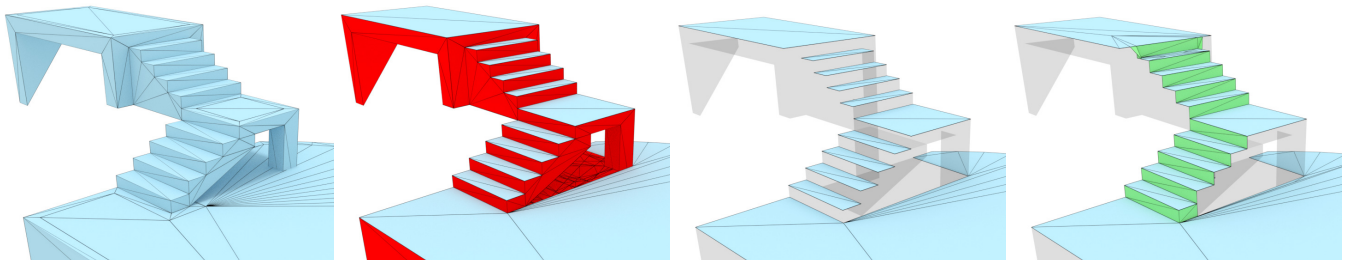
Annotating traversable gaps in walkable environments

Jordi L. Vermeulen Arne Hillebrand Roland Geraerts



(a) (b) (c) (d)

Fig. 1. The annotation process of traversable gaps in a 3D environment. (a) An unprocessed 3D environment. (b) The red areas are not walkable because the slope is too high or the vertical clearance to obstacles is too low. (c) The blue areas denote the walkable areas in the environment; note that there are gaps between the steps. (d) Our algorithm identifies and fills these gaps with the green polygons such that agents can continuously navigate the environment.

***Abstract*— Autonomous agents typically need a navigation mesh of a 3D virtual environment to allow efficient path planning. This mesh needs as input a continuous representation of the walkable areas. However, the *walkable environment* (WE), i.e. the parts of the 3D environment that an agent can walk on, may contain gaps. These may be due to the filtering steps performed to compute the WE, because of modelling errors in the 3D model, or simply be part of the geometry of the environment.**

We provide an algorithm that identifies and fills these gaps. We detect gaps, up to a given distance, between pairs of boundary edges of the walkable environment, and fill them with polygons. We employ a heuristic for choosing which pairs of edges should be connected.

We compare our algorithm to Recast [10], a voxel-based method for navigation mesh generation. 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 good results. However, our method is currently slower than Recast, and requires more memory.

I. INTRODUCTION

A big challenge for mobile autonomous agents (such as robots or virtual humans) is to navigate complex 3D environments. The 3D environment is typically represented as the free configuration space [9] of a cylinder of given height and radius, with an extra restriction on the slope of the surface. This free configuration space is computed as a *walkable environment* (WE) [5], [12], [14], from which a navigation mesh is generated for efficient path planning [16].

To allow accurate navigation, the WE needs to present a continuous representation of the 3D environment: agents can only navigate between connected parts of the WE. However, if we simply extract the WE, small steps or obstacles may cause traversable regions to become disconnected, as illustrated in Fig. 1. In addition, gaps may also be present in the environment due to modelling errors, or as part of the input geometry (e.g. metal grate floorboards). Such gaps need to be dealt with to allow realistic navigation of the environment.

Contribution. We present a novel method that identifies and fills gaps in a walkable environment so that an agent can continuously navigate the environment. We only annotate *traversable gaps* – that is, gaps that the agent can realistically navigate across. These gaps are annotated, allowing them to be taken into account during path- and motion planning (e.g. a foot placement system can avoid the gaps). This method preserves the full detail of the walkable environment, facilitating the generation of high-quality navigation meshes.

The rest of this paper is structured as follows. In Section II, we discuss related work in navigation mesh generation and gap filling techniques in polygonal environments. In Section III, we give some definitions and notational conventions. Section IV defines our problem more formally. In Section V, we describe our algorithm for detecting and filling gaps in walkable environments. Section VI describes the experiments we performed; the results are described and interpreted in Section VII. In Section VIII, we conclude that our method gives more detailed and semantically useful results than existing methods, and we discuss current limitations of our method.

II. 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 that determine the walkable area (and/or build a navigation mesh) deal with the presence of gaps in the input. We also consider the problem from the perspective of mesh repair.

