Universiteit Utrecht

Master Thesis

ICA-3476901

Navigating Jumping Agents Through Virtual Environments

Supervisor:
Dr. R.J. Geraerts

Author:
Pieter van de Kerkhof

Second Examiner:
Prof. Dr. Marc van Kreveld

July 7, 2016

Abstract

In many computer games and crowd simulations, a navigation mesh is used as the underlying data structure for navigating agents through the virtual environment. A navigation mesh defines the areas where the agents are allowed to walk, and it enables agents to compute walkable paths through the environment. However, a navigation mesh does not provide information about where the agents are allowed to jump, so it does not support the discovery of paths that include jumps over gaps and obstacles. As a result, the agents will often take long detours to move to their designated positions. They will always walk around the obstacles in the environment, and will never take shortcuts by jumping over them.

In this MSc thesis, we present algorithms for extending navigation meshes with jump links, which are objects that represent valid jumps in the environment. Our approach can generate these links fully automatically, and works on any environment that is defined by a triangle soup. In addition, we present algorithms for planning paths on navigation meshes that are extended with jump links. These algorithms enable the agents to find new paths through the environment, which include jumps over gaps and obstacles. We have implemented our methods as an extension to Recast Navigation [20], which is a commonly-used software package for generating navigation meshes. Our experiments show that this implementation is capable of producing commodious sets of jump links for large and complex environments in a matter of seconds. This makes our implementation a useful extension to the Recast Navigation pipeline, enabling it to produce navigation meshes for applications that involve jumping agents.

Contents

1 Introduction

Computer games and simulations often include autonomous characters, or *agents*, that inhabit the virtual world. One of the main challenges in these applications is navigating the agents through the virtual environment. When given the task to move to a certain position, an agent must be able to find a *path* that leads to that position. Ideally, this path should be smooth, relatively short, and should have a certain amount of clearance from static obstacles. Once a path has been found, the agent must follow this path without colliding with other agents or dynamic obstacles.

The problem of navigating agents through virtual environments has been studied extensively in the last couple of decades [14]. It has not only received much attention in the game industry, but also in the fields of robotics, crowd simulation, and animation, to name just a few. Most of the research dedicated to this problem focuses on agents that walk and run. However, little attention has been paid to agents with other types of locomotion, such as jumping, vaulting, or climbing. In this thesis, we propose a solution to the problem of navigating *jumping agents*, which are agents that have the ability to jump from one position to another.

1.1 Project Motivation

In most of the navigation models that are used for computer games, agents compute their paths on a *navigation mesh*, which is a data structure that represents the areas where the agents are allowed to walk. A navigation mesh does not provide information about where the agents are allowed to *jump*, so it does not support the discovery of paths that include a jump from one position to another. The only paths that can be found on a navigation mesh are paths that can be traversed by walking. Therefore, agents will often take long detours by walking around obstacles instead of jumping over them.

One possible solution to this problem is to extend the navigation mesh with *jump links*. These links represent valid jumps in the environments, and they can be used to compute paths that include these jumps (Figure 1). Jump links can be created manually, but this is a time-consuming task, which can be prone to errors. Hence, there is a need for algorithms that can generate jump links *automatically*. A few of such algorithms have been proposed in the past, but none of them has found its way to become a generally-excepted approach. In this thesis, we will propose a new method for the automatic generation of jump links, and we will show how agents can use these links to find paths that include jumps.

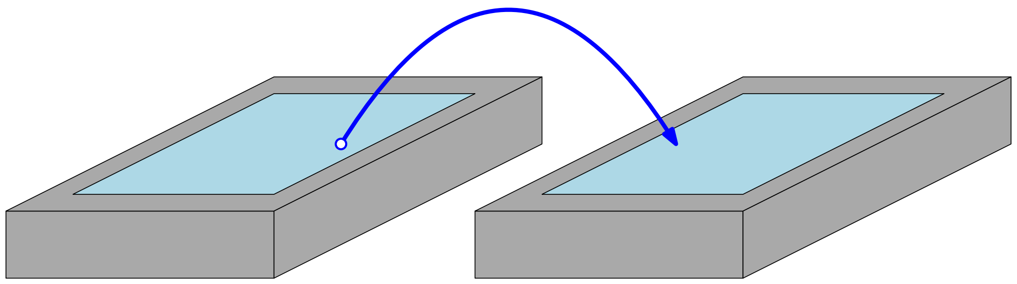


Figure 1: A virtual environment consisting of two platforms. The navigation mesh, consisting of the two light-blue polygons, represents the areas of the environment where the agents are allowed to walk. A jump link, which is shown in dark-blue, defines a valid jump between two positions in the navigation mesh, and can be used to plan a path from one platform to the other.

1.2 Contributions

We present an automated approach for generating jump links for arbitrary three-dimensional environments. Our approach consists of sampling positions from the walkable space of the environment, and evaluating potential jumps that start at these positions. For each sample point, we generate a set of line segments that approximate the land positions of the jumps that are collision free, and we create a jump link for each of these line segments. Each of our jump links thus consists of a sample point and a line segment, where the line segment represents a set of positions that an agent can jump to from the sample point.

We present a modified version of an existing path-planning method, which can be used to find paths on navigation meshes that are extended with our jump links. This method effectively utilizes the *point-to-segment* structure of the links, to find relatively-short paths that include jumps.

This project has been executed as part of an internship at the game studio *Guerrilla Games* [7], during the production of the game *Horizon: Zero Dawn* [11]. The navigation model that is used in this game is based on the navigation model of *Recast Navigation* [20], which is an open-source software package for navigating agents in virtual worlds. The methods that we present in this thesis are specifically designed for navigation meshes that can be constructed with this software.

We have created plug-in tools for Recast Navigation that contain our methods' functionalities, and we have tested this implementation on a variety of different environments. Our results show that our method is capable of generating commodious sets of jump links for large environments in a respectable amount of time.

1.3 Thesis Structure

The rest of this thesis is organized as follows. In Section 2, we discuss previous research that is related to our topic. We give a brief overview of navigation models that are commonly used for computer games, and discuss previous work related to the problem of navigating jumping agents. Then, in Section 3, we give an overview of some of the algorithms and data structures from *Recast Navigation*, which is the software in which we have implemented our methods. In Section 4, we give a mathematical description of the problem of navigating jumping agents, and we decompose this problem into two subproblems. Section 5 is devoted to the first subproblem, which is the automatic generation of jump links. We present our algorithm for finding jump links, and give a detailed description of all the steps that are involved. Section 6 is devoted to the second subproblem, which is planning a path that includes jumps. We present a modified version of an existing path-planning method, which can be used to find paths on a navigation mesh that is extended with jump links. In Section 7, we discuss the implementation of our techniques into the Recast Navigation software, and in Section 8, we discuss experiments that we performed to test this implementation. Finally, in Section 9, we conclude that our implementation can be expected to produce satisfying results for any virtual environment, but that there are still some problems that need to be solved before it will be ready to be used in a practical application.

2 Related Work

In this section, we discuss some previous work that is related to our topic. We will first give an overview of navigation models that are commonly used for computer games, and then discuss previous research related to the problem of navigating jumping agents.

2.1 Navigation Models in Computer Games

To be able to find paths through a virtual environment, agents need to know where they are allowed to walk. Together, the areas where the agents are allowed to walk form the *walkable space* of the environment. In this section, we give an overview of different data structures that can be used to represent the walkable space.

In most of the classic computer games, the environment is represented by a square grid, in which each cell is labeled “traversable” or “not-traversable” (Figure 20). Together, the traversable cells form a representation of the walkable space of the environment. By applying the A* algorithm [8] on these cells, an agent can compute a path to the position that it wants to move to. This approach is still used in many simple two-dimensional games, but is not suited for games with large and complex worlds. These games require large grids that consist of many small cells, which are costly in terms of storage, and lead to large computational overhead.

In most of the older three-dimensional games, the walkable space is represented by a *navigation graph* (which is also known as a *waypoint graph*, or *roadmap*) (Figure 2b). A node in this graph represents a position where an agent is allowed to stand, and an edge represents a traversable path between two nodes. A path between two arbitrary nodes can be found by applying the A* algorithm on the graph.

In today’s computer games, the walkable space is commonly represented by a *navigation mesh* [26] (Figure 2c). A navigation mesh is a polygon mesh that represents the surfaces of the environment that agents are allowed to walk on. The A* algorithm can be applied on the dual graph of this mesh to find a sequence of polygons that includes a path between two given positions. The *funnel algorithm* [10] can then be used to find the shortest path between the two positions that is contained within this sequence of polygons.

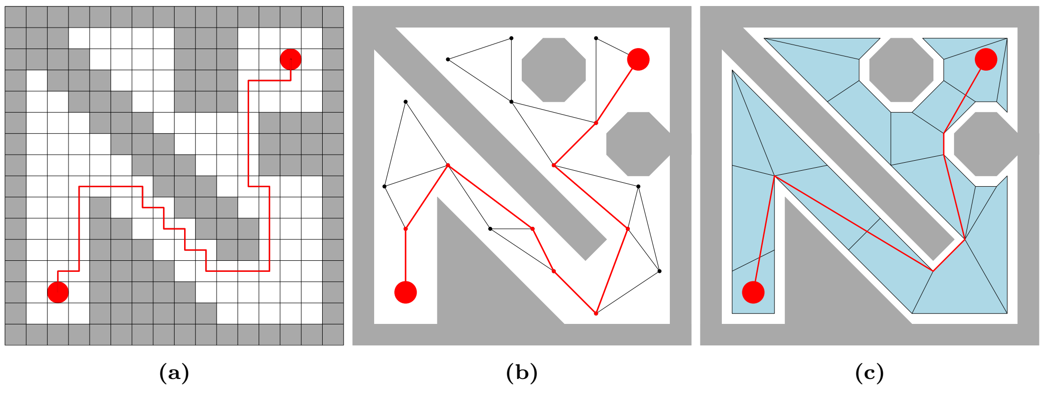


Figure 2: Different navigation models for computing a path between two positions in a virtual environment (which is shown from a top-down perspective). (a) The environment is represented by a square grid, where each cell is labeled “traversable” (white) or “not-traversable” (gray). (b) The walkable space is represented by a navigation graph. (c) The walkable space is represented by a navigation mesh.

An important advantage of a navigation mesh, with respect to a navigation graph, is that it represents *all* of the walkable space, and not just a subset of it. This gives agents the ability to deviate from their paths, as long as they stay within the boundaries of the navigation mesh. Having the ability to deviate from their paths, allows the agents to avoid collisions with other agents and dynamic obstacles. Algorithms that help agents to avoid collisions are often referred to as *steering models* [14].

A navigation mesh can be created manually, but this is a tedious and time-consuming task. Fortunately, there are many methods and tools for generating navigation meshes automatically [12]. One example is Mononen’s *Recast Navigation* project [20]. This software package provides algorithms for generating navigation meshes, and it provides algorithms for planning paths on these meshes. In this thesis project, we will extend the Recast Navigation software with our methods for navigating jumping agents.

2.2 Navigating Jumping Agents

To be able to find paths that include jumps, agents need to know where they are allowed to jump. Agents should not be allowed to perform jumps that will lead to collisions with the objects in the environment. Therefore, potential jumps should be evaluated before the agents are allowed to perform them. In this section, we review some previous research on evaluating potential jumps in a virtual environment.

Levine et al. propose a navigation model that incorporates the evaluation of potential jumps into the path-planning phase [17]. In this model, the walkable space is represented by a set of *landmarks*, which define positions in the environment where agents are allowed to stand. Examining if an agent can move freely from one landmark to another happens during the planning phase. This is done by iteratively testing random configurations of locomotion controllers (including a jump controller), and checking if the resulting trajectories form collision-free paths between the two landmarks. An advantage of evaluating jumps during the planning phase is that there is no need to search the entire environment for potential jumps; only jumps that would actually be included into an agent's path need to be evaluated. However, since evaluating a potential jump is a computationally-expensive operation, this approach leads to relatively long planning times for each agent. Therefore, it is not suited for games that involve large numbers of agents, and have limited computation time reserved for path queries.

If agents are required to find paths with jumps in a relatively short amount of time, then these jumps should be stored within the data structure that represents the navigable space. A navigation mesh, for example, can be extended with *jump links*, which are objects that represent valid jumps that can be incorporated into an agent's path (Figure 1). Some game engines, such as *Unity3D* [29] and *CryEngine* [3], provide tools that can be used to create these links. However, since the links must be placed manually, using such tools can be time-consuming and prone to errors. The user might forget to place the links at certain locations in the environment, or worse, place them at locations where agents should not be allowed to jump. Therefore, instead of generating jump links manually, it is better to generate them *automatically*.

One approach for generating jump links *semi*-automatically is to extract them from player data. Geisler uses machine learning on game statistics to determine the locations where players often jump from one position to another [6]. Kang et al. use a similar approach, and show that it is possible to utilize any newly-discovered jumps *while* the game is being played [13]. When a player discovers a jump between two positions, their model can insert this jump into the navigation mesh at runtime. One disadvantage of this approach is that it relies on the players of the game to discover the jumps. Also, the approach is only applicable to games where the player's character represents the same type of entity as the agents. If the player's character has a different shape than the agents, then it is not guaranteed that the agents can perform the same jumps as the player.

Ideally, the method for generating jump links should not rely on any human interaction. Therefore, there is a need for algorithms that can generate these links fully-automatically, using only the geometry of the environment as input. Two of such algorithms have been proposed by Mononen [21] and Sunshine-Hill [28]. Mononen's algorithm, which has recently been integrated into the Unity3D engine [30], samples jump trajectories that start on a boundary edge of the navigation mesh, and follow the direction of the outward-pointing normal vector of that edge. If a sufficient amount of neighboring trajectories is collision-free, these trajectories are merged together into a jump link (Figure 3). Sunshine-Hill's method also evaluates only those jumps that start on a boundary edge of the navigation mesh, and follow the direction of the normal vector of that edge. In contrast to Mononen's approach however, his algorithm evaluates all of these jumps in a single procedure, by sweeping the entire edge along a jump trajectory. An advantage of a jump link that is generated by Mononen's or by Sunshine-Hills's algorithm, is that it encodes *multiple* valid jump trajectories. This provides agents with some flexibility when incorporating the link into their paths, because the link contains several different jumps that the agents can choose from.

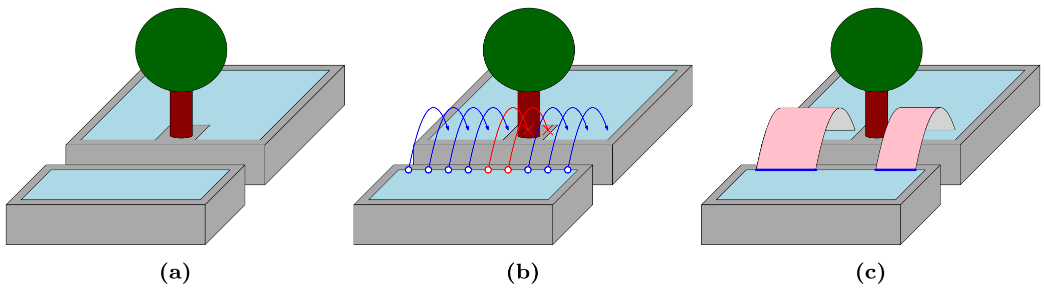


Figure 3: Visualization of Mononen's algorithm. (a) A virtual environment consisting of two platforms and a tree. The navigation mesh is shown in light blue. (b) Mononen's algorithm samples a set of jumps that start on a boundary edge of the navigation mesh, and have directions that are perpendicular to that edge. The two red jumps are invalid; one of them collides with the tree, and the other one does not land on the navigation mesh. The blue jumps are all valid. (c) Neighboring valid jumps are merged together into jump links. Each jump link encodes multiple valid jump trajectories, and can therefore be used in various ways; the agents are allowed to jump from any position on the dark-blue line segments, as long as they jump in the direction that is perpendicular to the boundary edge. The same type of jump link can be generated with Sunshine-Hill's algorithm.

The two algorithms described above only evaluate jumps that start on a boundary edge of the navigation mesh, and follow the direction of the normal vector of that edge. Jumps that do not meet these restrictions, such as the one shown in (Figure 4), will not be discovered by either of these methods. Algorithms that *are* able to find these types of jumps have been proposed by Choi et al. [1] and Lopez et al. [18]. Their approaches are based on randomly sampling pairs of positions from the walkable space, and 'fitting' jump trajectories between those positions. Since the positions are sampled randomly, the jump trajectories that are evaluated can have arbitrary directions, which do not depend on the geometry of the navigation mesh. However, a disadvantage of the jump links that are generated by these algorithms is that each of them only represents a *single* jump trajectory. Therefore, an agent does not have any flexibility when incorporating one of these link into its path, since there is only one way in which the jump can be made.

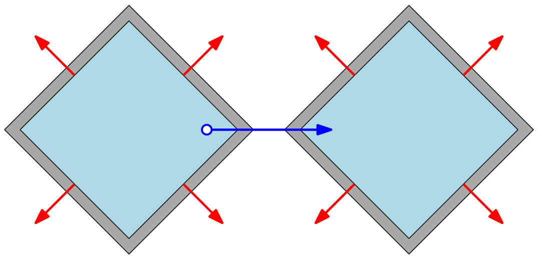


Figure 4: A top-down view on a virtual environment consisting of two platforms. The red arrows represent the directions of the outward-pointing normal vectors of the boundary edges of the navigation mesh. The blue arrow represents a jump from the left platform to the right platform. In this example, an agent can only jump from one platform to the other, by jumping in a direction that is not perpendicular to any boundary edges of the navigation mesh. Therefore, neither Mononen's algorithm nor Sunshine-Hill's algorithm will be able to generate any jump links for this environment.

In this project, we propose a new solution to the problem of navigating jumping agents. We focus on computer games with large and complex worlds that are inhabited by large populations of agents. We conclude, for these types of games, that the best way to tackle the problem is to extend the navigation mesh with jump links. These jump links should be created in a preprocessing stage (before any path planning takes place), to ensure relatively-short planning times for each agent. Also, generating jump links should be done automatically, without any need for human intervention. We have seen that several generation algorithms have been proposed, but that all of them have certain disadvantages. Jump links that are generated by the algorithms proposed by Choi et al. and Lopez et al. represent single jump trajectories, and do not provide the agents with any flexibility when it comes to using the links. Jump links that are generated by the algorithms proposed by Mononen and Sunshine-Hill represent infinite sets of jump trajectories, and can therefore be used in various ways. However, these algorithms fail to find jumps in directions that are not perpendicular to any boundary edges of the navigation mesh.

