead>
<tr>
<th>Environment</th>
<th>Amphitheatre</th>
<th>Skeleton building</th>
<th>Street</th>
<th>Terrace house</th>
<th>Town house</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle detection</td>
<td>0.636</td>
<td>0.987</td>
<td>0.296</td>
<td>0.772</td>
<td>0.771</td>
</tr>
<tr>
<td>Gap detection</td>
<td>0.636</td>
<td>0.987</td>
<td>0.294</td>
<td>0.755</td>
<td>0.776</td>
</tr>
<tr>
<td>Interval maps</td>
<td>0.635</td>
<td>0.987</td>
<td>0.294</td>
<td>0.785</td>
<td>0.791</td>
</tr>
<tr>
<td>Heuristic scores</td>
<td>0.674</td>
<td>1.000</td>
<td>0.296</td>
<td>0.789</td>
<td>0.795</td>
</tr>
<tr>
<td>Mutual intervals</td>
<td>0.675</td>
<td>1.000</td>
<td>0.296</td>
<td>0.790</td>
<td>0.795</td>
</tr>
<tr>
<td>Filling holes</td>
<td>0.689</td>
<td>1.000</td>
<td>0.296</td>
<td>0.731</td>
<td>0.738</td>
</tr>
<tr>
<td>Retrigration</td>
<td>0.688</td>
<td>1.000</td>
<td>0.296</td>
<td>0.755</td>
<td>0.753</td>
</tr>
</tbody>
</table>

Table 3: Fraction of 100,000 pairs of random free positions between which a path can be found for a variety of agent radii. There are no results in the terrace house environment for agents with a two-metre radius, as there are no free positions available for an agent that size.
Figure 36: The results given by Recast for a voxel size of 0.1 and 0.3 metres, respectively. Note that with a voxel size of 0.3 metres, the navigation mesh floats significantly higher above the surface, and that the posts and the wall have been connected in most places, despite having a height difference of more than the allowed 0.3 metres step height.

exceeds that of the door opening of the building. However, as the door opening is over two metres wide, but the building itself is only 3.5 metres wide, this falls right in between our two largest agent sizes. For an agent with a radius of two metres, the inside of the building no longer contains any free positions, so the numbers stay roughly the same.

In the skeleton building environment, we see that up to an agent radius of one metre, we can virtually always find a path between two free positions. Increasing the radius to two metres, the stairs are no longer wide enough to support our agents and we can only find paths to positions on the same floor. As there are five floors of equal size, we expect this to be about 20% of the pairs of free positions, which is reflected in our measurements.

The wide-open nature of the street environment means that we can always find a path between free positions in the same component. In the terrace house environment, however, we see a steep drop when the agent radius is increased from 0.3 to 0.5 metres: the stairs connecting the two floors are no longer traversable for an agent of this size. We see a similar, but smaller drop in the town house environment for the same radii, when the inside of the building is no longer reachable from the outside. This drop is much smaller because the surface area of the ground outside is much larger than that of the inside of the building.

7.4 Recast parameters

One thing we noticed is that obtaining the best results with Recast can require some tweaking of the parameters. We might be tempted to set the voxel size to 0.3 metres, thereby guaranteeing that holes and horizontal gaps are filled up to that size. In many cases, however, this results in a navigation mesh of much lower quality, as illustrated in Figure 36. The increased voxel size causes the lower and higher parts of the wall to be connected to each other, despite having a height difference of more than 0.3 metres. Some parts are also erroneously rejected due to low vertical clearance, as this is calculated on the voxel grid, and as such may underestimate the vertical clearance by up to twice the voxel height. This is especially a problem in environments such as the skeleton building, where the vertical clearance on the stairs is very close to the necessary minimum, as it causes the floors to be disconnected in the result.