The authors are with the Department of Information and Computing Sciences, Utrecht University, Utrecht, the Netherlands
{J.L.Vermeulen; A.Hillebrand; R.J.Geraerts}@uu.nl

A. 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.

Volumetric approaches work on a voxelised representation of the environment. The main advantage of such methods is that they circumvent many complicated and sometimes ambiguous cases involving degeneracies, overlaps, intersections and gaps. These cases often occur within a single voxel, making them simple to solve. A disadvantage is that these methods tend to be sensitive to the size of the voxels: small voxels will cause increased computation times and memory consumption on environments with large dimensions, whereas large voxels may result in a lack of detail. The result may also depend on how the input aligns with the grid. Two well-known volumetric methods are Recast [10] and NEOGEN [11]. Li and Huang [7] apply the morphological closing operator to a voxelised environment to close gaps.

Surface-based methods work directly on the polygonal representation of the environment, and thus have no problems with limited resolution. The downside is that these methods need to deal with the full complexity and possible ambiguity of each environment. Methods such as TopoPlan [6] assume the input contains no intersections between triangles, while others, like the method by Polak [12], incorporate processing steps to resolve such errors. Lopez et al. [8] solve the navigation of gaps during simulation, but this would make the simulation of large numbers of agents slow. We choose a surface-based approach for our algorithm, as we want to retain the full detail of the input.

B. Mesh repair

Mesh repair is concerned with the detection and reparation of errors in polygonal meshes. 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. This is undesirable for our application, as we wish to maintain a clear distinction between the walkable surfaces and traversable gaps, to facilitate accurate path planning. Moreover, the goal will often be to create a *closed mesh*, having a clearly defined inside and outside, which is not a property we want. A good overview of mesh repair techniques, including common sources of errors, is given by Attene et al. [3]. These techniques offer good inspiration, but are too general for our purposes.

III. PRELIMINARIES

A. Notation

We work in 3D Euclidean space, with points $\mathbf{p} = (p_x, p_y, p_z) \in \mathbb{R}^3$. We denote the magnitude of a vector $\mathbf{v}$ as $|\mathbf{v}|$. We take the XY-plane as the ground plane (the plane perpendicular to the gravity vector), with the positive Z-axis pointing up. We work on polygonal meshes containing only triangles, where each triangle T is a list of vertices $\langle \mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3} \rangle$, which we assume to be given in counterclockwise order. Each vertex corresponds to a point in $\mathbb{R}^3$. The normal of a triangle is the unit vector given by $\frac{(\mathbf{v_2}-\mathbf{v_1})\times(\mathbf{v_3}-\mathbf{v_1})}{|(\mathbf{v_2}-\mathbf{v_1})\times(\mathbf{v_3}-\mathbf{v_1})|}$. We define the slope of a triangle to be the angle between its normal and that of the ground plane.

B. Walkable environments

We take the definition of a walkable environment from Van Toll et al. [14], which is compatible with the definition given by Hillebrand et al. [5]. Specifically, the *walkable environment* (WE) of a 3D environment is the set of triangles that can be traversed by an agent. Hence, the WE is constrained by a minimum vertical clearance v_{min} and a maximum slope α_{max}. Triangles in the WE are adjacent if and only if agents can move directly from one to the other.

C. Additional definitions

We define several terms to allow an unambiguous discussion of gaps.

Definition 1. An edge $e = \{\mathbf{v_1}, \mathbf{v_2}\}$ is a **boundary edge** when it is part of only one triangle.

Definition 2. The **associated plane** of a boundary edge $e = \{\mathbf{v_1}, \mathbf{v_2}\}$ is the plane through $\mathbf{v_1}$ and $\mathbf{v_2}$ perpendicular to the ground plane. We orient the associated plane such that the triangle that e is a part of falls on the negative side. In point-normal notation, it is the plane formed by any point on e (we arbitrarily take $\mathbf{v_1}$), and the normal calculated as $\mathbf{n} = \frac{(\mathbf{v_2}-\mathbf{v_1})\times\mathbf{Z}}{|(\mathbf{v_2}-\mathbf{v_1})\times\mathbf{Z}|}$, where $\mathbf{Z}$ is the vector pointing straight up.

IV. PROBLEM STATEMENT

Informally, a gap is a region where parts of the mesh are close to each other, but not directly connected. We first make two assumptions:

Assumption 1. The input to our algorithm is a walkable environment WE characterised by a minimum vertical clearance v_{min} and a maximum slope α_{max}.

Assumption 2. The output of our algorithm is the given WE, augmented with polygons indicating traversable gaps. Triangles of the input WE may be subdivided, but are not otherwise modified.