Our method for generating jump links samples positions from the walkable space, and evaluates potential jumps that start at these positions. Unlike the other methods that follow this strategy, we do not just evaluate jumps in one direction, but we evaluate jumps in *every* direction. A jump link created by our algorithm consists of a start position (a sample point), and a connected set of valid *land positions*. When incorporating a jump link into its path, an agent is thus obliged to start the jump at the designated start position, but has some freedom when it comes to choosing the position to jump to. Our results show, if the start positions are sampled dense enough, that the agents can use our links to find relatively short paths that include jumps.

3 Preliminaries: Recast Navigation

In this section, we discuss concepts from Mononen's *Recast Navigation* project, which is an open-source software package for navigating agents through virtual environments [20]. Recast Navigation contains two toolkits, which are called *Recast* and *Detour*. Our methods for navigating jumping agents utilize some of the algorithms and data structures that are provided by these toolkits. Therefore, we will first give an overview of these algorithms and data structures, before we present our approach for navigating jumping agents.

The Recast toolkit can be used to generate a navigation mesh for a three-dimensional virtual world. The algorithms that are used to build this mesh operate on a *heightfield*, which is a data structure that approximates the shape of the virtual environment. In Section 3.1, we will explain how the heightfield is constructed, because we will reuse this data structure in our algorithm for generating jump links.

The Detour toolkit provides algorithms for navigating agents through a virtual environment. In Section 3.2, we give an overview of the algorithms that enable the agents to find paths through the environment, using a navigation mesh that is created with Recast. Later, in Section 6, we will show how these algorithms can be modified, so that the agents can use them to find paths that incorporate the jump links that are generated with our method.

3.1 Recast

Three-dimensional game worlds are commonly defined by large, unorganized collections of triangles, which are known as *triangle soups*. Together, the triangles from a triangle soup form the shape of a virtual environment. By using the Recast toolkit, it is possible to generate a navigation mesh for any environment that is defined by a triangle soup. The algorithms responsible for building this mesh operate on a *heightfield*, which is a data structure that approximates the shape of the environment. In this section, we explain how Recast constructs a heightfield for a given environment, because we will reuse this data structure in our algorithm for generating jump links. It should be noted that our explanation is not an exact representation of Recast's source code; for the sake of explanation, some concepts have been changed or simplified. For a more precise explanation of Recast's algorithms, we refer the reader to [20] and [22].

To initialize the heightfield, the axis-aligned bounding box of the triangle soup is tessellated into a cube grid. A single cube in this grid is called a *voxel*, and a vertical strip of voxels is called a *column* (Figure 5a). The voxels that are intersected by triangles from the triangle soup are labeled as "*solid*". Voxels that are not intersected by any triangles are labeled as "*open*" (Figure 5b). Together, the open voxels form a representation of the 'free space' of the game world, and the solid voxels form a representation of the 'obstructed space'.

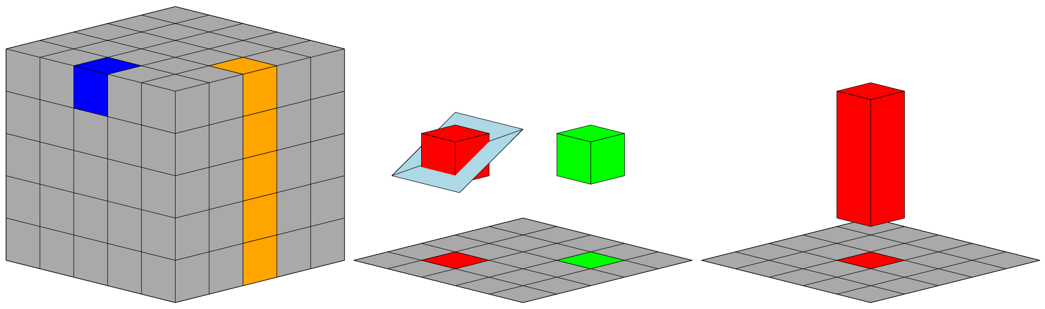


Figure 5: (a) A cube grid that is obtained by tessellating the axis-aligned bounding box of a triangle soup. A single cube in this grid is called a *voxel* (shown in blue), and a vertical strip of voxels is called a *column* (shown in orange). (b) Examples of solid and open voxels. The red voxel is a solid voxel, because it is intersected by two triangles from the triangle soup (shown in light blue). The green voxel is an open voxel, because it is not intersected by any triangles. (c) An example of a span, which is formed by merging three adjacent solid voxels that lie in the same column.

Adjacent solid voxels that lie in the same column are merged together into 'solid boxes', which are called *spans* (Figure 5c). These spans are divided into two categories: *walkable spans* and *not-walkable spans*. A span is labeled "walkable", if it satisfies the following two conditions (otherwise, it is labeled "not-walkable"):

1. The triangle, from which the top voxel of the span is derived, has a slope that does not exceed the value of the user-defined threshold parameter `maxSlope`. This condition ensures that the navigation mesh will not be build on top of the surfaces of the environment that are considered to be 'too steep to walk on'.
2. The distance between the top face of the span and the bottom face of the span that lies directly above it is larger than the height of an agent[1]. This condition ensures that the final navigation mesh will not represent areas where there is not enough room for agents to stand, due to overhanging obstacles.

Each walkable span represents a location in the environment where an agent is allowed to stand. Some of these spans can be grouped together to form a surface that agents are allowed to walk on. To determine which walkable spans form a walkable surface, the spans are connected to each other by *neighbor links*. Two walkable spans are considered to be neighbors, if an agent that is standing on the top face of one of the spans is able to walk directly to the top face of the other span. Specifically, a neighbor link between two walkable spans is created when the following three conditions are met:

1. The spans lie in different columns that are adjacent in the heightfield (not considering 'diagonal' adjacency).
2. The vertical distance between the top faces of the spans does not exceed the value of the user-defined threshold parameter `maxWalkableClimb`.
3. The vertical length of the intersection of the open areas above the spans is at least as large as the height of an agent.

Figure 6 shows examples of pairs of walkable spans that meet, or fail to meet these conditions. When multiple walkable spans are (indirectly) connected to each other by neighbor links,

[1]Recast assumes that all agents have the same size.

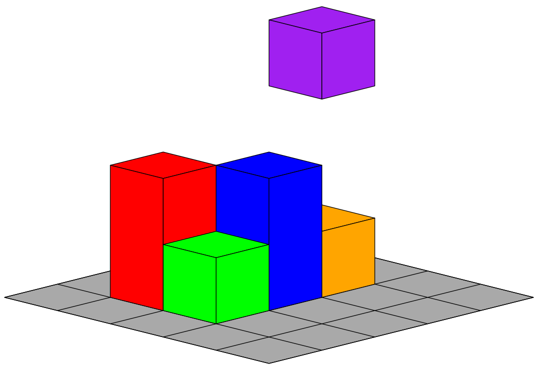
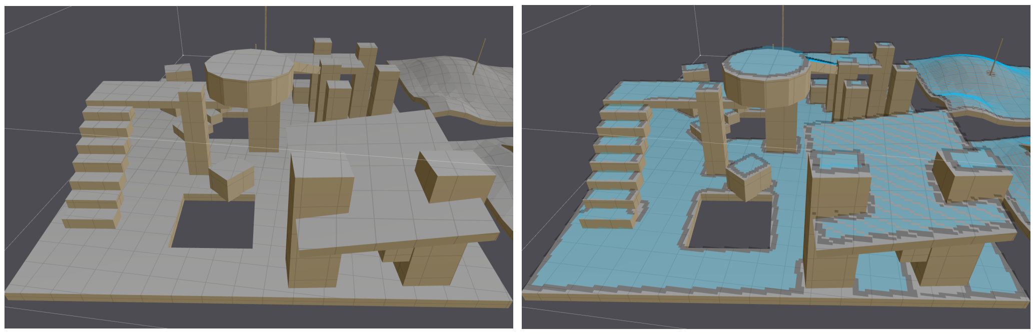


Figure 6: The red span is not a neighbor of the blue span, because its column is not adjacent to the column that contains the blue span. The column that contains the purple and orange spans *is* adjacent to the column that contains the blue span. However, the purple span is not a neighbor of the blue span, because the vertical distance between the top faces of these spans is too large. The orange span is also not a neighbor of the blue span, because the *open gap* between the two spans (i.e. the intersection of the open areas above the two spans) is too small for an agent to pass through. The green span *is* a neighbor of the blue span, because it meets all the necessary conditions.

they form a surface that agents are allowed to walk on. The walkable spans that have less than four neighbors form the boundary of such a surface. These spans are called *border spans*, and they often represent ledges or locations that are close to obstacles. To make sure that the agents will keep a sufficient amount of distance from the ledges and the obstacles, the border spans are relabeled as “not-walkable”. In addition, all the walkable spans that are ‘too close’ to a border span are relabeled “not-walkable” as well. Specifically, a walkable span is relabeled “not-walkable”, if the horizontal distance between that span and its closest connected border span is less than the width of an agent. This final pass over the spans ensures that the navigation mesh will not be build too close to ledges and obstacles.



(a) **(b)**

Figure 7: Two screenshots from the Recast Demo program [20]. (a) A virtual environment composed of a set of triangles. (b) The heightfield that is generated for the environment from Figure 7a. In this figure, only the top faces of the spans are shown. The top faces of the walkable spans are shown in light blue, and the top faces of the spans that are not walkable are shown in dark gray.

Figure 7 visualizes the resulting heightfield that is generated for a virtual environment. In the remaining steps in the Recast pipeline, a navigation mesh is extracted from the walkable spans in this heightfield and the neighbor links that connect them (Figure 8). For a detailed description of these steps, we refer the reader to [20] and [22].

The resulting navigation mesh is collection of polygons that represent the areas where agents are allowed to walk. Each of these polygons is defined by an alternating sequence of edges and vertices, forming a closed polygonal chain in $\mathbb{R}^3$. Formally, we should not call these polygonal chains “polygons”, because they are not necessarily planar objects (i.e. it may not always be possible to fit a plane through the vertices of a chain). However, when projected onto the ground plane, the polygonal chains from the navigation mesh form two-dimensional convex polygons, and it is on these projections that Detour's path planning algorithms operate. In the remainder of this thesis, when we speak of the "polygons of the navigation mesh", we will refer to the convex polygons that are obtained by projecting the edges and vertices of the navigation mesh onto the ground plane.

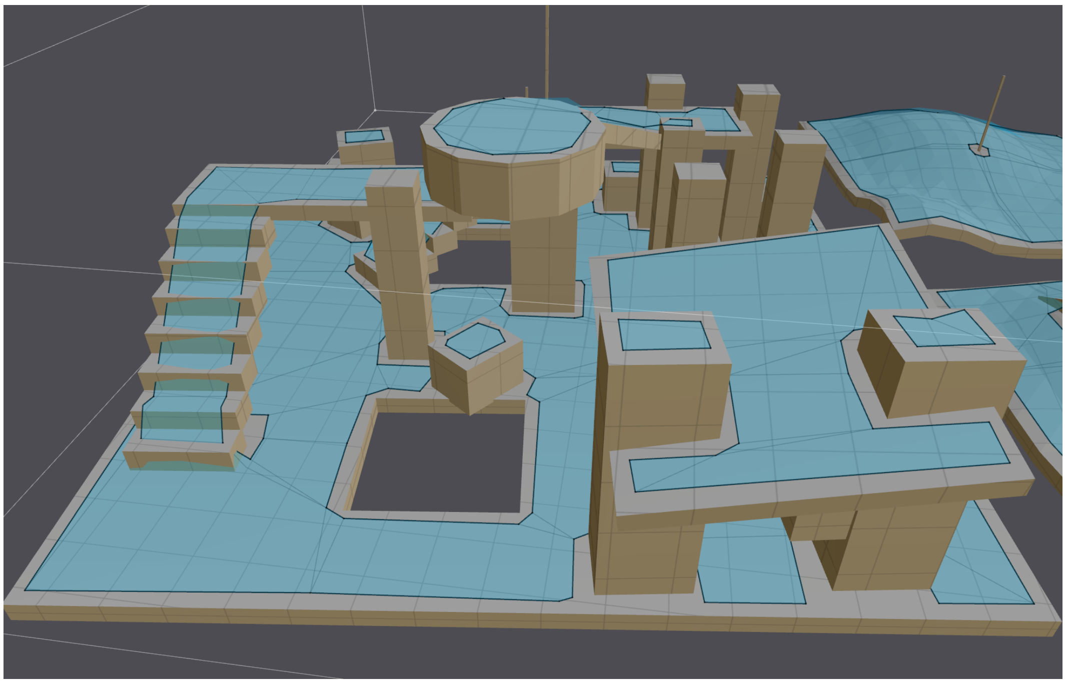


Figure 8: A screenshot from the Recast Demo program [20]. It shows the navigation mesh that is generated for the environment from Figure 7a.

3.2 Detour

In this section, we discuss two path-planning algorithms from the Detour toolkit. Agents can use these algorithms to find paths on a navigation mesh that is created with Recast. Later, in Section 6, we will show how these algorithms can be modified, so that they can be used to find paths on navigation meshes that are extended with jump links. It should be noted that our description of these algorithms is not an exact representation of Detour's source code; for the sake of explanation, some concepts have been changed or simplified. For an exact description of Detour's algorithms, we refer the reader to [20].

In Section 3.2.1, we will discuss Detour's *global path-planning algorithm*. This algorithm can be used to find a *global path* between a start position and a goal position. This global path defines an ordered set of adjacent polygons in the navigation mesh. This sequence of polygons, which is called a *corridor*, forms the input for Detour's *local path-planning algorithm*. This algorithm is used to find a *local path* that is contained within the corridor. In Section 3.2.2, we will discuss Detour's local planning algorithm, and show how an agent can use the local path to move to the goal position.

3.2.1 Global Path Planning

In this section, we discuss Detour's global path-planning algorithm. The input of this algorithm consists of a start position $\mathbf{s} \in \mathbb{R}^3$, a goal position $\mathbf{g} \in \mathbb{R}^3$, and a navigation mesh that is created with Recast. The output is a *corridor*, which is an ordered set of convex polygons, in which each two consecutive polygons share an edge.

The global planning algorithm operates on a two-dimensional navigation graph $G = (V, E)$, which is extracted from the navigation mesh (Figures 9a and 9b). The vertices in V are the centroids of the polygons of the navigation mesh[2]. A pair of vertices $\{v_i, v_j\} \in V \times V$ is connected with an edge $e_{i,j} \in E$ if, and only if, their polygons are adjacent (i.e. if the polygonal chains that define the polygons share an edge in the navigation mesh).

The start and the goal position are both assumed to be positions "in the navigation mesh". That is, when projected onto $\mathbb{R}^2$ (the ground plane), each of these positions is assumed to lie inside one of the polygons of the navigation mesh. If a position lies inside multiple polygons (which can happen if the navigation mesh represents vertically-overlapping areas of walkable space), then it is assumed that the polygon that the position 'belongs to' is known. Let $\mathbf{s}'$ be the projection of $\mathbf{s}$ onto $\mathbb{R}^2$, and let $\mathbf{g}'$ be the projection of $\mathbf{g}$ onto $\mathbb{R}^2$. If $\mathbf{s}'$ and $\mathbf{g}'$ lie inside the same polygon P, then the algorithm returns the set $\{P\}$. If, on the other hand, $\mathbf{s}'$ lies in a different polygon than $\mathbf{g}'$, then the graph vertex $v_s \in V$ that lies on the centroid of the polygon that contains $\mathbf{s}'$ is (temporarily) moved to $\mathbf{s}'$. Similarly, the graph vertex $v_g \in V$ that lies on the centroid of the polygon that contains $\mathbf{g}'$ is moved to $\mathbf{g}'$. The global planner then computes a *global path* between v_s and v_g by running the *A* algorithm* [8] on the new graph (Figure 9c). Such a global path is defined as an ordered set of vertices $V' \subseteq V$ (starting at v_s and ending at v_g), in which each pair of consecutive vertices is connected by an edge in E. The *length* of this path is defined as the sum of the lengths of the edges that connect the consecutive vertices in V'. The global path that is computed by the A* algorithm is the *shortest* global path, which means that it has the smallest length of all the possible global paths from v_s to v_g.

We will now give a brief description of the A* algorithm. For a more in-depth discussion of this algorithm, we refer to the reader to [32]. The A* algorithm performs a best-first search on the vertices of G. For each vertex $v \in V$ that is visited, the following values are computed:

- $f(v) \in \mathbb{R}$, which denotes v's *priority value*. It represents the estimated length of the shortest path from v_s to v_g that includes v.
- $g(v) \in \mathbb{R}$, which denotes the length of the shortest path from v_s to v found so far.
- $c(v) \in V$, which denotes v's preceding vertex in the shortest path from v_s to v found so far.

The algorithm maintains two lists of vertices during the search: the *open list* and the *closed list*. The open list $\mathcal{O}$ contains all the vertices in V that are candidates for processing. The open list is a priority queue in which the vertices are ordered by their priority values $f(v)$. Initially, this queue only contains v_s. The closed list $\mathcal{C}$ contains all the processed vertices. Whenever a vertex in V is processed, it is added to $\mathcal{C}$, and will never be processed again.

While $\mathcal{O}$ is not empty, the algorithm removes the vertex $v_i \in \mathcal{O}$ with the smallest $f(v)$-value from $\mathcal{O}$, and adds it to $\mathcal{C}$. For each *neighboring* vertex $v_j \notin \mathcal{C}$ of v_i (i.e. v_j is a vertex that is adjacent to v_i in G), the algorithm then computes the *tentative cost* $t(v_j) := g(v_i) + \mathit{length}(e_{i,j})$, where $g(v_i)$ is the length of the shortest path from v_s to v_i found so far, and $\mathit{length}(e_{i,j})$ is the Euclidean length of the edge $e_{i,j} \in E$ from v_i to v_j. The following cases can occur:

1. If $v_j \notin \mathcal{O}$ (which means that v_j is visited for the first time), then $g(v_j)$ is set to $t(v_j)$, and $f(v_j)$ is set to $g(v_j) + \mathit{dist}(v_j, v_g)$, where $\mathit{dist}(v_j, v_g)$ is the Euclidean distance between v_j and v_g. Also, $c(v_j)$ is set to v_i, which means that v_i precedes v_j in the shortest path from v_s to v_j found so far. Then, v_j is inserted into $\mathcal{O}$ according to its priority value $f(v_j)$.

[2]Recall, from the last paragraph of Section 3.1, that the "polygons of the navigation mesh" refer to the convex polygons that can be obtained by projecting the edges and vertices of the navigation mesh onto the ground plane.