Setting the voxel size as small as possible is also not an option, as this may cause gaps that we want to be filled to remain open, and on large environments the voxel count becomes too large for the program to handle. This illustrates that the filling of holes and horizontal gaps is more a convenient side-effect of the voxelisation than it is a feature of the Recast method. In contrast, our method was able to use the same sequence of filter steps for all the environments in our tests, barring the issue with memory limitations observed in the Recast nav test environment.
7.5 Conclusion

Overall, our method is more accurate than Recast and NEOGEN, and requires no extensive tweaking of parameters to achieve good results. The results obtained with our method are also semantically annotated, and retain the exact representation of the walkable area, making them more useful to path planning algorithms and animation systems. In some of the more complex scenes (i.e. the terrace house and town house scenes), our method will occasionally connect two boundary edges through walls, which is a problem that will need to be addressed. Issues with memory consumption currently also limit the pipeline as a whole to relatively simple environments: it would not work for a complex game world, or a detailed model of a large office building. This contrasts with the voxel-based methods such as Recast and NEOGEN, which are limited more by the physical dimensions of the environment than by the amount of geometry contained within.

We feel that if the existing issues with the handling of obstacles and the occasional presence of residual holes are addressed, our algorithm has the potential to be more effective than both Recast and NEOGEN for the generation of walkable environments. A challenge will be to increase the performance of our implementation enough to be useful in practice, regardless of the complexity of the input environment.

8 Conclusion

In this thesis we have presented a method for finding and filling gaps in walkable environments as an extension of the filtering pipeline developed by Polak [12]. Our algorithm does not modify the input and makes a clear distinction between the walkable areas and the gaps that can be traversed. Our method works by finding pairs of boundary edges that are possibly within a user-specified gap size \( d_{\text{max}} \) of each other, splitting them at their closest points, and finding the point on each half at which the edges are exactly \( d_{\text{max}} \) apart. When two such points are found, we generate a gap filler, which is a polygon containing the area between the points. For edges belonging to different cycles of boundary edges, we calculate an interval map of the edges it can connect to, and heuristically choose a destination for each interval based on the distance when projected onto the ground plane. For edges belonging to the same cycle of boundary edges, and when this cycle forms a simple polygon when projected onto the ground plane, we project all the gap fillers between edges in this cycle to the ground plane, take their set-theoretic union, and lift the vertices back to 3D space.

We compared our extended pipeline to two existing voxel-based methods of navigation mesh generation, Recast [10] and NEOGEN [11]. Our algorithm gives a more accurate representation of the walkable environment and the way gaps can be traversed, and the result is more useful to path planning and animation systems. However, our method still has some unsolved shortcomings that will need to be addressed in the future, which we discuss below.

8.1 Future work

Our algorithm has several shortcomings that need to be solved for it to work automatically in all realistic situations. The gap fillers do not take into account obstacles in the environment, which may cause edges to be connected through walls, or result in a gap filler with low vertical clearance. This is a hard problem to solve, as not all obstructions should necessarily have an effect. For instance, we would not mind it if there is a small obstruction in the gap filler connecting two steps when that obstruction is caused by the rejected geometry that connected the two steps in the input (as seen in Figure 16).

Furthermore, our method currently ignores holes when their projected boundary is not a simple polygon. Solving this would require a modification of the algorithm such that projection is not necessary, or use a different projection method. The field of mesh parameterisation [13] may offer mapping algorithms from 3D to 2D that would suit our applications. Such an algorithm would have to also be able to map newly created vertices from 2D back to 3D.

In addition, our algorithm will in some cases leave holes in the output when the boundary of two cycles do not follow each other closely, even though the distance is less than \( d_{\text{max}} \). This could be resolved by considering not single pairs of boundary edges, but chains of boundary edges which form a continuous connection. Residual holes may also appear when there is an interaction between more than two cycles of boundary edges. These holes could be explicitly detected and filled in a post-processing step to our
algorithm. For both cases, we refer to Barequet and Sharir [6], who take a similar approach in the context of mesh repair.

Lastly, our implementation is slow, and, more importantly, may require exorbitant amounts of memory. This is due to our use of the exact computation offered by CGAL. We could greatly improve performance and memory consumption with a robust implementation using finite precision, but this is often challenging when complex geometrical algorithms are involved.

Acknowledgements

I would like to thank both my supervisors for their patience and the many valuable points of discussion and feedback. In particular I would like to thank Arne for spending many hours of his time helping me fix issues in both external applications and my own. I am also grateful to Mihai Polak for the many useful discussions and ideas during the development of the algorithm, and his continued work on the existing pipeline during the first half of my project.

In addition, I would like to express my thanks to Wouter van Toll for making available to me the modified version of Recast I used in my experiments, and his insights into the parameter settings. Lastly I wish to thank Ramon Oliva for giving me permission to use NEOGEN.

References