Assumption 3. We fill gaps up to a maximum distance of $d_{max} < \frac{v_{min}}{2}$.

The first and second assumption simply define our input and output. The third assumption prevents ambiguous cases in our algorithm. Note that this assumption is not restrictive in practice: for agents representing humans, d_{max} will be on the order of 0.3 metres (large enough to cover typical stairs and steps), while v_{min} will be on the order of 2 metres, roughly the height of a person.

We only consider boundary edges for filling gaps, ignoring the interior of each triangle. This avoids the creation of singular edges (i.e. 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. Most navigation mesh generation methods do not handle such geometry well.

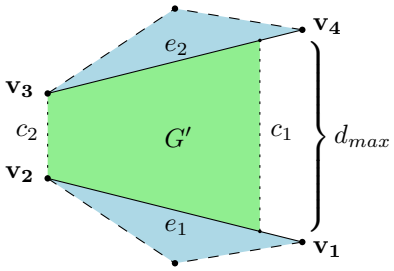


Fig. 2. Two boundary edges e_1 and e_2, with the triangles they are part of shown in blue, are connected by a gap filler G', shown in green. G' is bounded by connections c_1 and c_2, both having length at most d_{max}.

We first define the part of a boundary edge that may be used to connect to another boundary edge:

Definition 3. The **connectable section** of a boundary edge e_1 with respect to boundary edge e_2 is the part of e_1 that is on the non-negative side of the associated plane of e_2.

The restriction by the associated planes ensures that a boundary edge cannot be connected to something that lies "behind" it (this is further explained in Section V-B.3).

Our definition of a gap in a walkable environment is then as follows:

Definition 4. There is a **gap** between two boundary edges e_1 and e_2 iff the connectable section of e_1 with respect to e_2 is a non-degenerate line segment and vice versa.

We define the gap G between two boundary edges e_1 and e_2 as the set of points that lie on a line segment of length at most d_{max} with its endpoints on the connectable sections of e_1 and e_2. Unfortunately, the resulting shape is not necessarily linear, so we cannot represent it with a finite set of triangles. Therefore, we use the representation depicted in Fig. 2. 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 want the gap filler to be as wide as possible. In this context, we define the *width* of a gap as follows:

Definition 5. If the gap filler G' between boundary edges e_1 and e_2 is given by connections $c_1 = \{\mathbf{p_1}, \mathbf{p_2}\}$ and $c_2 = \{\mathbf{p_3}, \mathbf{p_4}\}$, with $\mathbf{p_1}$ and $\mathbf{p_3}$ on e_1 and $\mathbf{p_2}$ and $\mathbf{p_4}$ on e_2, then the **width** of G' is defined as $\text{width}(G') = \min(|\mathbf{p_1} - \mathbf{p_3}|, |\mathbf{p_2} - \mathbf{p_4}|)$.

We want the gap fillers to be as wide as possible to allow the agents passing over it to be as wide as possible. In addition, this 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 V-B), we have multiple ways of connecting a maximum width gap. 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 $|\mathbf{p_1} - \mathbf{p_3}|$ and $|\mathbf{p_2} - \mathbf{p_4}|$ equal and as large as possible.

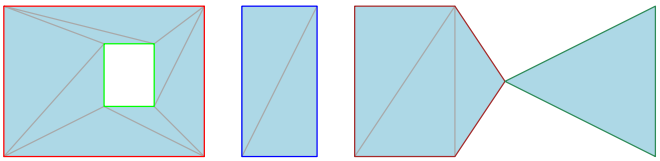


Fig. 3. An environment with five cycles of boundary edges, highlighted in different colours.

It is important to note that the presence of a gap filler does not imply an agent 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_{max} apart. Such cases will need to be handled during simulation by the path- or motion planning algorithm. For instance, this algorithm may give a higher cost to sections of the path running over a gap filler, or it 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.

V. THE ALGORITHM

Our algorithm 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 more detail. Note that filling a hole means connecting the gaps between boundary edges in the same cycle. When analysing complexity, we use n for the number of triangles and m for the number of boundary edges in the input. Note that $m \in \Theta(n)$ in the worst case, but we typically expect the number of boundary edges to be lower.

A. Finding cycles of boundary edges

In the first step, we simply find the cycles of boundary edges in the WE, which we can do in $O(n)$ time, as connectivity information is contained in the WE. Note that we consider multiple cycles touching in a single vertex to be separate cycles, as we do not consider triangles touching in one point to be adjacent. See Fig. 3 for an example.