2. If $v_j \in \mathcal{O}$, then this means that v_j has been visited before. In this case, if $t(v_j) \geq g(v_j)$, then current path to v_j (the path where v_i precedes v_j) is at least as long as another path to v_j that was found earlier. However, if $t(v_j) < g(v_j)$, then the current path to v_j is shorter than any of the paths that were found earlier. In that case, $g(v_j)$, $f(v_j)$, and $c(v_j)$ are updated in the same way as in Case 1, and v_j is re-inserted into $\mathcal{O}$ according to its new priority value $f(v_j)$.

This process repeats itself until v_g is removed from $\mathcal{O}$. If $\mathcal{O}$ runs empty before this happens, then it follows that no path from v_s to v_g exists. When v_g is removed from $\mathcal{O}$, the algorithm returns the global path defined by the sequence $V' = \{v_0, v_1, \ldots, v_n\}$, where $v_0 = v_s$, $v_n = v_g$, and $v_i = c(v_{i+1})$, for $i \in \{1, 2, \ldots, n-1\}$. The vertices in V' define a corridor $C = \{P_0, P_1, \ldots, P_n\}$, in which each P_i is the polygon that corresponds to the vertex $v_i \in V'$ (Figure 9c). This corridor forms the input for Detour's *local path-planning algorithm*, which we will discuss in the next section.

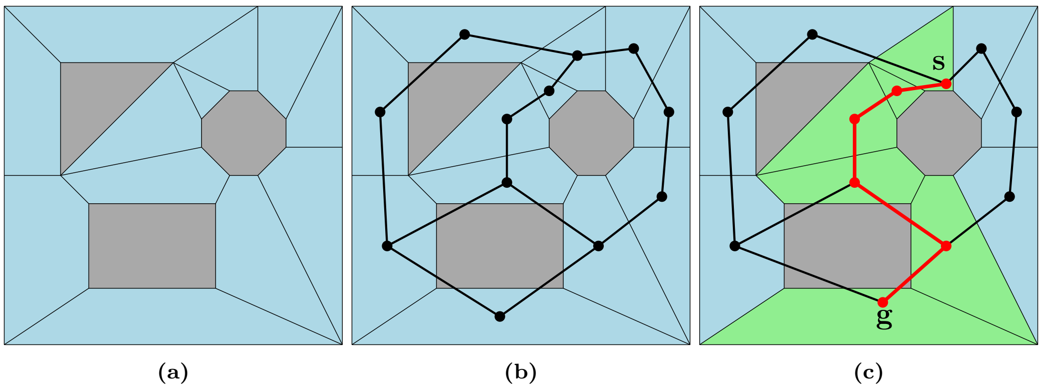


Figure 9: (a) A top-down view on a virtual environment. The dark-gray polygons represent obstacles in the environment, and the light-blue polygons represent the navigation mesh. (b) The navigation graph that is extracted from the navigation mesh. The vertices in this graph are the centroids of the polygons of the navigation mesh. Two vertices are connected with an edge if, and only if, their polygons are adjacent. (c) A global path between a start position **s** and a goal position **g**. The path is shown in red, and the corresponding corridor is shown in light green.

3.2.2 Local Path Planning

In this section, we discuss Detour's local path-planning algorithm, which is also known as the *simple stupid funnel algorithm* (SSFA) [19]. The input of this algorithm consists of a start position $\mathbf{s}' \in \mathbb{R}^2$, a goal position $\mathbf{g}' \in \mathbb{R}^2$, and a corridor $C = \{P_0, P_1, \ldots, P_n\}$ [3]. It is assumed that $\mathbf{s}'$ lies in P_0 and that $\mathbf{g}'$ lies in P_n. The output of the algorithm is an open polygonal chain, defined by an ordered set of vertices $V = \{v_0, v_1, \ldots, v_m\} \subset \mathbb{R}^2$, where $v_0 = \mathbf{s}'$ and $v_m = \mathbf{g}'$ (Figure 10). We will call such a polygonal chain a *local path* from $\mathbf{s}'$ to $\mathbf{g}'$, and we define its *length* as the sum of the Euclidean lengths of the line segments in the chain. The path that is computed by the SSFA is the *shortest* local path from $\mathbf{s}'$ to $\mathbf{g}'$ in C. This means that it has the smallest length of all the possible local paths from $\mathbf{s}'$ to $\mathbf{g}'$ that lie completely within the union of the polygons in C.

The SSFA maintains a *funnel* while iterating over the polygons in C. This funnel, F, is defined by three points $\mathbf{a}, \mathbf{l}, \mathbf{r} \in \mathbb{R}^2$ and two integers $i_l, i_r \in \mathbb{N}$. The point $\mathbf{a}$ is called the *apex* of F, and $\mathbf{l}$ and $\mathbf{r}$ are called the *left* and *right portal points* of F, respectively. The integers i_l and i_r are respectively called the *indices* of the left and right portal points. If $\mathbf{a}$, $\mathbf{l}$, and $\mathbf{r}$ are different points, then they are always defined such that the sequence $\{\mathbf{a}, \mathbf{l}, \mathbf{r}\}$ is ordered counter-clockwise (the points can never be co-linear). If $\mathbf{l} \neq \mathbf{a}$, then the half line that starts

[3]Here, $\mathbf{s}'$ and $\mathbf{g}'$ are the projections of $\mathbf{s}$ and $\mathbf{g}$ onto $\mathbb{R}^2$, respectively, where $\mathbf{s}$ and $\mathbf{g}$ are the input positions of the global planning algorithm, and C is the corridor that is defined by the global path that is computed by the global planning algorithm.

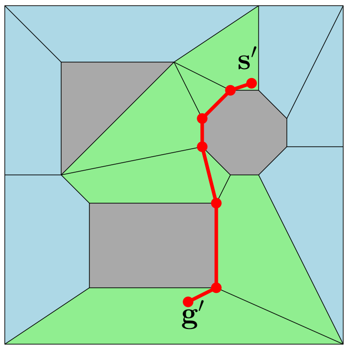


Figure 10: The local path that is computed by the SSFA for the example from Figure 9.

at $\mathbf{a}$ and passes through $\mathbf{l}$ is called the *left leg* of the funnel. Similarly, if $\mathbf{r} \neq \mathbf{a}$, then the half line that starts at $\mathbf{a}$ and passes through $\mathbf{r}$ is called the *right leg* of the funnel.

Suppose that $\mathbf{a}$, $\mathbf{l}$, and $\mathbf{r}$ are three different points. Let L be the line through $\mathbf{l}$ and $\mathbf{r}$, and let $\mathbf{p} \in \mathbb{R}^2$ be a point that lies on the other side of L than $\mathbf{a}$. Then $\mathbf{p}$ can have the following five different *orientations* with respect to F (Figure 11):

- $\mathbf{p}$ can lie on the other side of the left leg of F than $\mathbf{r}$. In that case, we say that $\mathbf{p}$ lies "to the left of F".
- $\mathbf{p}$ can lie on the left leg of F.
- $\mathbf{p}$ can lie on the same side of the left leg of F as $\mathbf{r}$, and on the same side of the right leg of F as $\mathbf{l}$. In that case, we say that $\mathbf{p}$ lies "inside F".
- $\mathbf{p}$ can lie on the right leg of F.
- $\mathbf{p}$ can lie on the other side of the right leg of F than $\mathbf{l}$. In that case, we say that $\mathbf{p}$ lies "to the right of F".

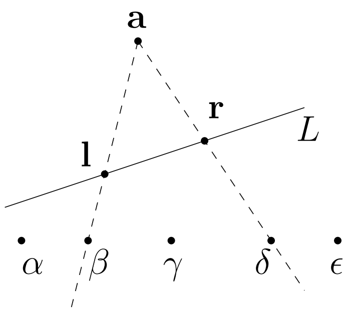


Figure 11: A funnel F, defined by the points $\mathbf{a}, \mathbf{l}$, and $\mathbf{r}$. The point α lies to the left of F; the point β lies on the left leg of F; the point γ lies inside F; the point δ lies on the right leg of F, and the point ϵ lies to the right of F.

The SSFA starts with a list of vertices V, which only contains $\mathbf{s}'$. If $\mathbf{g}'$ lies inside the same polygon as $\mathbf{s}'$ (which means that C only contains one polygon), then $\mathbf{g}'$ is added to V, and the algorithm terminates. Otherwise, the algorithm creates the funnel F. The apex $\mathbf{a}$ and the left and right portal points $\mathbf{l}$ and $\mathbf{r}$ are all set to $\mathbf{s}'$, and the indices i_l and i_r are set to 0. The algorithm then runs a loop over the shared edges of each two consecutive polygons in C, and triggers events depending on the orientations of the endpoints of these edges with respect to F. It maintains a counter $j \in \{0, 1, \ldots, n-1\}$, $n = |C|$, that denotes the *index* of the edge that is currently being processed. This edge, e_j, is the shared edge between the polygons P_j and P_{j+1}. We will denote the *left* and *right* endpoints of e_j by $\mathbf{e}_j^l$ and $\mathbf{e}_j^r$, respectively. That is, if we order the vertices of P_j counter-clockwise, then $\mathbf{e}_j^l$ is the first vertex of P_j that is incident to e_j, and $\mathbf{e}_j^r$ is the second vertex of P_j that is incident to e_j.

Depending on the orientations of $\mathbf{e}_j^l$ and $\mathbf{e}_j^r$ with respect to F, the following events can be triggered for e_j:

1. The left leg of F is moved to $\mathbf{e}_j^l$. This means that $\mathbf{l}$ is set to $\mathbf{e}_j^l$, and i_l is set to $j+1$. This event gets triggered if $\mathbf{l} = \mathbf{a}$, if $\mathbf{e}_j^l$ lies on the left leg of F, or if $\mathbf{e}_j^l$ lies inside F.
2. The right leg of F is moved to $\mathbf{e}_j^r$. This means that $\mathbf{r}$ is set to $\mathbf{e}_j^r$, and i_r is set to $j+1$. This event gets triggered if $\mathbf{r} = \mathbf{a}$, if $\mathbf{e}_j^r$ lies on the right leg of F, or if $\mathbf{e}_j^r$ lies inside F.
3. The algorithm is restarted at $\mathbf{l}$. This means that $\mathbf{l}$ is added to V, $\mathbf{a}$ and $\mathbf{r}$ are set to $\mathbf{l}$, and i_r and j are set to i_l. This event gets triggered if $\mathbf{e}_j^r$ either lies to the left of F, or lies on the left leg of F.
4. The algorithm is restarted at $\mathbf{r}$. This means that $\mathbf{r}$ is added to V, $\mathbf{a}$ and $\mathbf{l}$ are set to $\mathbf{r}$, and i_l and j are set to i_r. This event gets triggered if $\mathbf{e}_j^l$ either lies to the right of F, or lies on the right leg of F.

If the algorithm is not restarted after e_j is processed, then j is incremented with 1. The algorithm then continues by processing the edge that corresponds to the new value of j. Once j reaches n, the algorithm considers the orientation of $\mathbf{g}'$ with respect to F. If $\mathbf{g}'$ lies to the left of F, then the algorithm is restarted at $\mathbf{l}$, and if $\mathbf{g}'$ lies to the right of F, then the algorithm is restarted at $\mathbf{r}$. In any other case, $\mathbf{g}'$ is added to V, and the algorithm terminates. The local path is then found by connecting each pair of consecutive vertices in V with a line segment. Figure 12 visualizes the steps of the algorithm with an example. More examples can be found in [4] and [19].

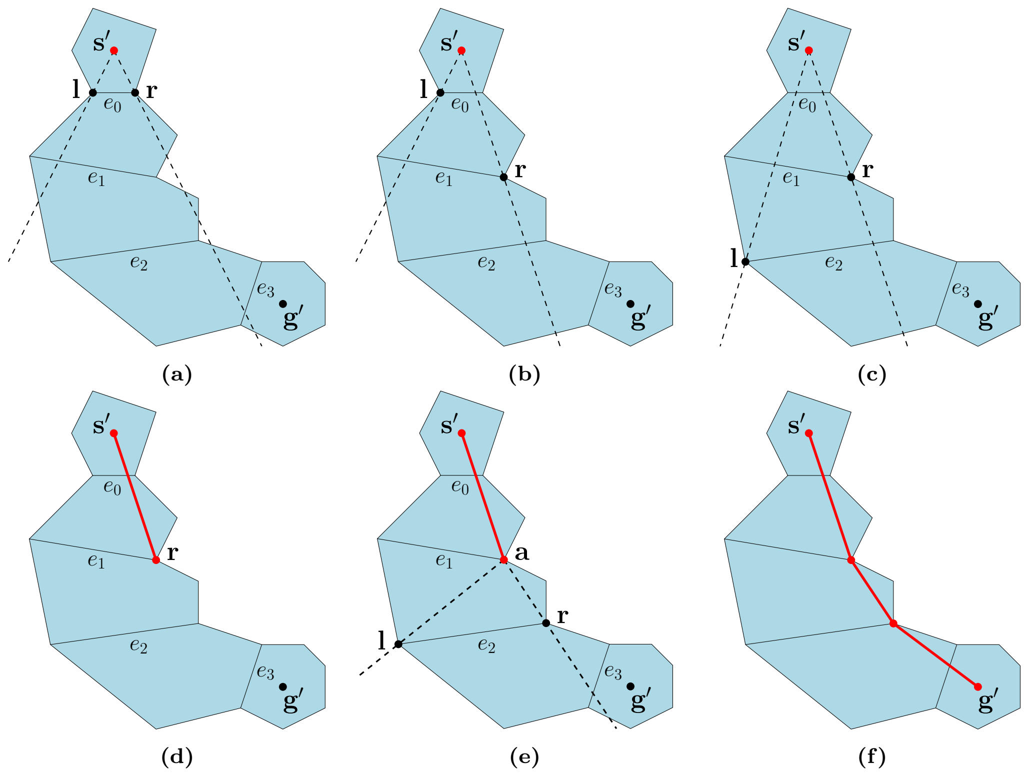


Figure 12: Visualization of the steps of the SSFA. Initially, this algorithm creates a funnel F, whose apex $\mathbf{a}$ and whose left and right portal points $\mathbf{l}$ and $\mathbf{r}$ are all set to the start position $\mathbf{s}'$. (a) In the first iteration, since $\mathbf{l} = \mathbf{a}$ and $\mathbf{r} = \mathbf{a}$, $\mathbf{l}$ is set to $\mathbf{e}_0^l$ and $\mathbf{r}$ is set to $\mathbf{e}_0^r$ (Events 1 and 2, respectively). (b) In the second iteration, since $\mathbf{e}_1^r$ lies inside F, $\mathbf{r}$ is set to $\mathbf{e}_1^r$ (Event 2). (c) In the third iteration, since $\mathbf{e}_2^l$ lies inside F, $\mathbf{l}$ is set to $\mathbf{e}_2^l$ (Event 1). (d) In the fourth iteration, since $\mathbf{e}_3^l$ lies to the right of F, the algorithm is restarted at $\mathbf{r}$. This means that $\mathbf{r}$ is added as a vertex to the local path, and that $\mathbf{a}$ and $\mathbf{l}$ are set to $\mathbf{r}$ (Event 3). (e) In the fifth iteration, since $\mathbf{l} = \mathbf{a}$ and $\mathbf{r} = \mathbf{a}$, $\mathbf{l}$ is set to $\mathbf{e}_2^l$ and $\mathbf{r}$ is set to $\mathbf{e}_2^r$ (Events 1 and 2, respectively). (f) The final local path from the start to the goal position, after the remaining steps of the algorithm are completed.

The local paths that are computed by the SSFA form the input for Detour's steering algorithms. These algorithms are responsible for moving the agents across the environment. Given a local path from an agent's current position to a goal position, a *desired velocity* can be calculated by subtracting the agent's current xy-position from the position of the next vertex in the local path. Given the desired velocity, the algorithm computes a *steering vector*, which serves as the *actual* velocity of the agent in the current update. The steering vector prevents the agent from colliding with other agents and from traveling outside the walkable space of the environment. The details of Detour's steering method fall beyond the scope of this project, so we will not discuss them here. For more information about this method, we refer the reader to [20].

4 Problem Description

The goal of this project is to find a solution to the problem of navigating agents that can jump. In this section, we give a formal definition of this problem. We start by noting that the problem is an instance of the *motion-planning problem*, which originated in the field of robotics [16]. Motion planning is the process of composing a sequence of motions that enables a robot (an agent) to fulfill a given task. Using the terminology from robotics, the context of the motion-planning problem can be described as follows. The virtual world, in which the motions take place, is called the *workspace* of the robot, and is denoted by W. In most applications, W equals $\mathbb{R}^2$, $\mathbb{R}^3$, or a subset of one of those (e.g. a rectangle or box that contains all the objects in the world). The robot's orientation in W is defined by its *configuration parameters*. Usually, this set of parameters includes a position $\mathbf{p} \in W$ and a heading $\theta \in [0, 2\pi]$, but it can also include parameters that describe the orientation of the joints of the robot. The set of all the possible configurations of these parameters is called the *configuration space*, and is denoted by C. The configuration space can be decomposed into the *obstacle space* and the *free configuration space*. The obstacle space C_{obs} is the set of configurations that cause the robot to collide with an object in W. The free configuration space C_{free} is defined as $C \setminus C_{obs}$, and consists of all collision-free configurations.

The motion-planning problem can be described as finding a continuous sequence of collision-free configurations that brings the robot from its original state into a desired goal state. Such a sequence of configurations is called a *path*, and it can be defined as the image of a continuous function $f : [a, b] \to C_{free}$, where $[a, b]$ is a interval in $\mathbb{R}$. Formally, the motion-planning problem can be defined as follows:

Problem 4.1. The Motion-Planning Problem *Given a start configuration $\boldsymbol{s} \in C_{free}$ and a goal configuration $\boldsymbol{g} \in C_{free}$, find a path p, which is the image of a continuous function $f : [a, b] \to C_{free}$, for which $f(a) = \boldsymbol{s}$ and $f(b) = \boldsymbol{g}$.*

In this project, we attempt to solve the motion-planning problem for *jumping agents*, which are agents whose locomotion consists of walking and jumping. In other words, we want to be able to compute paths through a virtual world, which can be traversed by combining walking and jumping motions. Before giving a formal definition of this problem, we will first define its context.

We assume that the workspace of the agents, W, is equal to $\mathbb{R}^3$. We will denote the horizontal axes of W by x and y, and denote the vertical axis of W by z (Figure 13a). The agents will be represented by their bounding volumes, which we assume are congruent cylinders whose axes are aligned with the z-axis of W (Figure 13b). Each agent will have only one configuration parameter, a position $\mathbf{p} \in W$, which will denote the coordinates of the center of the bottom face of the agent's bounding cylinder. The configuration space of the agents, C, is thus equal to $\mathbb{R}^3$.