B. Detecting gaps

In this step we detect gaps between pairs of boundary edges. For each boundary edge $e_1 = \{\mathbf{v_1}, \mathbf{v_2}\}$ we determine its axis-aligned bounding box and expand it by d_{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_{max} of the edge. For each boundary edge $e_2 = \{\mathbf{v_3}, \mathbf{v_4}\}$ in the query result, we attempt to make a connection with e_1. As each bounding box can intersect at most $O(m)$ 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 four steps:

1) Split edges at closest points.

2) Find two connections between pairs of split edges.
3) Limit result to non-negative side of associated plane.
4) Test for intersection with obstacles.

The first three steps take constant time, while the last step takes at most $O(f)$ time, with f being the number of triangles in the 3D environment (note that in practice, this typically takes much less time). This gives an overall time complexity of $O(fm^2)$ for the detection of all gaps.

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 $\mathbf{u_1}$ and $\mathbf{u_2}$, with $\mathbf{u_1}$ on e_1 and $\mathbf{u_2}$ on e_2. If $|\mathbf{u_1}-\mathbf{u_2}| > d_{max}$, we know that there exist no points on e_1 and e_2 with distance at most d_{max}, so the gap is too large, and we discard it. Similarly, unless the edges are parallel, if $|\mathbf{u_1}-\mathbf{u_2}| = d_{max}$, there is only one pair of points with distance at most d_{max}: the gap is discarded, as the resulting connection would have no area. Otherwise, we use the closest points to split the edges into $e_{11} = \{\mathbf{v_1}, \mathbf{u_1}\}$, $e_{12} = \{\mathbf{u_1}, \mathbf{v_2}\}$, $e_{21} = \{\mathbf{v_3}, \mathbf{u_2}\}$ and $e_{22} = \{\mathbf{u_2}, \mathbf{v_4}\}$. Because our edges are always counter-clockwise 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}.

This approach does not work when e_1 and e_2 are parallel: we have no unique closest points $\mathbf{u_1}$ and $\mathbf{u_2}$. In this case, we connect each vertex as far along the other edge as possible.

2) Connect pairs of split edges: When e_1 and e_2 are not parallel, we need to connect the pairs of split edges to each other. For e_{11} and e_{22}, we do this by finding the points $\mathbf{p_1}$ on the supporting line of e_{11} and $\mathbf{p_2}$ on the supporting line of e_{22} which are equally far from the closest points and have distance d_{max} to each other. In other words, we have $|\mathbf{p_1} - \mathbf{u_1}| = |\mathbf{p_2} - \mathbf{u_2}|$ and $|\mathbf{p_1} - \mathbf{p_2}| = d_{max}$. If either of these points is not on its respective edge, the vertex of that edge must be within d_{max} of the other edge, and we connect that vertex as far along the edge as possible. The connection between e_{12} and e_{21} is found in the same way.

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 Fig. 4 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 Fig. 5, it could mean that the resulting polygons could no longer be oriented consistently, making the interpretation of the walkable surface ambiguous. 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.

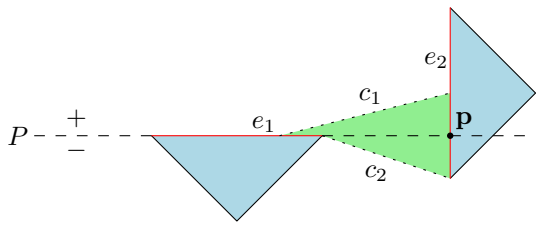


Fig. 4. Connection c_2 has a vertex on the negative side of the associated plane P of e_1. We restrict that vertex of c_2 to the intersection of P and e_2, represented by point $\mathbf{p}$. If no intersection exists, we detect no gap between the two edges.

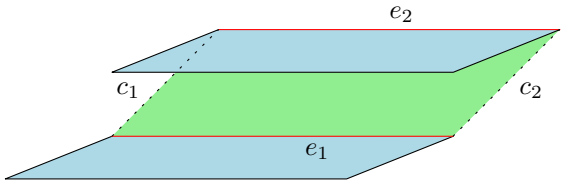


Fig. 5. A connection that cannot be oriented consistently. Such a connection is invalid.

4) Test obstacle intersection: The space between two boundary edges is not necessarily empty; obstacles may be present in the 3D environment, such as walls. To prevent connecting two boundary edges through obstacles, we test the final gap filler for intersection with the 3D environment. We discard any gap filler that has an intersection, unless the intersecting geometry is coplanar with the gap filler. This exception prevents discarding gap fillers when they overlap with the vertical parts of steps that were removed when computing the WE.

C. Connecting cycles

For the connections made between distinct cycles of boundary edges, we employ a heuristic. We do this because there is no unique way to connect these cycles, and because considering all possible choices is too expensive. The 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.

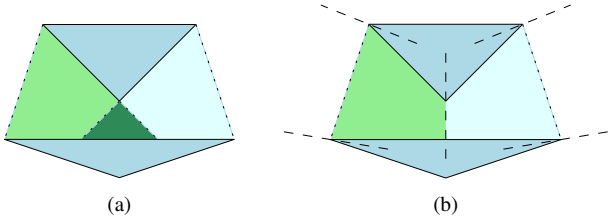


Fig. 6. Two gap fillers before and after being restricted by vertical planes along the bisectors of adjacent edges. The blue polygons are the original input, the light green and light cyan colours show the different gap fillers, and the dark green shows the part where they overlap. Fig. (a) shows the two gap fillers originally constructed, while Fig. (b) shows the result of limiting the gap fillers by using the vertical planes along the bisectors of adjacent edges (shown as dashed lines).

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 Fig. 6. Then, it determines 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 with the associated plane. The remaining gap fillers are used to build *interval maps* of fillers along each edge. The interval maps subdivide each edge into intervals that are connected by the same set of fillers. 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 $e_1 = \{\mathbf{v_1}, \mathbf{v_2}\}$ for the filler connecting it to edge $e_2 = \{\mathbf{v_3}, \mathbf{v_4}\}$, with the filler having points $\mathbf{p_1}$ and $\mathbf{p_2}$ on e_1 and $\mathbf{p_3}$ and $\mathbf{p_4}$ on e_2. We first calculate the points on e_1 corresponding to the interval as $\mathbf{i_1} = e_1(t_1)$ and $\mathbf{i_2} = e_1(t_2)$. We then convert the interval on e_1 to an interval on the filler, which we calculate as $[t'_1, t'_2]$, with $t'_1 = \frac{|i_1 - p_1|}{|p_2 - p_1|}$ and $t'_2 = \frac{|i_2 - p_1|}{|p_2 - p_1|}$. We obtain the corresponding part of the filler on e_2 by taking $\mathbf{i_3} = \mathbf{p_4} + t'_1 * (\mathbf{p_3} - \mathbf{p_4})$ and $\mathbf{i_4} = \mathbf{p_4} + t'_2 * (\mathbf{p_3} - \mathbf{p_4})$. Note that $\mathbf{p_3}$ and $\mathbf{p_4}$ are reversed with respect to the usual counter-clockwise 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 $\mathbf{i_3}$ and $\mathbf{i_4}$ to the line segment $\{\mathbf{i_1}, \mathbf{i_2}\}$ when projected onto the ground plane.

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. 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.

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:

Definition 6. 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 takes each mutual interval and connects 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 worst case, for a total worst case complexity of $O(m^3)$. 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 $O(m^3)$.

D. Filling holes

Holes are filled using the gap fillers detected between boundary edges in the same boundary edge cycle. We currently 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 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 that are not on the existing boundary, for which we need to derive a Z-coordinate. We take a weighted average of the Z-coordinates of all boundary vertices, with weight inversely proportional to the distance to the new vertex.

VI. 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 can give some insight into the size of agents that would be able to navigate the environment.
- **Performance:** we are interested in the computational performance of our algorithm, so we measure the time it takes to apply our algorithm to each environment.

In addition, we perform a manual visual inspection to look for possible defects or errors. We compare the results obtained with our algorithm to those obtained by Recast [10]. We also attempted a comparison with NEOGEN [11], but the implementation made available to us gave an empty output on most of our environments, so we disregard it here.

TABLE I
DESCRIPTION OF THE FOUR TEST ENVIRONMENTS.

Name	Vertices	Triangles	Source	Description
Holes	399	431	Our own	A horizontal plane with holes of different shapes and sizes.
Platforms	128	192	Our own	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.
Terrace house	2400	6155	3D Warehouse[1]	A two-storey house. We removed the doors and windows to allow passage between different areas, as well as the cars in the parking space and the toilet bowl and kitchen sink.
Town house	1476	2676	3D Warehouse[2]	A two-storey house on a road. We removed the doors and windows. We also removed the roof beams to decrease the time needed to find the WE. We added a large rectangle under the entire scene to model the ground.