We assume that W contains a finite collection of triangles $T = \{T_1, T_2, \ldots, T_n\}$. Together, these triangles form the shape of our virtual environment, which we will denote by E.

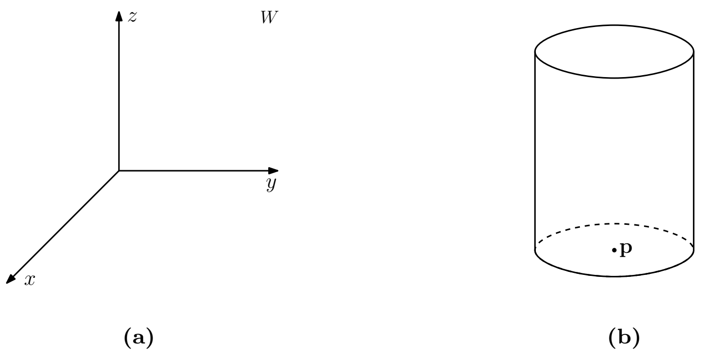


Figure 13: (a) The workspace of the agents, W, has a right-handed Cartesian coordinate system, where x and y denote the horizontal axes, and z denotes the vertical axis. (b) The bounding volume of an agent is a cylinder, whose axis is aligned with the z-axis of W. The position of an agent, $\mathbf{p} \in W$, denotes the position of the center of the bottom face of the agent's bounding cylinder.

Specifically, we define E as the subset of W that is formed by the union of the triangles in T, i.e. $E := \bigcup_{i=0}^{n} T_i$. We define the *walkable space* of E, which we will denote by N, as the subset of E that consists of all the positions where the agents are allowed to stand[4]. We define the free configuration space C_{free} as the union of N with the set of all the configurations for which the agent's bounding cylinder is not intersected by any of the triangles in T (notice that this set includes aerial positions). The obstacle space C_{obs} is defined as $C \setminus C_{free}$, and represents all invalid configurations.

We assume that the agents are subjected to a constant gravitational force, and that they are not affected by air resistance. This allows us to represent the ballistic trajectory that an agent traverses during a jump as a parabolic arc. The precise shape of this arc is determined by the configuration of the agents' *jump parameters*, which we define as follows:

Definition 4.2. *Each agent has the following* ***jump parameters****:*

- *A* ***jump height*** $h \in \mathbb{R}^+$*, which defines the vertical distance that the agent will travel upwards during a jump (Figure 14a).*
- *A* ***jump distance*** $d \in \mathbb{R}^+$*, which defines the horizontal distance that the agent will travel during a jump, before revisiting the height at which the jump was initiated (Figure 14b).*

We assume that the settings of the jump parameters are the same for each agent. Multiple different settings may be possible, but we assume that the total number of possible settings is finite. The different settings of the agents' jump parameters define the different *jumps* that an agent can perform. Formally, we define a jump as follows:

Definition 4.3. *A* ***jump*** *is defined by a jump height* $h \in \mathbb{R}^+$*, a jump distance* $d \in \mathbb{R}^+$*, a start position* $\boldsymbol{j} \in N$*, and a direction* $\theta \in [0, 2\pi]$ *(Figure 14c). We define it as a continuous function* $j : [t_{start}, \infty) \to C$*, for which*

$$\left.\begin{aligned} j_x(t) &= \boldsymbol{j}_x + \cos(\theta)(t - t_{start}) \\ j_y(t) &= \boldsymbol{j}_y + \sin(\theta)(t - t_{start}) \\ j_z(t) &= \boldsymbol{j}_z + \frac{4h}{d^2}(d(t - t_{start}) - (t - t_{start})^2) \end{aligned}\right\} \quad t \in [t_{start}, \infty). \tag{1}$$

The ***trajectory*** *of the jump is defined as the image of* j*; it is a parabolic arc that represents the infinite ballistic trajectory that the agent will traverse, when it does not collide with*

[4]Throughout this project, we will use different data structures to represent N. In our approach for generating jump links, we will use Recast to generate a heightfield for E, and use the top voxels of its walkable spans to represent N. In our path-planning algorithms, N will be represented by a navigation mesh that is created with Recast.

anything whilst being airborne.

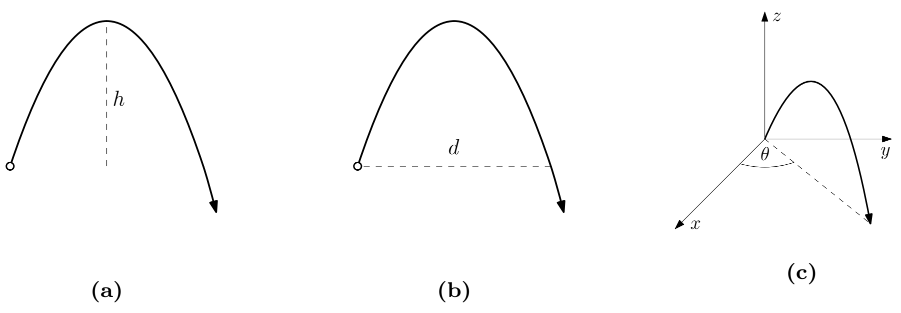


Figure 14: Visualization of the parameters that define the trajectory of a jump. (a) The *jump height*, $h \in \mathbb{R}^+$, is the vertical distance that an agent will travel upwards during a jump. (b) The *jump distance*, $d \in \mathbb{R}^+$, is the horizontal distance that an agent will travel during a jump, before revisiting the height at which the jump was initiated. (c) The *jump direction*, $\theta \in [0, 2\pi]$, is the angle between the positive x-axis and the projection of the jump's trajectory onto the ground plane.

Notice, in the definition above, that the start position of a jump must be a position in N. Applications in which agents are allowed to jump when they are not standing on the ground fall beyond the scope of this project. We assume that an agent can jump from any position in N, and that it can jump in any direction $\theta \in [0, 2\pi]$. However, from a path-planning perspective, jumping is only useful when it enables the agent to reach another position in N. In other words, we are only interested in jumps that have a *land position* in N. We define the land position of a jump as follows:

Definition 4.4. *The **land position $\boldsymbol{l}$** of a jump j is defined as*

$$\boldsymbol{l} := j(t_{land}), \text{ where } t_{land} := \min\{t \in (t_{start} + d/2, \infty) \mid j(t) \in N\}. \tag{2}$$

Notice that the land position always lies on the part of the trajectory where the agent is on its way down (the point $j(t_{start} + d/2)$ is the highest point on j's trajectory). If $j(t) \notin N$, $\forall t \in (t_{start} + d/2, \infty)$, then j does not have a land position.

In most computer games, there is a maximum distance that an agent can fall down without dying. In this project, we assume that such a *maximum fall distance* exists, and we will denote it by m. This means that an agent will not survive a jump that has a land position that lies more than a vertical distance of m below the highest point on the jump's trajectory. Since we do not want agents to accidentally commit suicide when trying to jump over a gap or an obstacle, we will consider such jumps to be *invalid*. Also, we do not want an agent to collide with any objects in W while it is airborne. For example, an agent should not attempt to jump over a fence, when this means that it will hit its head against an overhanging tree branch. Even though the agent may still be able to clear the fence this way, the result will likely be visually unpleasing. Therefore, we also consider a jump to be invalid, if the agent's bounding cylinder is intersected by a triangle in T while traversing the jump's trajectory. These requirements lead to the following definition of *valid* and *invalid* jumps:

Definition 4.5. *Let j be a jump. We call j a **valid** jump if, and only if, it meets all of the following conditions:*

- *j has a land position $\boldsymbol{l} := j(t_{land})$, where t_{land} is defined as in (2).*
- *j must be collision free, i.e. $j(t) \in C_{free} \setminus N$, $\forall t \in (t_{start}, t_{land})$.*
- *An agent must be able to survive j, i.e. $j(t_{land})$ must lie within a vertical distance of m below the highest point on j's trajectory.*

*If j does not satisfy all of these conditions, then we call it an **invalid** jump.*

We are now ready to give a formal definition of the problem that we attempt to solve in this project. As we mentioned before, we want to be able to compute a path through W that can be traversed by combining walking and jumping motions. We can define such a path as the image of a continuous piecewise function, for which each subfunction either defines a path through N, or is a valid jump between two positions in N. We can define the corresponding motion-planning problem as follows:

Problem 4.6. The Motion-Planning Problem for Jumping Agents *Given a start position $\boldsymbol{s} \in N$ and a goal position $\boldsymbol{g} \in N$, find a path p, which is the image of a function $f : [t_0, t_n] \rightarrow C_{free}$ that satisfies the following conditions:*

1. *$f(t_0) = \boldsymbol{s}$ and $f(t_n) = \boldsymbol{g}$.*
2. *f is a continuous piecewise function on $[t_0, t_n]$. This means that f consists of n subfunctions $f_0, f_1, \ldots, f_{n-1}$, where each f_i is a continuous function $f_i : [t_i, t_{i+1}] \rightarrow C_{free}$. Furthermore, it means that $f_0(t_0) = f(t_0)$, $f_{n-1}(t_n) = f(t_n)$, and $f_i(t_{i+1}) = f_{i+1}(t_{i+1}), \forall i \in \{0, 1, \ldots, n-2\}$.*
3. *Each subfunction f_i of f either defines a path through N, or is a valid jump between two positions $\boldsymbol{j}, \boldsymbol{l} \in N$.*

Problem 4.6 is the problem that we attempt to solve in this project. As was mentioned in Section 2.2, we seek a solution that separates the problem of finding valid jumps in the environment from the problem of finding a path through the environment. We thus decompose Problem 4.6 into the following sub problems:

1. Finding valid jumps in the environment.
2. Finding a path through the environment, which can be traversed by combining walking and jumping motions.

In Section 5, we will present our approach for solving the first sub problem; we will present an automated method for generating jump links that represent valid jumps in the environment. In Section 6, we will present our approach for solving the second sub problem; we will present a modified version of an existing path-planning method that can be used to find paths on navigation meshes that are extended with jump links. These paths will consist of walkable line segments and jump trajectories, and agents will be able to traverse them by combining walking and jumping motions.

5 Generating Jump Links

In this section, we present our method for generating jump links. We will give a general overview of our method in Section 5.1, and we will explain the individual steps in Sections 5.2 to 5.5.

5.1 Overview

In this section, we give an overview of our method for generating jump links. The input of this method consists of a virtual environment and a model of a jumping agent. The virtual environment E is defined as the union of a finite collection of triangles $T \subset \mathbb{R}^3$. The agent model A defines the width and the height of the agents' bounding cylinders. Given T and A, we will use Recast to construct a navigation mesh for E, and we will extend this navigation mesh with jump links.

In addition to defining the size of the agents, A also defines the configuration of the agents' jump parameters, and defines the maximum fall distance (Section 4). We will assume that there is only one possible configuration of the jump parameters, which means that we assume that the jumps that the agents can perform all have congruent trajectories. However, if an application requires jump links that represent trajectories with different shapes and sizes, then our algorithms can be repeated for each possible configuration of the jump parameters, producing a separate set of jump links for each configuration.

Similar to Recast, our algorithm for generating jump links operates on a data structure that approximates the shape of the virtual world. Specifically, our algorithm operates on a *dilated heightfield*, which is a modified version of the heightfield that is used by Recast. The dilated heightfield D is obtained by relabeling every open voxel in Recast's heightfield, that does not lie completely within the free configuration space C_{free}, as "solid" (Section 5.2). The set of open voxels in D is thus a subset of C_{free}, and we will use it as a representation of C_{free}. Together, all the other solid voxels in D will form our representation of the obstacle space C_{obs}. This will allow us to check jumps for collisions by tracing their trajectories through D, and will prevent us from having to use computationally-expensive volume-sweep algorithms. We will use the set of *walkable voxels* in D, which are the top voxels of D's walkable spans, to represent the walkable space N. We can then estimate the land position of a jump, by checking which walkable voxel in D is the first to be intersected by the trajectory of the jump.

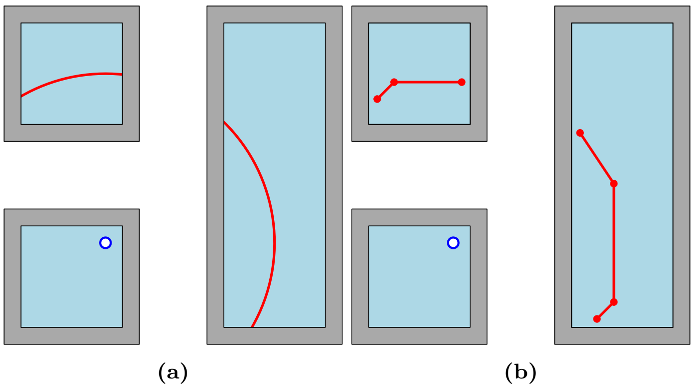


Figure 15: A top-down view on a virtual environment consisting of three platforms. The navigation mesh is shown in light-blue. The small dark-blue circle represents a sample point in the navigation mesh. (a) The red circular arcs represent the land positions of the valid jumps that start at the sample point. (b) The red line segments represent an approximation of the land positions of the valid jumps that start at the sample point.

We will start the generation process by sampling a set of points S, representing the start positions of potential jumps in the environment (Section 5.3). For each sample point $\mathbf{s} \in S$, we will try to find the land positions of all the valid jumps that start at $\mathbf{s}$. Together, these land positions form a set of *curves* in the walkable space. In Figure 15a, for example, the land positions of all the valid jumps that start at the indicated point form a set of circular arcs. We will not attempt to compute the exact curves that are formed by a set of land positions, because the complexity of these curves is not convenient for path planning. Instead, we will generate a set of line segments, called *land segments*, that approximate these curves (Figure 15b). To generate these line segments, we will first compute a list of *land voxels* $V_{\mathbf{s}}$ (Section 5.4). Each of these land voxels $v \in V_{\mathbf{s}}$ will be a walkable voxel in D, representing the land positions of a set of valid jumps that start at $\mathbf{s}$ (Figure 16a). After we compute $V_{\mathbf{s}}$, we will obtain an initial set of line segments by connecting the midpoints of the neighboring voxels in $V_{\mathbf{s}}$. That is, we will connect the midpoints of each two voxels $v_i, v_j \in V_{\mathbf{s}}$ that are part of two neighboring walkable spans in D. The result will be a set of polygonal chains C (Figure 16b). We will *simplify* each chain in C with a line-simplification algorithm, to obtain a more compact representation of the land positions (Section 5.5). This means that we will transform each polygonal chain $c \in C$ into a new polygonal chain c', that is similar to c, but consists of fewer vertices (Figure 16c). We will then create a jump link for each line segment in the simplified chain. That is, for each line segment l of c', we will create a jump link that consists of the start position $\mathbf{s}$ and the land segment l, where l represents the land positions of a set of valid jumps that start at **s**.

Figure 16 shows an overview of the steps of our method. In the following sections, we will provide detailed descriptions of these steps.

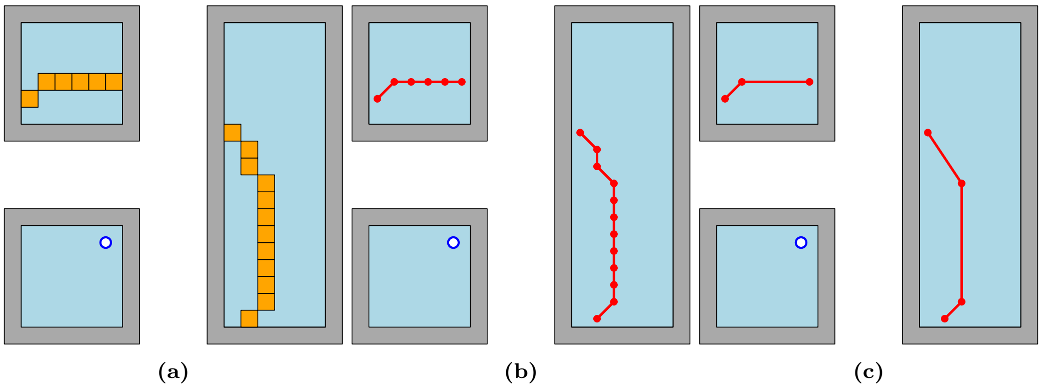


Figure 16: A visualization of the steps of our method for generating jump links, using the example from Figure 15. (a) We generate a set of land voxels that represent the land positions of the valid jumps that start at the sample point. (b) We connect the midpoints of neighboring land voxels, yielding a set of polygonal chains. (c) We simplify each polygonal chain, and we create a jump link for each line segment in the simplified chains.

5.2 Constructing the Dilated Heightfield

In this section, we show how to construct the *dilated heightfield*, which is the data structure on which our algorithm for generating jump links operates. To construct this data structure, we need the collection of triangles T that defines the virtual environment, and we need the agent model A that defines the width and the height of the agents' bounding cylinders. Given T and A, we use Recast to generate a navigation mesh, which we will later extend with jump links. Recall, from Section 3.1, that Recast extracts this navigation mesh from a *heightfield*, which is a three-dimensional grid, consisting of *open* and *solid* voxels. The open voxels in this heightfield represent the free space of the virtual world. However, if the position of an agent (i.e. the position of the center of the bottom face of the agent's bounding cylinder) lies inside an open voxel, then it is not guaranteed that the agent's bounding cylinder is not intersected by any triangles in T (Figure 17a). Therefore, the set of open voxels in Recast's heightfield is not necessarily a subset of the free configuration space C_{free}.

We construct the dilated heightfield D, by *dilating* Recast's heightfield. This means that we relabel every open voxel in Recast's heightfield, that does not lie completely within C_{free}, as "solid". We can use the remaining set of open voxels as a representation of C_{free}, and use the new set of solid voxels as a representation of the obstacle space C_{obs}. This will enable us to test a jump for collisions, by iterating over the voxels that are intersected by the jump's trajectory. If the trajectory intersects an open voxel, then we may conclude (with certainty) that an agent will not collide with anything while traversing the intersection. However, if the trajectory intersects a solid voxel, then there is a possibility that the agent will collide with the geometry of the environment while traversing the intersection.

A naive way to construct the dilated heightfield is as follows (more sophisticated techniques can be found in [27]): first, iterate over all the spans in Recast's heightfield. For each span s, relabel every open voxel that lies below s (in the same column), and has a vertical distance to s that is less than the height of an agent, as "solid". Then, merge these new solid voxels together with s. Next, in a secondary pass, iterate over all the spans again. For each span s, relabel every open voxel that overlaps with s on the vertical axis, and has a horizontal distance to s that is less than half the width of an agent. Finally, merge adjacent solid voxels that lie in the same column together into spans, and label these spans as "not-walkable". Figure 17b shows an example of a dilated heightfield.