[1] `https://3dwarehouse.sketchup.com/model.html?id=ud50c65de-58f8-4018-95cd-8b524733e14c`
[2] `https://3dwarehouse.sketchup.com/model.html?id=ufceb8906-a322-426a-8cea-f0e60120447d`

A. Test environments

We tested our algorithm on 17 different inputs taken from a variety of sources. Eight of the environments were created by us to test specific aspects of the algorithm's performance, the other nine were taken from online sources and chosen to more closely represent real use cases. We highlight four of the results here, but the reader is encouraged to view the full results in reference [17]. Table I describes the four environments.

B. Implementation

Our algorithm was implemented in C++ and makes extensive use of CGAL [2]. In particular, we use the exact computation provided by CGAL's CORE library. This means we never have problems with errors due to limited precision, but have higher memory consumption and running times. 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 is sufficiently precise. The scores simply use 64-bit floating-point values, as a high precision is not important in these calculations.

We obtain a walkable environment from a 3D environment by applying the pipeline proposed by Polak [12]. We used the ECM crowd simulation framework [15] to measure the success rate of planning paths for agents of different widths.

C. Testing system

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 [2] and Boost version 1.63 for the R-tree and the interval maps [1].

D. Recast

We used a slightly modified version of Recast [10], used in a comparative study of navigation meshes by Van Toll et al. [16]. This version allows the program to be run from the command line, and exports the results to a file. To make a fair comparison possible, we used the smallest allowed voxel size (i.e. 0.1 metres). In addition, we set the climb height to the value of d_{max}, 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 reference [16].

VII. RESULTS

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. The images with a top-down perspective are rendered orthographically to prevent perspective distortions near the edges.

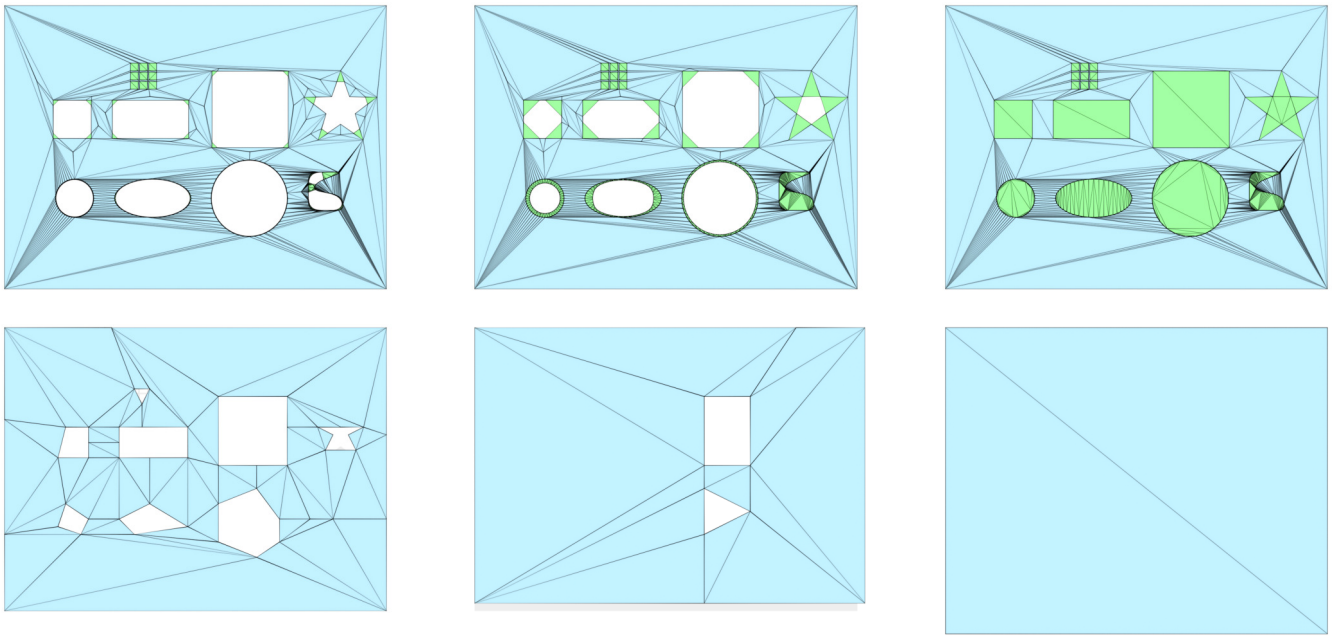
A. Holes environment

We include this environment to explore the effects of using different values of d_{max}, as seen in Fig. 7. As expected, the holes are progressively more filled as d_{max} increases, expanding from the places where the boundaries are closest together, until they are all completely filled when $d_{max} = 1.0$. The time needed to run our algorithm is dominated by the step in which holes are filled, and decreases from 66 seconds at $d_{max} = 0.1$ to 11 seconds at $d_{max} = 1.0$. This is because fewer new vertices are introduced in the environment, as more holes are filled completely.