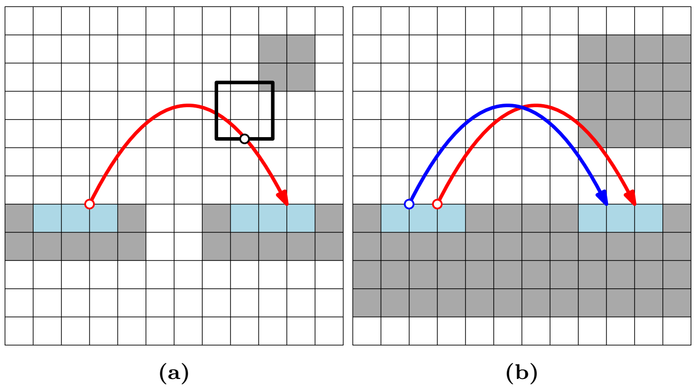


Figure 17: (a) A side view on a heightfield H that is generated by Recast. The *walkable voxels* of H (i.e. the top voxels of H's walkable spans) are shown in light blue, and the other solid voxels are shown in gray. The white cells represent the open voxels in H. The thick black square represents the bounding cylinder of an agent traversing the trajectory of a jump between two walkable voxels. Notice that the cylinder intersects a solid voxel, even though it is positioned inside an open voxel. (b) The dilated heightfield D that is generated for H. Notice that the red trajectory intersects a solid voxel in D, which means that the corresponding jump is not necessarily collision free. The blue trajectory however, does not intersect any solid, not-walkable voxels. Therefore, we may conclude, with certainty, that an agent traversing this trajectory will not collide with the geometry of the environment whilst airborne.

5.3 Choosing the Sample Points

In this section, we show how we generate our set of sample points S. Each of these sample points will represent a position in the navigation mesh, and will represent the start position of a set of potential jumps. Later, we will evaluate the jumps that start at our sample points, which involves checking for collisions with the geometry of the environment. As we mentioned in Section 5.2, we will perform these collision checks, by iterating over the voxels in the dilated heightfield D that are intersected by the trajectories of the jumps. In order to avoid detecting a collision with the solid voxel that represents the geometry near a sample point, we will choose our sample points from the top faces of walkable spans in D. Specifically, each of our sample points will be a *corner* of the top face of a walkable span. The reason for sampling from the corners of the spans will be made clear in Section 5.4.

To obtain a set of points that is (approximately) uniformly distributed spatially, we sample our points at a fixed user-defined step size $d \in \mathbb{N}$. We iterate over the columns in D, whose integer coordinates in D are divisible by d, and for each column, we iterate over its walkable spans. For each of those walkable spans $s \in D$, we check if the bottom-left[5] corner $\mathbf{c}_s$ of the top face of s is a point in the navigation mesh. That is, we check if the navigation mesh contains polygons that were derived from sets of walkable spans that include s, and we check if any of these polygons contains the projection of $\mathbf{c}_s$ onto $\mathbb{R}^2$. Only if the navigation mesh contains such a polygon, do we add $\mathbf{c}_s$ to S.

To determine if the corner of a span is a point in the navigation mesh, we use algorithms from the Recast pipeline. This pipeline contains an optional step, in which a *detail mesh* is extracted from the navigation mesh. This detail mesh is a triangle mesh, that represents a more accurate approximation of the walkable surfaces of the environment than the original navigation mesh. The detail mesh is not required for path planning, but it can be used to refine point positions returned by various Detour functions [23]. During the construction of the detail mesh, the *watershed algorithm* [9] is used to determine for each polygon of the navigation mesh, the set of walkable spans from which the polygon was derived. We will use the same technique to pre-compute a mapping that maps the bottom-left corner $\mathbf{c}_s$ of the

[5]Here, the *bottom-left* corner of the top face of s refers to the corner of the face that lies to the 'south-west' of the center of s, when viewing s from a top-down perspective. The choice of sampling points from the bottom-left corners is arbitrary.

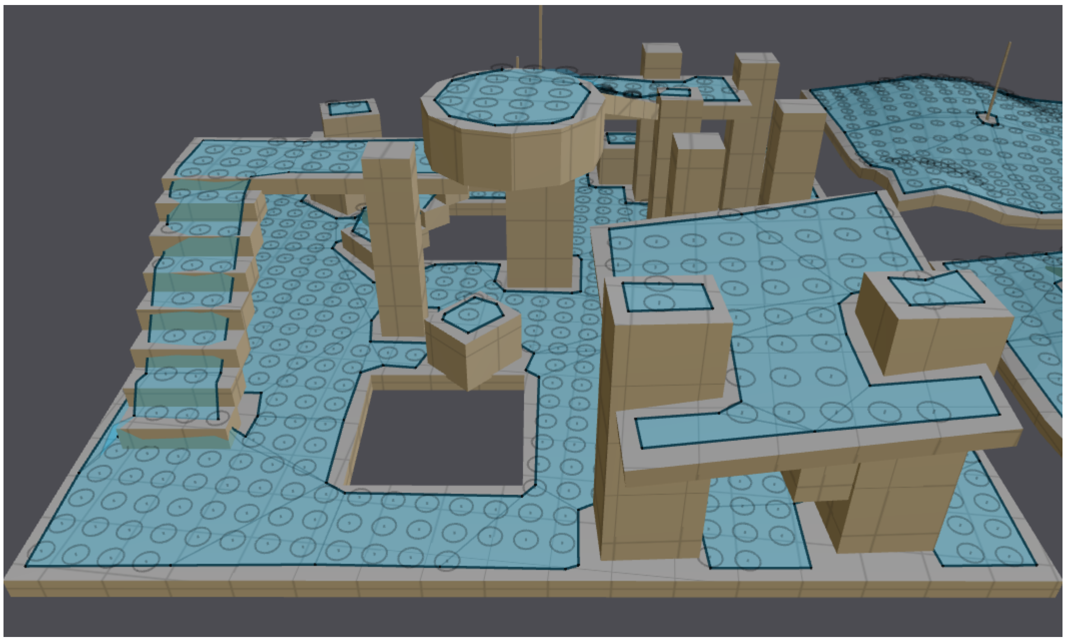


Figure 18: A screenshot from the Recast Demo program, in which we have implemented our method for generating jump links. It shows a set of sample points that is generated for a virtual environment. These sample points are the bottom-left corners of the top faces of walkable voxels, whose grid coordinates are divisible by a user-defined step size $d \in \mathbb{N}$. The sample points are indicated by horizontal circles, which are centered at the sample points.

top face of each walkable span $s \in D$, to the polygon in the navigation mesh that was derived from s, and that contains the projection of $\mathbf{c}_s$ onto $\mathbb{R}^2$. During our sampling procedure, we then check if a sampled corner maps to a polygon of the navigation mesh, and we add the corner to our list of sample points only if the mapping exists. For more information on the detail mesh and the algorithms for constructing it, we refer the reader to [9, 20, 22].

5.4 Finding Landing Voxels

In this section, we present our algorithm for finding a list of *land voxels* $V_\mathbf{s}$ for each of our sample points $\mathbf{s} \in S$. Together, the land voxels in $V_\mathbf{s}$ will form a representation of the land positions of the valid jumps that start at $\mathbf{s}$ (Figure 19). Later, we will use these land voxels to generate land segments for our jump links. In Section 5.4.1, we will discuss some concepts and notations that are used by our algorithm. Then, in Section 5.4.2, we will explain the steps of our algorithm.

5.4.1 Preliminaries

The input of our algorithm for finding land voxels consists of a sample point $\mathbf{s} \in S$, the dilated heightfield D, and the agent model A. Recall that we assume that A only contains one configuration $\{d, h\}$ of the agents' jump parameters. This means that we assume that the jumps that the agents can perform all have jump distance d, and jump height h (Definition 4.3). For convenience, we define $J_\mathbf{s}$ as the set of all the jumps that start at $\mathbf{s}$, and that have d and h as their jump parameters.

Our algorithm operates on a two-dimensional square grid G that represents a top-down view on D (Figure 20). Each cell in this grid corresponds to a column in D, and thus represents a vertical strip of voxels. As we discussed in Section 5.3, our sample points are corners of the top faces of walkable spans in D, so it follows that $\mathbf{s}$ aligns with a vertex of G when viewed from above. For convenience, we will call this vertex the *origin* of G. We will denote

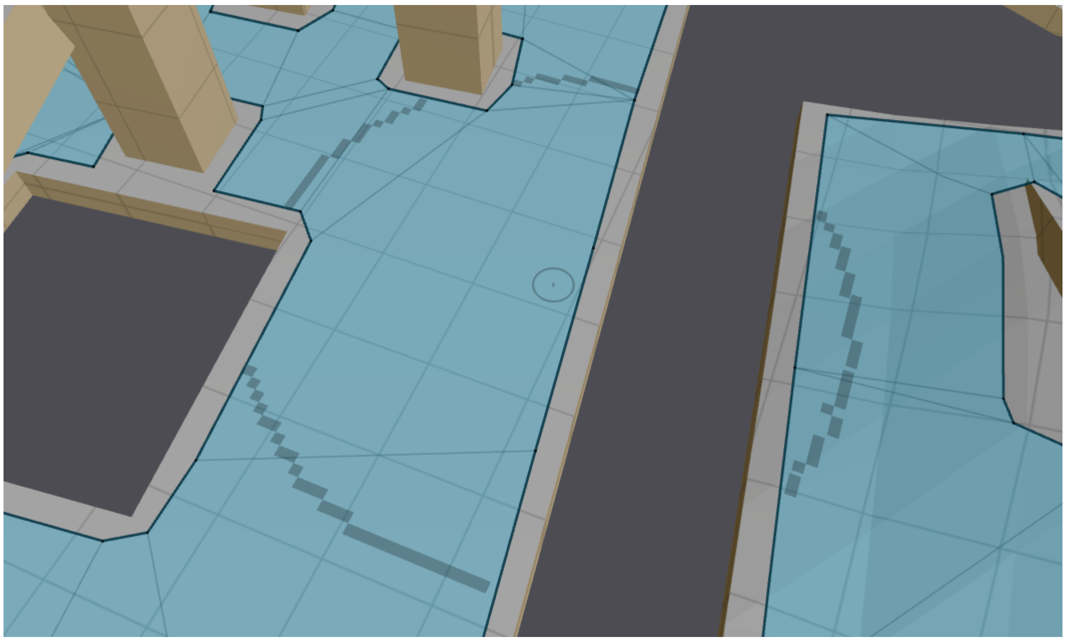


Figure 19: A screenshot from the Recast Demo program, in which we have implemented our method for generating jump links. It shows the set of land voxels corresponding to a sample point. The sample point is indicated by the gray circle, and the land voxels are indicated by gray squares.

each cell in G by $C_{i,j}$, $i, j \in \mathbb{Z}$, where i and j are the relative grid coordinates of the cell's bottom-left vertex with respect to the origin.

We will iterate over the cells in G (i.e. the columns in D), to search for walkable voxels that are intersected by trajectories of jumps in $J_\mathbf{s}$. We will only consider cells that are completely contained within a circle R, centered at the origin of G (Figure 20). The radius of this circle, which we call the *search range*, is defined as $r := \frac{1}{2}d(1 + \sqrt{m/h})$, where m is the maximum fall distance defined by A. It follows from Equation 1 (Definition 4.3) that r defines the maximum horizontal distance from $\mathbf{s}$ to the land position of a survivable jump in $J_\mathbf{s}$. That is, an agent will only be able to survive a jump $j \in J_\mathbf{s}$, if the horizontal distance between its land position and $\mathbf{s}$ is less or equal to r. Therefore, we will not search through cells that lie (partially) outside of R, because it is not guaranteed that an agent will be able to survive jumps to the walkable voxels that their columns may contain.

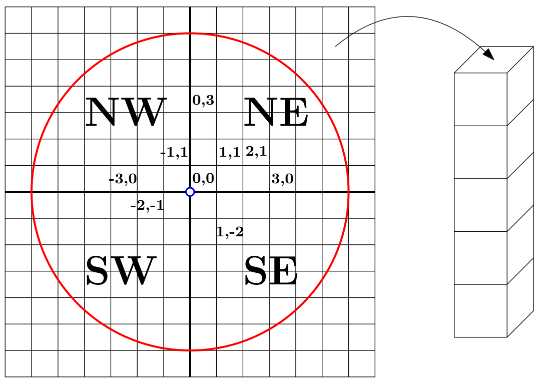


Figure 20: A cutout of the two-dimensional square grid G that represents a top-down view on D. Each cell in this grid represents a column in D, which is a vertical strip of voxels. The blue circle represents our sample point $\mathbf{s}$. The grid coordinates of some of the cells in G are written inside the cells. The red circle is the circle R, representing the maximum horizontal distance between $\mathbf{s}$ and the land position of a survivable jump in $J_\mathbf{s}$. Finally, the four quadrants are labeled as "NE", "NW", "SW", and "SE", indicating the north-east, the north-west, the south-west, and the south-east quadrant, respectively.

We will divide the cells in G into four disjoint sets, which we call *quadrants*. We define the *north-east* quadrant as the set of cells that lie to the north-east of the origin of G (i.e. the cells with positive ij-coordinates). The other three quadrants, which we call the *north-west*, the *south-west*, and the *south-east* quadrant, are defined analogously (Figure 20). We will iterate over the cells in G on a per-quadrant basis. That is, we will first iterate over the cells in the north-east quadrant, then over the cells in the north-west quadrant, then over the cells in the south-west quadrant, and finally over the cells in the south-east quadrant.

Suppose that we are iterating over the cells in the north-east quadrant. For each cell $C_{i,j}$ that we consider during this search, we will iterate over the solid voxels in the corresponding column in D, and check if any of them are intersected by trajectories of jumps in $J_{\mathbf{s}}$. When we encounter a solid voxel v that is intersected by trajectories, we compute two half lines. The first of these half lines F_{tl} starts at the origin of G, and passes through the top-left corner of $C_{i,j}$. The second half line F_{br} starts at the origin of G, and passes through the bottom-right corner of $C_{i,j}$ (Figure 21a). Obviously, any trajectory of a jump in $J_{\mathbf{s}}$ that intersects v must lie 'between' these half lines when projected onto the ground plane (Figure 21b). However, this does not mean that *every* trajectory, whose projection lies between the half lines, intersects v. Nevertheless, we will use the area that is spanned by F_{tl} and F_{br} as a representation of the set of trajectories that intersect v. We will call the subset of this area, that lies 'behind' v's grid cell, the *shadow* of v (Figure 21c). During the rest of the search, we can then ignore all the grid cells that lie completely within this shadow. In Figure 21c, for example, this means that we can ignore the cell $C_{3,3}$ (i.e. the cell that lies directly to the north-east of the gray cell $C_{2,2}$), because we assume that any trajectory that passes through its column will hit the solid voxel in $C_{2,2}$'s column before hitting any of the voxels in $C_{3,3}$'s column.

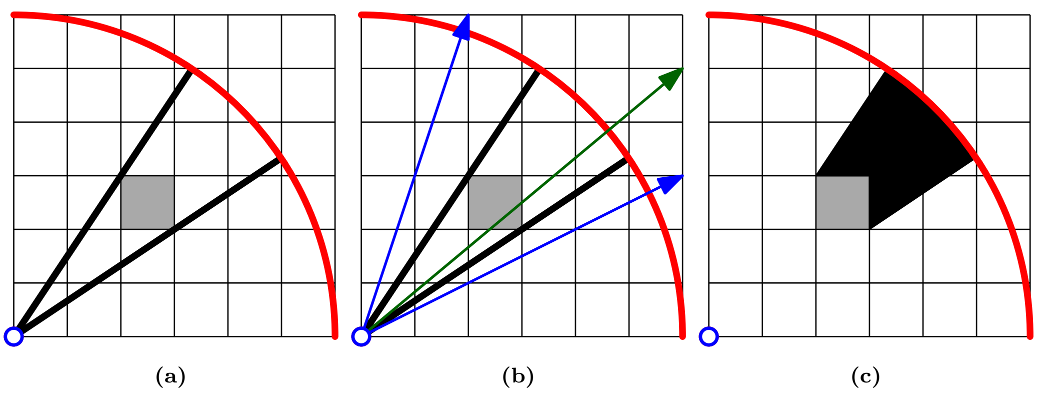


Figure 21: A cutout of the north-east quadrant. The blue circle represents the origin of G. The red circular arc represents R. The gray cell $C_{2,2}$ represents a column in D, containing a solid voxel v, that is intersected by trajectories of jumps in $J_{\mathbf{s}}$. (a) The two black line segments represent F_{tl} and F_{br}. (b) The three arrows represent the projections of trajectories of jumps in $J_{\mathbf{s}}$ onto the ground plane. Obviously, the two blue trajectories cannot intersect v, because they do not pass through the column of $C_{2,2}$. Only the trajectories, whose projections lie between F_{tl} and F_{br} (such as the green trajectory), have the potential to intersect v. (c) The black area represents the shadow of v. The cell $C_{3,3}$ lies completely within this shadow. This means that we can ignore $C_{3,3}$ in the remainder of the search, because we assume that any trajectory that passes through its column will hit v before hitting any of the voxels that this column may contain.

Our algorithm uses the shadow concept described above to check if jumps to walkable voxels are collision free. The order in which we iterate of the grid cells will be defined, such that voxels that are the first to be intersected by trajectories of jumps in $J_{\mathbf{s}}$, are the first to be encountered during the search. Whenever we encounter a solid voxel that is intersected by trajectories, we will compute its shadow, and we will stop searching through cells that lie completely within this shadow. Whenever we encounter a walkable voxel w, we will check if its grid cell overlaps with the shadows of any other solid voxels. If this cell lies completely outside these shadows, then we can be sure that every jump from $\mathbf{s}$ to w is collision free.

5.4.2 Algorithm

Our algorithm for finding land voxels consists of four sub-algorithms. Each of these sub-algorithms operates on one the four quadrants. We will first discuss the sub-algorithm that handles the north-east quadrant, and discuss the algorithms that handle the other three quadrants later. The algorithm that operates on the north-east quadrant (Algorithm 1) is a *recursive* algorithm, which means that it makes calls to itself. The input consists of a list of land voxels $V_{\mathbf{s}}$, a start cell C_{start}, and two two-dimensional vectors $\mathbf{a}_b, \mathbf{a}_t \in \mathbb{N} \times \mathbb{N}$ with positive integer coordinates. In the initial call, the list of land voxels $V_{\mathbf{s}}$ is set to an empty list, the start cell C_{start} is set to $C_{0,0}$, $\mathbf{a}_b$ is set to $(1, 0)$, and $\mathbf{a}_t$ is set to $(0, 1)$.

Let A_t be the half line that starts at the origin of G, and passes through the point $\mathbf{a}_t$. Similarly, let A_b be the half line that starts at the origin of G, and passes through $\mathbf{a}_b$. Together with R, these half lines form the boundary of an area that has the shape of a piece of pie (Figure 22a). This area, which we call the *search area*, contains the set of grid cells that will be searched by the current instance of the algorithm. Notice, in the initial call to the algorithm, that the search area contains all the cells in the north-east quadrant that lie within the search range r.

Starting at the start cell C_{start}, we iterate over the cells in the north-east quadrant from left to right. When we encounter a cell that either lies completely below A_b, or lies partially outside of R, we move up one row in G. We then iterate over the cells in this row from left to right, starting at the left-most cell that lies (partially) below A_t, and lies completely inside R. If no such cell exists, then we terminate the current instance of the algorithm. For each cell that we encounter, we iterate over the solid voxels in its corresponding column from top to bottom, and check if any of them are intersected by trajectories of jumps in $J_{\mathbf{s}}$.

Suppose that in the column of the cell $C_{i,j}$, we encounter a solid voxel v that is intersected by trajectories of jumps in $J_{\mathbf{s}}$. We then perform the following steps. We first check is v is a walkable voxel. If this is the case, then we add v to $V_{\mathbf{s}}$ if all of the following conditions are met:

1. Any jump $j \in J_{\mathbf{s}}$, whose trajectory intersects v, must hit v 'on the way down'. That is, if $j(t_v)$, with $t_v := \min\{t \in (t_{start}, \infty) \mid j(t) \in v\}$), is the first intersection point of j's trajectory with v, then $t_v > t_{start} + d/2$.
2. $C_{i,j}$ must lie completely below A_t, and must lie completely above A_b. Otherwise, $C_{i,j}$ lies (partially) inside the shadow of a solid voxel, so it is not guaranteed that every jump from $\mathbf{s}$ to v is collision free.
3. The midpoint of v must be a point in the navigation mesh. That is, the navigation mesh must contain a polygon that was derived from v, and that contains the projection of the midpoint of v onto the ground plane. This condition is necessary to make sure that the land segments of our jump links will lie inside the navigation mesh, when projected onto the ground plane. We can use similar techniques as described in Section 5.3, to determine if the midpoint of v is a point in the navigation mesh.

Now, if $C_{i,j}$ lies completely above A_b, and if the cell $C_{i+1,j}$ lies completely inside R, then there are still unvisited cells that lie inside the pie-shaped area defined by A_b and the half line that starts at the origin of G, and passes through the bottom-right corner of $C_{i,j}$. In this case, we create a new instance of the algorithm that will search through the cells that lie within this area. That is, we call Algorithm 1, setting $C_{start} = C_{i+1,j}$, $\mathbf{a}_t = (i + 1, j)$, and leaving $\mathbf{a}_b$ as it was (Figure 22b).

Let $C_{k,j+1}$ be the left-most cell that lies in the row above $C_{i,j}$, and lies (partially) below A_t. If $C_{i,j}$ lies completely below A_t, and if $C_{k,j+1}$ lies completely inside R, then there are still unvisited cells that lie inside the pie-shaped area defined by A_t and the half line that starts at the origin of G, and passes through the top-left corner of $C_{i,j}$. In this case, we create a new instance of the algorithm that will search through the cells that lie within this area.

That is, we call Algorithm 1, setting $C_{start} = C_{k,j+1}$, $\mathbf{a}_b = (i, j+1)$, and leaving $\mathbf{a}_t$ as it was (Figure 22c).

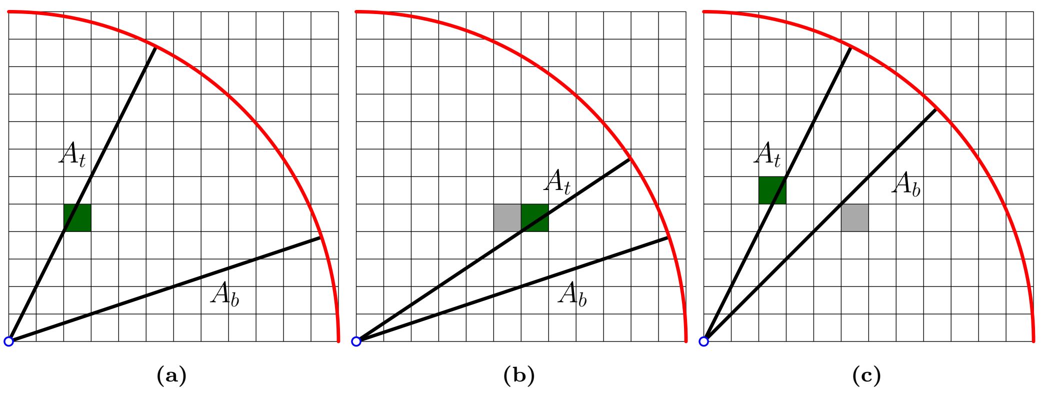


Figure 22: A cutout of the north-east quadrant. The blue circle represents the origin of G. The red circular arc represents R. (a) An example of what the input of a call to Algorithm 1 may look like. The green cell represents the start cell C_{start}, and the black line segments represent A_t and A_b. We will only iterate over cells that lie (partially) below A_t, lie (partially) above A_b, and lie completely inside R. (b) Suppose that the column of the gray cell $C_{5,4}$ contains a solid voxel that is intersected by trajectories of jumps in $J_\mathbf{s}$. We then create a new instance of the algorithm, setting C_{start} to the green cell $C_{6,4}$, setting A_t to the half line through the bottom-left corner of the gray cell, and leaving A_b as it was. (c) Similarly, we create a new instance of the algorithm, setting C_{start} to the green cell $C_{2,5}$, setting A_b to the half line through the top-right corner of the gray cell, and leaving A_t as it was.

After the potential recursion calls described above, we terminate the current instance of the algorithm. The pseudo code of the algorithm is given in Algorithm 1. Notice, in the pseudo code, that we perform the first potential recursion call, before potentially adding the current voxel to $V_\mathbf{s}$. This means that the algorithm will return the land voxels in a 'counter-clockwise' order. That is, the land voxels in $V_\mathbf{s}$ will be ordered by the angles, with respect to the positive x-axis, of the vectors pointing from the origin of G to the midpoints of the corresponding grid cells of the land voxels. In Section 5.5, we will see that having the land voxels ordered this way, will allow us to generate an initial set of line segments for the land-segment generation in linear time.

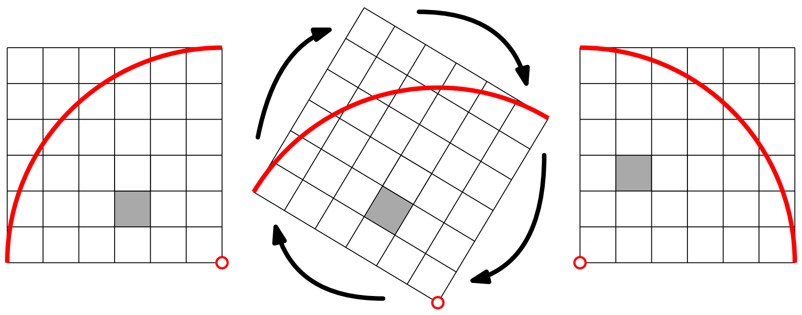


Figure 23: Visualization of the rotation of the north-west quadrant, so that it aligns with the north-east quadrant. Notice that the gray cell $C_{1,2}$ in the north-east quadrant, corresponds to the gray cell $C_{-3,1}$ in the north-west quadrant. When we run Algorithm 1 on the north-west quadrant, and we need to access the column that corresponds to $C_{1,2}$ (line 6), then we will access the column that corresponds to $C_{-3,1}$.

Let us now discuss how we handle the other three quadrants (i.e. the north-west, the south-west, and the south-east quadrant). The sub-algorithms that operate on these quadrants are 'rotated' versions of Algorithm 1. Informally, this means that we first rotate each of the three quadrants around the origin of G until they align with the north-east quadrant, and then run Algorithm 1 on the rotation of each quadrant (Figure 23). We thus use the algorithm designed for the north-east quadrant, exactly as it is described in Algorithm 1, to find land voxels in the other three quadrants as well. In practice, this means that we need to transform the coordinates of a cell in the north-east quadrant, in order to access the proper column in D (line 6 of Algorithm 1). For example, when we run the algorithm on the north-west quadrant, and we need to access the column corresponding to the cell $C_{1,2}$, then we 'actually' need to access the column that corresponds to the cell $C_{-3,1}$. Similarly, when we run the algorithm on the south-east quadrant, and we need to access the column corresponding to the cell $C_{1,2}$, then we 'actually' need to access the column that corresponds to the cell $C_{-2,2}$, etc. Our main algorithm for finding land voxels uses this approach to process the four quadrants in counter-clockwise order, starting at the north-east quadrant. That is, we first call Algorithm 1 on the north-east quadrant, then on the north-west quadrant, then on the south-west quadrant, and finally on the south-east quadrant.

Algorithm 1 `FindLandVoxels`

```
Input: A list of land voxels $V_{\mathbf{s}}$, a start cell $C_{a,b}$, and two vectors $\mathbf{a}_t, \mathbf{a}_b \in \mathbb{N} \times \mathbb{N}$.
1: $A_t \leftarrow$ half line that starts at $(0,0)$, and passes through $\mathbf{a}_t$
2: $A_b \leftarrow$ half line that starts at $(0,0)$, and passes through $\mathbf{a}_b$
3: $i \leftarrow a$
4: $j \leftarrow b$
5: while true do
6:     $C \leftarrow$ the column in $D$ corresponding to $C_{i,j}$
7:     for all solid voxels $v \in C$ do
8:         if $v$ is intersected by trajectories of jumps in $J_{\mathbf{s}}$ then
9:             if $C_{i,j}$ lies completely above $A_b$ and $C_{i+1,j}$ lies completely inside $R$ then
10:                FindLandVoxels($V_{\mathbf{s}}$, $C_{i+1,j}$, $(i+1, j)$, $\mathbf{a}_b$)
11:            if $v$ is walkable and $v$ satisfies the three conditions mentioned above then
12:                Add $v$ to $V_{\mathbf{s}}$.
13:            $C_{k,j+1} \leftarrow$ the left-most cell in the row $j+1$ that lies (partially) below $A_t$
14:            if $C_{i,j}$ lies completely below $A_t$ and $C_{k,j+1}$ lies completely inside $R$ then
15:                FindLandVoxels($V_{\mathbf{s}}$, $C_{k,j+1}$, $\mathbf{a}_t$, $(i, j+1)$)
16:            return
17:    if $C_{i+1,j}$ lies (partially) above $A_b$ and $C_{i+1,j}$ lies completely inside $R$ then
18:        $i \leftarrow i+1$
19:        continue
20:    $C_{k,j+1} \leftarrow$ the left-most cell in the row $j+1$ that lies (partially) below $A_t$
21:    if $C_{k,j+1}$ lies completely inside $R$ then
22:        $j \leftarrow j+1$
23:        $i \leftarrow k$
24:    else
25:        return
```

5.5 Constructing Land Segments

In this section, we discuss our method for computing the land segments of our jump links. The input of this method consists of a sample point $\mathbf{s} \in S$, and a corresponding list of land voxels $V_{\mathbf{s}} = \{v_0, v_1, \ldots, v_n\}$, computed by the algorithm discussed in Section 5.4. The output is a set of line segments L. These line segments, which we call *land segments*, will represent the land positions of the valid jumps that start at $\mathbf{s}$. For each land segment $l \in L$, we will create a jump link object that consists of $\mathbf{s}$ and l. In Section 5.3, we showed that the sample point $\mathbf{s}$ is a point in the navigation mesh, which means that its projection onto the ground plane lies inside a polygon of the navigation mesh. When projected onto the ground plane, each land segment that we generate for $\mathbf{s}$ will also lie inside a polygon of the navigation mesh, and it will be a different polygon than the one that contains the projection of $\mathbf{s}$. Therefore, each jump link that we generate will represent a set of valid jumps from one polygon of the navigation mesh to another (Figure 24).

To obtain an initial set of line segments, we connect the midpoints of the *neighboring* land voxels in $V_{\mathbf{s}}$. That is, we connect the midpoints of each two consecutive land voxels $v_i, v_{i+1} \in$

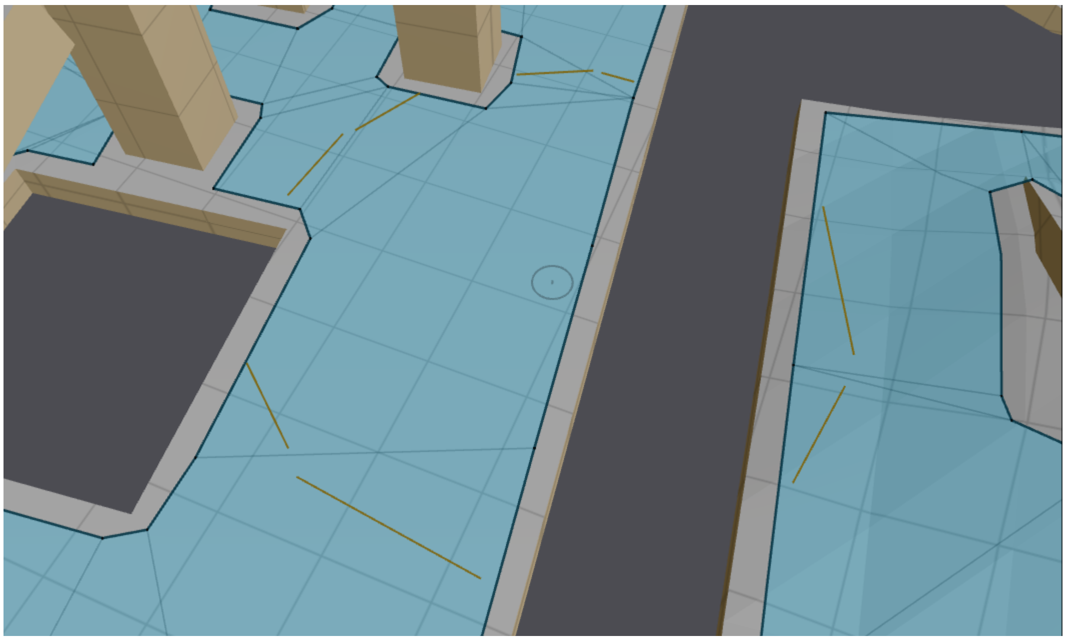


Figure 24: A screenshot from the Recast Demo program, in which we have implemented our method for generating jump links. It shows the set of land segment that we generated using the set of land voxels from Figure 19. Notice that each land segment is completely contained within a single polygon of the navigation mesh.

$V_{\mathbf{s}}$, if they are part of two neighboring walkable spans in the dilated heightfield D, or if their spans lie in columns that are diagonally adjacent in D, and share at least one neighbor span. We want each line segment to lie inside a single polygon of the navigation mesh when projected onto the ground plane. Therefore, we will not connect the midpoints of two land voxels, if these midpoints represent positions in different polygons of the navigation mesh. We will ignore land voxels whose midpoints represent positions in the polygon that contains the projection of $\mathbf{s}$. The result is a set of polygonal chains C. When projected onto the ground plane, each polygonal chain in C will lie completely inside a single polygon of the navigation mesh.

The polygonal chains in C consists of many vertices (the midpoints of the land voxels), and many small line segments. Creating a jump link for each of those line segments would be costly in terms of storage, and would result in long path-planning times. Therefore, before generating the land segments, we will first *simplify* the polygonal chains. That is, for each chain $c \in C$, we will compute a second chain c', that is similar to c, but is defined only by a subset of the vertices in c. This will give us a set of simplified chains C', consisting of fewer, and longer line segments that the original chains in C. The line segments in the chains in C' will be the land segments of our jump links.

Algorithms for simplifying polygonal chains are often called *line simplification algorithms*. The input of such an algorithm consists of polygonal chain c, defined by a finite sequence of vertices $V = \{v_0, v_1, \ldots, v_n\}$. The output is another polygonal chain c' that is similar to c, but is defined only by a subsequence V' of V. Shi and Cheung [24] have shown that the line simplification algorithm that generally produces the best results is the *Douglas-Peucker* (DP) algorithm [5]. Given a sequence of vertices $V = \{v_0, v_1, \ldots, v_n\}$, the DP algorithm first creates the subsequence $V' = \{v_0, v_n\}$. It then finds the vertex $v_i \in \{v_1, v_2 \ldots, v_{n-1}\}$ that has the largest distance to the line segment from v_0 to v_n, and inserts it into V'. Then, it finds the vertices $v_j \in \{v_1, v_2 \ldots, v_{i-1}\}$ and $v_k \in \{v_{i+1}, v_{i+2}, \ldots, v_{n-1}\}$ that have the largest distances to the line segments from v_0 to v_1, and from v_1 to v_n, respectively, and inserts them into V'. This process is repeated until V contains no more vertices, whose distances to the polygonal chain defined by V' are larger than a user-defined threshold distance ϵ.

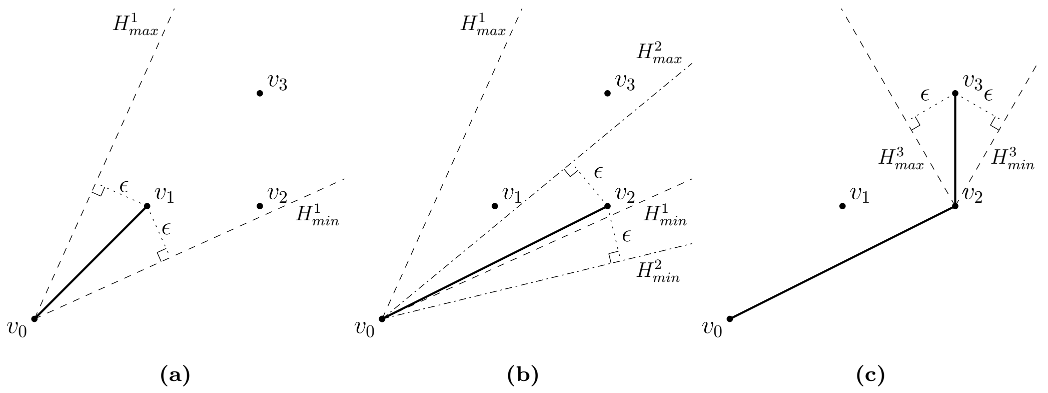


Figure 25: Example of how the ZS algorithm computes a polygonal chain, given a sequence of four vertices $V = \{v_0, v_1, v_2, v_3\} \subset \mathbb{R}^2$. (a) The algorithm begins with a polygonal chain defined by the subsequence $V' = \{v_0, v_1\}$. It computes two half lines, H^1_{min} and H^1_{max}, that start at v_0, and whose distances to v_1 are equal to ϵ. The angle interval $I = [\alpha_{min}, \alpha_{max}]$ is then defined by setting α_{min} to the angle of H^1_{min} (with respect to the positive x-axis), and setting α_{max} to the angle of H^1_{max}. Since v_2 lies 'between' H^1_{min} and H^1_{max}, it follows that the distance between v_1 and the line segment $l_{0,2}$ from v_0 to v_2 is less than ϵ. Therefore, in the first iteration of the algorithm, v_2 replaces v_1 in V', yielding $V = \{v_0, v_2\}$. (b) After v_2 is processed, the algorithm computes the half lines H^2_{min} and H^2_{max} that start at v_0, and whose distances to v_2 are equal to epsilon. Since the angle β_{max} of H^2_{max} is smaller than α_{max}, β_{max} replaces α_{max} in I (α_{min} stays the way it was, since the angle β_{min} of H^2_{min} is smaller than α_{min}). Since v_3 lies above H^2_{max}, it follows that the distance between v_2 and the line segment $l_{0,3}$ from v_0 to v_3 is larger than ϵ. Therefore, in the second iteration of the algorithm, v_3 is added to V', yielding $V' = \{v_0, v_2, v_3\}$. (c) The final polygonal chain computed by the ZS algorithm. Notice that the original chain defined by V is simplified, by removing the vertex v_1. If V would contain additional vertices, then the algorithm would continue by replacing α_{min} and α_{max} with the angles of the half lines H^3_{min} and H^3_{max}, respectively.