Recast also progressively fills the holes more as the voxel size increases, but this method is less predictable, and the shape of the remaining holes is often only loosely related to the shape of the hole in the input. Setting the voxel size to 1.0 ensures coverage of the entire area, but makes the result protrude over the edges of the input. Recast processes the environment in 0.1 ms.

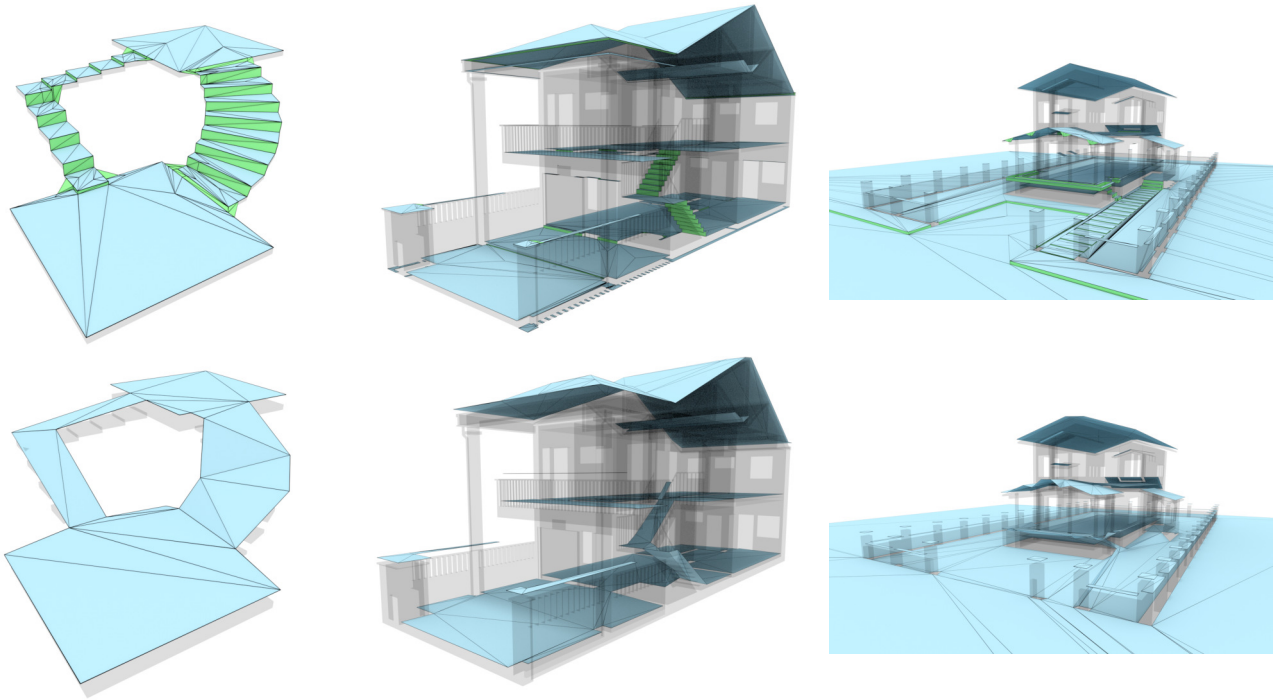
B. Platforms environment

Fig. 8a shows that 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 two



(a) d_{max} = voxel size = 0.1 (b) d_{max} = voxel size = 0.3 (c) d_{max} = voxel size = 1.0

Fig. 7. The results for the Holes environment. The top row contains the results from our algorithm and the bottom row those obtained with Recast.



(a) Platforms (b) Terrace house (c) Town house

Fig. 8. The results for the Platforms, Terrace house and Town house environments. The top row shows the results obtained with our algorithm and the bottom row those obtained with Recast.

steps above it, indicating that it is possible to cut that corner of the stairs. Our algorithm takes about 50 seconds.

Recast handles all the steps well, resulting in one large connected area, but there is much 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. Recast processes the environment in 0.9 ms.

C. Terrace house environment

Fig. 8b shows that our algorithm outputs two main components: the interior of the house, containing both the ground- and first floor, and the roof. Our algorithm takes about 18 seconds on this environment.

Recast 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. Recast processes the environment in 203 ms.

D. Town house environment

Fig. 8c shows that 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. Our algorithm needs about 946 seconds to process this environment, but about 98% of the time is spent discarding the gap fillers that intersect the environment.