We can use the DP algorithm to generate land segments for our jump links, by simplifying the polygonal chains that are obtained by connecting the midpoints of neighboring land voxels in $V_{\mathbf{s}}$. However, an alternative, and faster approach is to use the Zhao-Saalfeld (ZS) algorithm [33]. Given a sequence of vertices $V = \{v_0, v_1, \ldots, v_n\}$, the ZS algorithm first creates the subsequence $V' = \{v_0, v_1\}$. It then iterates over the remaining vertices in V in the order defined by V. For each vertex v_j, it checks if any of the vertices in $V_{i+1,j-1} = \{v_{i+1}, v_{i+2}, \ldots, v_{j-1}\}$, where v_i is the second-to-last vertex in V', has a horizontal distance to the line segment $l_{i,j}$ from v_i to v_j that is larger than a user-defined threshold distance ϵ. If there exists such a vertex, then v_j is added to V'. Otherwise, v_j replaces the last vertex in V'. To check if $V_{i+1,j-1}$ contains a vertex that has a horizontal distance to $l_{i,j}$ that is larger than ϵ, the algorithm maintains an interval of angles $I = [\alpha_{min}, \alpha_{max}] \subset [0, 2\pi]$. If the angle of the projection of $l_{i,j}$ onto $\mathbb{R}^2$ (with respect to the positive x-axis) lies outside this interval, then $V_{i+1,j-1}$ contains a vertex whose horizontal distance to $l_{i,j}$ is larger than ϵ, and v_j is added to V'. Otherwise, $V_{i+1,j-1}$ contains no such vertices, and the last vertex in V' is replaced by v_j. After the vertex v_j is processed, I is updated as follows. First the angle β_{min} is calculated, which defines the minimum angle (with respect to the positive x-axis) of any half line $H \subset \mathbb{R}^2$ that starts at the xy-position of the second-to-last vertex in V', and for which the horizontal distance between v_j and H is less than or equal to ϵ. Similarly, β_{max} is calculated, which defines the maximum angle of any half line $H \subset \mathbb{R}^2$ that starts at the xy-position of the second-to-last vertex in V', and for which the distance between v_j and H is less than or equal ϵ. If β_{min} is larger than α_{min}, or if v_j was added to V' (and did not replace the last vertex of V'), then β_{min} replaces α_{min} in I. Similarly, if β_{max} is smaller than α_{max}, or if v_j was added to V', then β_{max} replaces α_{max} in I. By maintaining the angle interval I, the algorithm can check if $V_{i+1,j-1}$ contains any vertices that lie too far away from $l_{i,j}$ in constant time, which results in a total running time of $O(n)$. Figure 25 shows an example of the steps of the ZS algorithm.

Shi an Cheung have shown that the ZS algorithm generally produces slightly-less accurate results than the DP algorithm [24]. However, in the context of our algorithm for generating jump links, the ZS algorithm does have some advantages over the DP algorithm. First of all, the running time of the ZS algorithm is always $O(n)$, which is smaller than the running time of the DP algorithm, which is $O(n)$ in the best case, and $O(n \log n)$ in the worst case. Second, using the ZS algorithm gives us the opportunity to generate land segments for the sample point $\mathbf{s}$, *while* we search for land voxels for $\mathbf{s}$. That is, we can integrate the ZS algorithm into the algorithm for finding land voxels, by replacing line 12 of Algorithm 1 with the steps explained above. We can then obtain our set of land segments L directly, by connecting the midpoints of each two consecutive neighboring land voxels in the resulting list of land voxels $V_{\mathbf{s}}$. This means that we no longer have to store every land voxel that we find, which results in lower storage requirements for the main algorithm for generating jump links. It is for these reasons that we have chosen to use the ZS algorithm for generating the land segments.

6 Path Planning

In this section, we present our method for planning a path through a virtual environment, which can be traversed by combining walking and jumping motions. We assume that we have a navigation mesh that is generated with Recast, and that we have a set of jump links that is generated with the method discussed in Section 5. We present algorithms that can be used to to find a path between two positions in the navigation mesh, which can include the jumps that are encoded by the jump links (Figure 26).

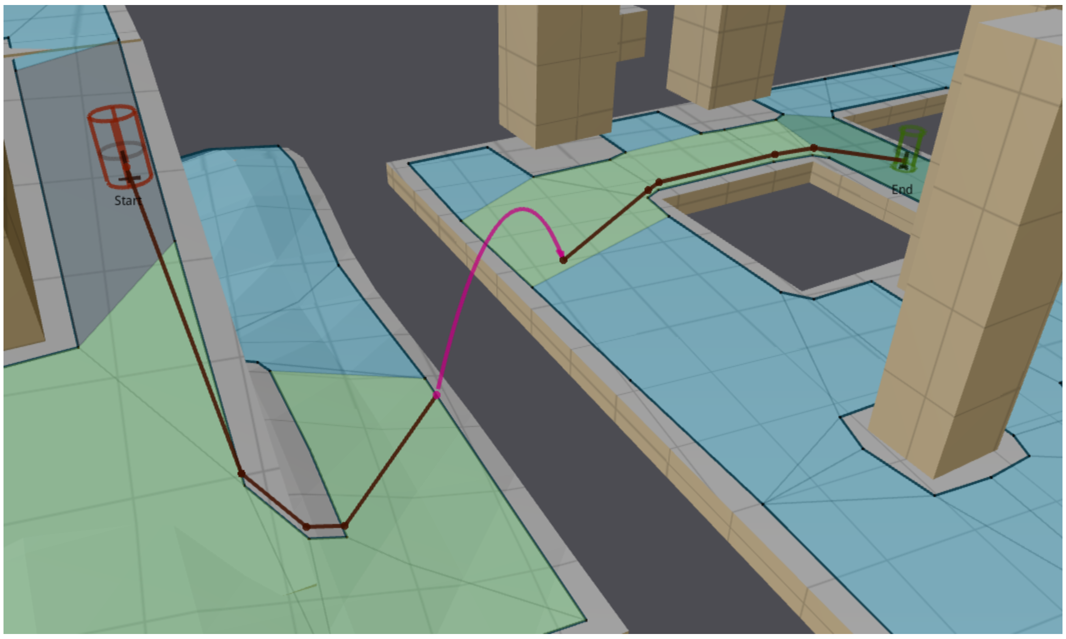


Figure 26: A screenshot from the Recast Demo program, which is the software in which we have implemented our methods. It shows a path that includes a jump between two platforms. The start position is indicated by the red cylinder in the left side of the figure, and the goal position is indicated by the green cylinder in the right side of the figure. The polygon that contains the start position is drawn in dark-red, and the polygon that contains the goal position is drawn in dark-green. The other polygons in the corridors are drawn in light-green. The local path is indicated by dark-red line segments, and the jump in the path is indicated by the pink parabolic arc. In Section 7.2, we explain how we can derive this arc and the line segments from the two-dimensional polygonal chain that defines the local path.

Similar to Detour's path-planning method (Section 3.2), our method consists of two phases: a *global* planning phase, and a *local* planning phase. Our global path-planning algorithm is a modified version of Detour's global path-planning algorithm, as it is described in Section 3.2.1. It computes a *global path* between a start position and a goal position, which defines an ordered set of polygons in the navigation mesh. This ordered set of polygons defines an ordered set of corridors, in which each two consecutive corridors are connected by at least one jump link. This sequence of corridors forms the input of our local path-planning algorithm, which is a modified version of Detour's local path-planning algorithm, as it is described in Section 3.2.2. This algorithm computes a *local path*, which is defined as a two-dimensional open polygonal chain that connects the projections of the start position and the goal position onto the ground plane. Some of the line segments in this chain represent paths through the walkable space of the environment, and others represent trajectories of jumps between two positions in the walkable space. We will discuss our global path-planning algorithm in Section 6.1, and discuss our local path-planning algorithm in Section 6.2. Finally, in Section 6.3, we will discuss how agents can use the local paths to navigate themselves to their goal positions.

6.1 Global Path Planning

In this section, we discuss our global path-planning algorithm, which is a modified version of Detour's global path-planning algorithm. The input of our algorithm consists of a start position $\mathbf{s} \in \mathbb{R}^3$, a goal position $\mathbf{g} \in \mathbb{R}^3$, a navigation mesh that is created with Recast, and a set of jump links J that is generated with the method discussed in Section 5.

Our global path-planning algorithm runs the A* algorithm on a navigation graph, which is an extended version of the navigation graph $G = \{V, E\}$ that is used by Detour. Recall, from Section 3.2.1, that the vertices in V are the centroids of the polygons in the navigation mesh, and that a pair of vertices is connected by an edge in E if, and only if, their polygons share an edge in the navigation mesh. We will call the edges in E the *walkable edges*, because each of them represents a walkable path between two polygons in the navigation mesh. We will extend G by adding extra edges to E, which we call *jumpable edges.* We add a jumpable edge $e_{i,j}$ from a vertex $v_i \in V$ to a another vertex $v_j \in V$ if, and only if, J contains at least one jump link, whose start position lies inside the polygon of v_i, and whose land segment lies inside the polygon of v_j, when projected onto the ground plane. The jumpable edges are *directed edges*, in contrast to the walkable edges, which are all undirected. In other words, if there exists a jumpable edge $e_{i,j}$ from v_i to v_j, then an agent can jump from v_i's polygon to v_j's polygon, but not necessarily the other way around. It follows that a vertex $v_j \in V$ is a neighbor of a vertex $v_i \in V$ if, and only if, there exists a jumpable edge $e_{i,j} \in E$ from v_i to v_j, or if E contains a walkable edge that connects the two vertices. We define the length of a jumpable edge $e_{i,j}$ as $length(e_{i,j}) := ||e_{i,j}|| + c$, where $||e_{i,j}||$ is the Euclidean length of the edge, and $c \in \mathbb{R}^+$ is a user-defined constant. This constant, which we call the *jump cost*, represents the agents' preference between walking and jumping. An agent will prefer to jump from one polygon to another, if the length of the shortest walkable path in the graph is larger than the distance between the centroids of the polygons plus the jump cost.

Our global planning algorithm computes a global path by performing the A* algorithm on the navigation graph described above. The vertices in this path define a sequence polygons, in which the first polygon contains the projection of $\mathbf{s}$ onto $\mathbb{R}^2$, and in which the last polygon contains the projection of $\mathbf{g}$ onto $\mathbb{R}^2$. This sequence of polygons defines a sequence of corridors $C = \{C_0, C_1, \ldots, C_n\}$, in which each pair of consecutive corridors is connected by at least one jump link. That is, for each two consecutive corridors $C_i, C_{i+1} \in C$, there will be at least one jump link in J, whose start position lies inside the last polygon of C_i, and whose land segment lies inside the first polygon of C_{i+1}, when projected onto the ground plane. This sequence of corridors forms the input of our local path-planning algorithm, which we discuss in the next section.

6.2 Local Path Planning

In this section, we discuss our local path-planning algorithm, which is a modified version of Detour's local path-planning algorithm. The input of our algorithm consists of a start position $\mathbf{s}' \in \mathbb{R}^2$, a goal position $\mathbf{g}' \in \mathbb{R}^2$, a set of jump links J, and a sequence of corridors

$C = \{C_0, C_1, \ldots, C_n\}$ [6]. It is assumed that $\mathbf{s}'$ lies in the first polygon of C_0, that $\mathbf{g}'$ lies in the last polygon of C_n, and that for each two consecutive corridors $C_i, C_{i+1} \in C$, J contains at least one jump link, whose start position lies inside the last polygon of C_i, and whose land segment lies inside the first polygon of C_{i+1} when projected onto the ground plane. The output is a local path from $\mathbf{s}'$ to $\mathbf{g}'$, which is an open polygonal chain defined by an ordered set of vertices $V = \{v_0, v_1, \ldots, v_m\} \subset \mathbb{R}^2$, with $v_0 = \mathbf{s}'$ and $v_m = \mathbf{g}'$. If C contains more than one corridor, then V will include pairs of consecutive vertices that represent jumps between consecutive pairs of corridors. The first vertex in such a pair will represent the start position of a jump, and the second vertex will represent its land position. The line segments in the local path that connect such vertices will thus represent the trajectories of the jumps in the path. All the other line segments in the local path will represent the walkable parts of the path.

Notice that J may contain multiple jump links that connect two consecutive corridors $C_i, C_{i+1} \in C$. That is, there may be multiple jump links in J, whose start positions lie in the last polygon of C_i, and whose land segments lie in the first polygon of C_{i+1}, when projected onto the ground plane. One of the tasks of the local planning algorithm is to decide which one of those jump links will be incorporated into the local path. Before we explain how this jump link is selected (Section 6.2.2), we will first explain the other steps of the algorithm (Section 6.2.1).

6.2.1 The Modified Funnel Algorithm

In this section, we show how our local planning algorithm works, under the assumption that we already know which jump links in J will be incorporated into the local path. We thus assume that we have selected a set of jump links $J' = \{j_0, j_1, \ldots, j_{n-1}\} \subseteq J$, in which each link j_i connects the two consecutive corridors C_i and C_{i+1} in C. That is, if j_i is a jump link in J', then j_i's start position will lie inside the last polygon of C_i, and its land segment will lie inside the first polygon of C_{i+1} when projected onto the ground plane. For convenience, we will treat the jump links in J' as two-dimensional objects in the remainder of this section. When we speak of the start position of a jump link $j_i \in J'$, we will thus refer to its projection onto $\mathbb{R}^2$. Similarly, when we speak of j_i's land segment, we will refer to the projection of the land segment onto $\mathbb{R}^2$.

As we mentioned before, our local planning algorithm is a modified version of Detour's local planning algorithm, which is also known as the *simple stupid funnel algorithm* (SSFA) [19]. If C only contains a single corridor (which means that there will be no jumps in the local path), then our local planning algorithm is identical to the SSFA. Let us therefore assume that C contains multiple corridors. Our algorithm can then be summarized as follows. We first use the regular SSFA to find a local path from $\mathbf{s}'$ to the start position of the first jump link j_0. That is, we run the SSFA on the first corridor C_0, setting $\mathbf{s}'$ as the start position, and setting j_i's start position as the goal position. This gives us a set of vertices, defining an open polygonal chain that connects $\mathbf{s}'$ to the start position of j_0. We add these vertices to our output list of vertices V, and use a modified version of the SSFA to compute a local path from j_0's start position to the next *target position*, which is either the start position of the second jump link j_1, or, if C_1 is the last corridor in C, is the goal position $\mathbf{g}'$. We then add the vertices of this second path to V (with the exception the first vertex, which is j_i's start position), and compute the next local path to the next target position. We repeat this process until the goal position $\mathbf{g}'$ is added to V.

Suppose that we have computed a local path from $\mathbf{s}'$ to the start position of the jump link $j_i \in J'$. Our goal is then to find a local path from j_i's start position to the next target position. The first vertex of this path will j_i's start position, and the second vertex will be a point on j_i's land segment. This second vertex will represent the land position of the jump

[6] Here, $\mathbf{s}'$ and $\mathbf{g}'$ are the projections of $\mathbf{s}$ and $\mathbf{g}$ onto $\mathbb{R}^2$, respectively, where $\mathbf{s}$ and $\mathbf{g}$ are the input positions of the global planning algorithm, J is a set of jump links that is generated with the method described in Section 5, and C is the corridor that computed by our global planning algorithm.

from C_i to C_{i+1}. The last vertex of the local path will be the target position $\mathbf{t} \in \mathbb{R}^2$, which is either the start position of the next jump link j_{i+1}, or, if j_i is the last jump link in J', is the goal position $\mathbf{g}'$. We can distinguish the following cases:

1. C_{i+1} only contains one polygon
2. C_{i+1} contains multiple polygons

Let us first consider the first case, where C_{i+1} only contains one polygon. In this case, the target position $\mathbf{t}$ lies in the same polygon as the land segment of j_i. Let L be the line that passes through the endpoints of j_i's land segment. We can then decompose Case 1 into the following subcases:

(a) $\mathbf{t}$ lies on j_i's land segment.

(b) $\mathbf{t}$ lies on L (but not on j_i's land segment), or j_i's start position and $\mathbf{t}$ lie on opposite sides of L.

(c) j_i's start position and $\mathbf{t}$ lie on the same side of L.