Recast loses all the steps again. The front parts 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. Recast processes the environment in 400 ms.

E. Agent width

The path planning experiments gives similar results between our method and Recast, indicating that both methods produce results with roughly equally wide corridors.

VIII. CONCLUSION

We have presented a method for finding and filling gaps in walkable environments. Our algorithm does not modify the input and makes a clear distinction between the walkable areas and the gaps that can be traversed.

We compared our algorithm to Recast [10], a voxel-based method for navigation mesh generation. 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, due to the semantic annotation. However, Recast is much faster than our own method.

Limitations. Discarding gaps when they intersect an obstacle is too restrictive. For instance, we do not typically care about intersections where only a small part of the environment protrudes through a gap filler. In addition, we only handle direct intersections; gap fillers with low vertical clearance are not accounted for.

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 a different projection method. The field of mesh parameterisation [13] may offer mapping algorithms from 3D to 2D that would suit our application. Such an algorithm must also map newly created vertices from 2D back to 3D.

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_{max}. This could be resolved by considering chains of boundary edges forming 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 [4], who take a similar approach in the context of mesh repair.

Finally, our implementation is slow, and may require large amounts of memory. We could improve this with a robust implementation using finite precision, but this is often challenging for complex geometrical algorithms.

REFERENCES

[1] "Boost C++ libraries," 2016, http://www.boost.org (version 1.63).

[2] "Computational geometry algorithms library," 2016, http://www.cgal.org (version 4.9).

[3] M. Attene, M. Campen, and L. Kobbelt, "Polygon Mesh Repairing: An Application Perspective," *ACM Computing Surveys*, vol. 45, no. 2, pp. 15:1–15:33, 2013.

[4] G. Barequet and M. Sharir, "Filling gaps in the boundary of a polyhedron," *Computer Aided Geometric Design*, vol. 12, no. 2, pp. 207–229, 1995.

[5] A. Hillebrand, M. van den Akker, R. Geraerts, and H. Hoogeveen, "Performing multicut on walkable environments," in *International Conference on Combinatorial Optimization and Applications*, 2016, pp. 311–325.

[6] F. Lamarche, "Topoplan: a topological path planner for real time human navigation under floor and ceiling constraints," *Computer Graphics Forum*, vol. 28, no. 2, pp. 649–658, 2009.

[7] T.-Y. Li and P.-Z. Huang, "Planning humanoid motions with striding ability in a virtual environment," in *Proceedings of the IEEE International Conference on Robotics and Automation*, vol. 4. IEEE, 2004, pp. 3195–3200.

[8] T. Lopez, F. Lamarche, and T.-Y. Li, "Space-time planning in changing environments: using dynamic objects for accessibility," *Computer Animation and Virtual Worlds*, vol. 23, no. 2, pp. 87–99, 2012.

[9] T. Lozano-Perez, "Spatial planning: A configuration space approach," *IEEE Transactions on Computers*, no. 2, pp. 108–120, 1983.

[10] N. Mononen, "Recast navigation toolkit," 2016, https://github.com/recastnavigation/recastnavigation.

[11] R. Oliva and N. Pelechano, "NEOGEN: Near optimal generator of navigation meshes for 3D multi-layered environments," *Computers & Graphics*, vol. 37, no. 5, pp. 403–412, 2013.

[12] R. M. Polak, "Extracting walkable areas from 3D environments," MSc thesis, Utrecht University, 2016.

[13] A. Sheffer, E. Praun, and K. Rose, "Mesh Parameterization Methods and their Applications," *Foundations and Trends in Computer Graphics and Vision*, vol. 2, no. 2, pp. 105–171, 2006.

[14] W. van Toll, A. F. Cook IV, M. J. van Kreveld, and R. Geraerts, "The Medial Axis of a Multi-Layered Environment and its Application as a Navigation Mesh," 2017, https://arxiv.org/abs/1701.05141.

[15] W. van Toll, N. Jaklin, and R. Geraerts, "Towards Believable Crowds: A Generic Multi-Level Framework for Agent Navigation," in *ICT.OPEN 2015*, 2015.

[16] W. van Toll, R. Triesscheijn, M. Kallmann, R. Oliva, N. Pelechano, J. Pettré, and R. Geraerts, "A Comparative Study of Navigation Meshes," in *Proceedings of the 9th International Conference on Motion in Games*, 2016, pp. 91–100.

[17] J. L. Vermeulen, "Bridging Gaps in Walkable Environments," MSc thesis, Utrecht University, 2017.