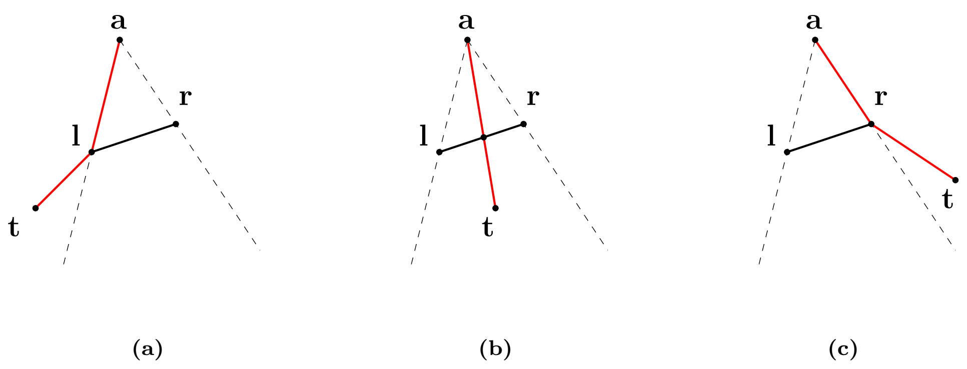


Figure 27: Examples of local paths computed for Case 1b. The point $\mathbf{a}$ is the apex of the funnel F, which is the start position of the jump link j_i. The points $\mathbf{l}$ and $\mathbf{r}$ are the left and right portal points of F, which are the left and right endpoints of j_i's land segment, respectively. The point $\mathbf{t}$ is the target position. The local path that is computed by the modified SSFA is shown in red. (a) $\mathbf{t}$ lies to the left of F. Therefore, in the first iteration of the SSFA, the algorithm is restarted at $\mathbf{l}$, which means that $\mathbf{l}$ is added as a vertex to the local path. In the second iteration, $\mathbf{t}$ is added as a vertex to the local path. (b) $\mathbf{t}$ lies inside F, so the SSFA adds $\mathbf{t}$ as a vertex to the local path. In this case, we modify the SSFA by including an intermediate step, in which the intersection of the line segment from $\mathbf{a}$ to $\mathbf{t}$ with j_i's land segment is added a vertex to the local path, before $\mathbf{t}$ is added to the local path. (c) $\mathbf{t}$ lies to the right of F. Therefore, in the first iteration of the SSFA, the algorithm is restarted at $\mathbf{r}$, which means that $\mathbf{r}$ is added as a vertex to the local path. In the second iteration, $\mathbf{t}$ is added as a vertex to the local path.

The first subcase is the simplest one. In this case, the local path from j_i's start position to $\mathbf{t}$ can be found by connecting these positions with a single line segment. Let us now consider the second subcase, in which $\mathbf{t}$ either lies on L (but not on j_i's land segment), or lies on the other side of the land segment than j_i's start position. Our goal is to find the *shortest* local path from j_i's start position to $\mathbf{t}$, whose second vertex is a point on j_i's land segment. We can find this path by using a modified version of the SSFA. We first create a funnel F, whose apex is j_i's start position, and whose left and right portal points are the left and right endpoints j_i's land segment, respectively[7]. We then apply the steps of the SSFA as they are described in Section 3.2.2, setting F as the initial funnel, setting j_i's start position as the start position, and setting $\mathbf{t}$ as the goal position. If $\mathbf{t}$ lies inside F, then we include an additional step, in which the intersection of the line segment from F's apex to $\mathbf{t}$ with j_i's land segment is added to V, before $\mathbf{t}$ is added to V (Figure 27b). In Figure 27, we show some examples of the resulting paths.

[7]Here, the left endpoint $\mathbf{l}$ and the right endpoint $\mathbf{r}$ of the land segment of a jump link are defined, such that the sequence $\{\mathbf{a}, \mathbf{l}, \mathbf{r}\}$ is ordered counter-clockwise, where $\mathbf{a}$ is the start position of the jump link.

Now, let us consider the third subcase, in which **t** lies on the same side of j_i's land segment as j_i's start position. Again, our goal is the find the shortest local path from j_i's start position to **t**, whose second vertex is a point on j_i's land segment. We can find this path by using a modified version of the SSFA. We first create a funnel F, whose apex is is the reflection of j_i's start position in L, and whose left and right portal points are the *right* and *left* endpoints of j_i's land segment, respectively. Then, we apply the steps of the SSFA, setting F as the initial funnel, setting j_i's start position as the start position, and setting **t** as the goal position. If **t** lies inside F, then we include an additional step, in which the intersection of the line segment from F's apex to **t** with j_i's land segment is added to V, before **t** is added to V (Figure 28b). In Figure 28, we show some examples of the resulting paths.

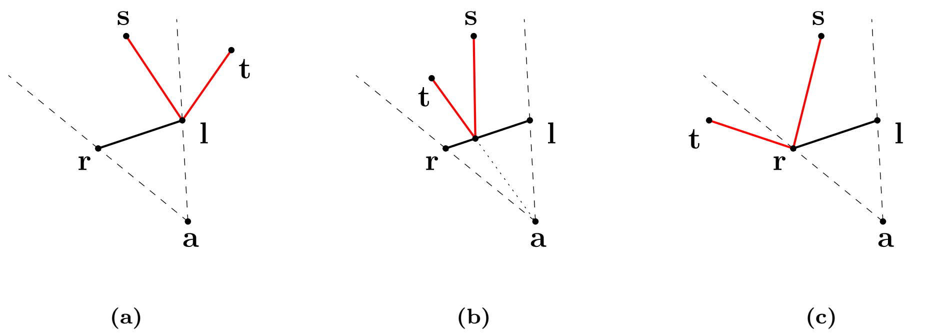


Figure 28: Examples of local paths computed for Case 1c. The point **a** is the apex of the funnel F, which is the reflection of the start position **s** of the jump link j_i in the line that passes through the endpoints of j_i's land segment. The points **l** and **r** are the left and right portal points of F, which are the *right* and *left* endpoints of j_i's land segment, respectively. The point **t** is the target position. The local path that is computed by the modified SSFA is shown in red. (a) **t** lies to the left of F. Therefore, in the first iteration of the SSFA, the algorithm is restarted at **l**, which means that **l** is added as a vertex to the local path. In the second iteration, **t** is added as a vertex to the local path. (b) **t** lies inside F, so the SSFA adds **t** as a vertex to the local path. In this case, we modify the SSFA by including an intermediate step, in which the intersection of the line segment from **a** to **t** with j_i's land segment is added a vertex to the local path, before **t** is added to the local path. (c) **t** lies to the right of F. Therefore, in the first iteration of the SSFA, the algorithm is restarted at **r**, which means that **r** is added as a vertex to the local path. In the second iteration, **t** is added as a vertex to the local path.

The following theorem shows that the local path, computed with the method described above, is indeed the shortest local path from j_i's start position to **t**:

Theorem 6.1. *Let $\mathbf{s} \in \mathbb{R}^2$ be the start position of a jump link, let $l \subset \mathbb{R}^2$ be the land segment of that jump link, and let $\mathbf{t} \in \mathbb{R}^2$ be the target position. Furthermore, let p be the local path from $\mathbf{s}$ to $\mathbf{t}$ that is computed by the local planning algorithm for Case 1. Then p is the shortest local path from $\mathbf{s}$ to $\mathbf{t}$ that has a vertex on l.*

Proof (outline): A local path from **s** to **t** is an open polygonal chain, defined by a finite sequence of vertices $\{v_0, v_1, \ldots, v_n\}$, with $v_0 = \mathbf{s}$ and $v_n = \mathbf{t}$. The length of such a path is defined as the sum of lengths of the line segments in the chain. In order to prove Theorem 6.1, we have to show that the local path p, computed by our local planning algorithm, has the smallest length of all the possible local paths from **s** to **t** that have an intermediate vertex on l.

Let L be the line that passes through the endpoints of l. We will not prove Theorem 6.1 for the case where **t** lies on L, or the case where the line that passes through **s** and **t** is orthogonal to L. The proof for these cases is relatively trivial, and we leave it as an exercise for the reader. We thus assume that **t** does not lie on L, and that the line that passes through **s** and **t** is not orthogonal to L. In this case, the path p that is computed by our local planning algorithm is defined by a sequence of three vertices $\{\mathbf{s}, \mathbf{v}, \mathbf{t}\}$, in which the second vertex **v** is point on l. Let **w** be a point on l. Obviously, the sequence $\{\mathbf{s}, \mathbf{w}, \mathbf{t}\}$

defines the shortest local path from $\mathbf{s}$ to $\mathbf{t}$ in which $\mathbf{w}$ is a vertex (any other path that contains additional vertices must be at least as long as this path). Therefore, it will be sufficient to show that the sequence $\{\mathbf{s}, \mathbf{v}, \mathbf{t}\}$ defines a shorter path from $\mathbf{s}$ to $\mathbf{t}$ than any other sequence $\{\mathbf{s}, \mathbf{w}, \mathbf{t}\}$, where $\mathbf{w} \neq \mathbf{v}$ is a point on l.

If $\mathbf{t}$ and $\mathbf{s}$ lie on opposite sides of L, then our algorithm sets the second vertex $\mathbf{v}$ of p to the point on l that lies closest to the intersection of the line segment from $\mathbf{s}$ to $\mathbf{t}$ with L. If, on the other hand, $\mathbf{t}$ and $\mathbf{s}$ lie on the same side of L, then $\mathbf{v}$ is set to the point on l that lies closest to the intersection of the line segment from $\mathbf{m}$ to $\mathbf{t}$ with L, where $\mathbf{m}$ is the reflection of $\mathbf{s}$ in L. Notice, in this case, that the length of the line segment that connects $\mathbf{s}$ to $\mathbf{v}$ is equal to the length of the line segment that connects $\mathbf{m}$ to $\mathbf{v}$ (Figure 29a). Therefore, the length of p is equal to the length of the local path defined by the sequence $\{\mathbf{m}, \mathbf{v}, \mathbf{t}\}$. Let us define the point $\mathbf{j}$ as follows: if $\mathbf{t}$ and $\mathbf{s}$ lie on opposite sides of L, then $\mathbf{j} := \mathbf{s}$, and if $\mathbf{t}$ and $\mathbf{s}$ lie on the same side of L, then $\mathbf{j} := \mathbf{m}$. The points $\mathbf{j}$ and $\mathbf{t}$ thus lie on opposite sides of L. Let $\mathbf{i}$ be the intersection point of the line segment from $\mathbf{j}$ to $\mathbf{t}$ with L. Then it follows that the second vertex $\mathbf{v}$ of p is the point on l that lies closest to $\mathbf{i}$, and that the length of p is equal to the length of the local path defined by the sequence $\{\mathbf{j}, \mathbf{v}, \mathbf{t}\}$.

Now, let $\mathbf{j}'$ be the projection of $\mathbf{j}$ onto L, and let T_j be the triangle defined by the points $\{\mathbf{j}, \mathbf{i}, \mathbf{j}'\}$. Let l_j denote the length of the hypotenuse of T_j (i.e. $l_j := ||\mathbf{i} - \mathbf{j}||$), and let α be the interior angle of T_j at $\mathbf{i}$. Since T_j is a right triangle, we can express the lengths of the other sides of T_j as $||\mathbf{j} - \mathbf{j}'|| = l_j \cdot \sin(\alpha)$ and $||\mathbf{i} - \mathbf{j}'|| = l_j \cdot \cos(\alpha)$. Similarly, let $\mathbf{t}'$ be the projection of $\mathbf{t}$ onto L, let T_t be the triangle defined by the points $\{\mathbf{t}, \mathbf{i}, \mathbf{t}'\}$, and let l_t denote the length of the hypotenuse of T_t. Notice, that the interior angle of T_t at $\mathbf{i}$ is equal to α (Figure 29b). Therefore, we can express the lengths of the other sides of T_t as $||\mathbf{t} - \mathbf{t}'|| = l_t \cdot \sin(\alpha)$ and $||\mathbf{i} - \mathbf{t}'|| = l_t \cdot \cos(\alpha)$.

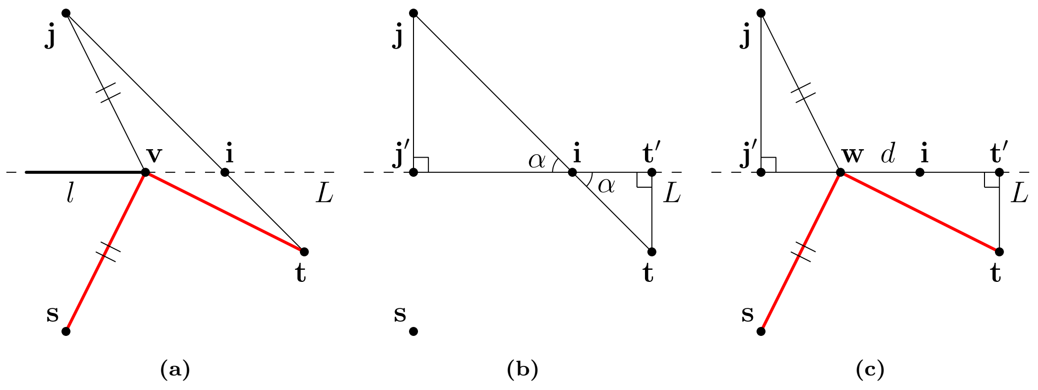


Figure 29: Supporting figures for the proof of Theorem 6.1. (a) The start position of the jump link is the point $\mathbf{s}$, and its land segment is the thick black line segment l. The target position is the point $\mathbf{t}$. The line that passes through the endpoints of l is the dashed line that is denoted by L. The point $\mathbf{v}$ is the point on l that lies closest to the intersection point $\mathbf{i}$ of the line segment from $\mathbf{j}$ to $\mathbf{t}$ with L, where $\mathbf{j}$ is the reflection of $\mathbf{s}$ in L. The red line segments form the local path p, defined by the sequence $\{\mathbf{s}, \mathbf{v}, \mathbf{t}\}$. Notice that the length of the line segment from $\mathbf{s}$ to $\mathbf{v}$ is equal to the length of the line segment from $\mathbf{j}$ to $\mathbf{v}$. Therefore, the length of p is equal to the length of the local path defined by the sequence $\{\mathbf{j}, \mathbf{v}, \mathbf{t}\}$. (b) The projection of $\mathbf{j}$ onto L is the point $\mathbf{j}'$, and the projection of $\mathbf{t}$ onto L is the point $\mathbf{t}'$. Notice that the interior angles of T_j and T_t at $\mathbf{i}$ are equal, where T_j is the triangle defined by the points $\{\mathbf{j}, \mathbf{i}, \mathbf{j}'\}$, and T_t is the triangle defined by the points $\{\mathbf{t}, \mathbf{i}, \mathbf{t}'\}$. These angles are denoted by α. (c) The point $\mathbf{w}$ is a point on the half line that starts at $\mathbf{i}$ and passes through $\mathbf{j}'$. We denote the distance between $\mathbf{w}$ and $\mathbf{i}$ by d. The red line segments form the local path defined by the sequence $\{\mathbf{s}, \mathbf{w}, \mathbf{t}\}$, which has the same length as the local path defined by the sequence $\{\mathbf{j}, \mathbf{w}, \mathbf{t}\}$.

Now, let $\mathbf{w}$ be a point of L. We assume, without loss of generality, that $\mathbf{w}$ lies on the half line that starts at $\mathbf{i}$ and passes through $\mathbf{j}'$, and we denote the distance between $\mathbf{w}$ and $\mathbf{i}$ by d (Figure 29c). Let q be the local path defined by the sequence $\{\mathbf{s}, \mathbf{w}, \mathbf{t}\}$. Then the length of q is equal to the length of the local path defined by the sequence $\{\mathbf{j}, \mathbf{w}, \mathbf{t}\}$. Using the triangles

T_j and T_t and the theorem of Pythagoras, we can express the length of q as

$$\begin{aligned}||\mathbf{j}-\mathbf{w}||+||\mathbf{t}-\mathbf{w}|| &= \sqrt{||\mathbf{j}-\mathbf{j}'||^2+(||\mathbf{i}-\mathbf{j}'||-d)^2}+\sqrt{||\mathbf{t}-\mathbf{t}'||^2+(||\mathbf{i}-\mathbf{t}'||+d)^2}\\ &= \sqrt{l_j^2\sin(\alpha)^2+(l_j\cos(\alpha)-d)^2}+\sqrt{l_t^2\sin(\alpha)^2+(l_t\cos(\alpha)+d)^2}\\ &= \sqrt{d^2-2l_j\cos(\alpha)\cdot d+l_j^2}+\sqrt{d^2+2l_t\cos(\alpha)\cdot d+l_t^2}.\end{aligned}$$

Now, let $f:\mathbb{R}^+\rightarrow\mathbb{R}^+$ be the function that maps the distance between $\mathbf{w}$ and $\mathbf{i}$ to the length of q. That is,

$$f(d)=\sqrt{d^2-2l_j\cos(\alpha)\cdot d+l_j^2}+\sqrt{d^2+2l_t\cos(\alpha)\cdot d+l_t^2}.$$

We will show that f is increasing for all $d>0$. In other words, we will show that the further $\mathbf{w}$ is removed from $\mathbf{i}$, the longer the length of q will be. From this, it will follow that p is the shortest path from $\mathbf{s}$ to $\mathbf{t}$, since its second vertex $\mathbf{v}$ is the point on l that lies closest to $\mathbf{i}$. To show that f is increasing for all $d>0$, we show that the derivative f' of f is larger than zero for all $d>0$. We will proof this statement by contradiction. We thus suppose that there exists value $d>0$, for which $f'(d)\leq 0$. Using the chain rule for differentiation, we can write this expression as

$$\frac{d-l_j\cos(\alpha)}{\sqrt{d^2-2l_j\cos(\alpha)\cdot d+l_j^2}}+\frac{d+l_t\cos(\alpha)}{\sqrt{d^2+2l_t\cos(\alpha)\cdot d+l_t^2}}\leq 0.$$

The denominators of both the fractions in this expression are strictly positive by definition, as is the numerator of the right fraction. If $d\geq l_j\cos(\alpha)$, then the numerator of the left fraction is positive, which means that the sum of both fractions must be strictly positive (a contradiction). Let us therefore assume that $d<l_j\cos(\alpha)$. Moving the left fraction to the right side of the equation and squaring both sides yields

$$\frac{(d+l_t\cos(\alpha))^2}{d^2+2l_t\cos(\alpha)\cdot d+l_t^2}\leq\frac{(d-l_j\cos(\alpha))^2}{d^2-2l_j\cos(\alpha)\cdot d+l_j^2},$$

which can be simplified to

$$2l_tl_j\cos(\alpha)\leq l_td-l_jd.$$

Since we deducted that $d<l_j\cos(\alpha)$, it follows that

$$2l_tl_j\cos(\alpha)>l_tl_j\cos(\alpha)>l_td>l_td-l_jd,$$

which is a contradiction. ∎

Now, let us consider the second main case, in which C_{i+1} contains multiple polygons. In this case, $\mathbf{t}$ lies inside a different polygon than j_i's land segment, which means that we have to 'cross' one or more polygon edges to reach it. Let e be the first of those polygon edges (i.e. e is shared edge between the first and second polygon in C_{i+1}), and let L be the line that passes through the endpoints of j_i's land segment. Then we can decompose Case 2 into the following subcases:

(a) The endpoints of e either lie on L, or lie on the other side of L than j_i's start position.

(b) Both endpoints of e lie on the same side of L as j_i's start position, or one of them lies on L and the other one lies on the same side of L as j_i's start position.

(c) The endpoints of e lie on opposite sides of L.

To find a local path from j_i's start position to $\mathbf{t}$, we use similar strategies as we used for Case 1. In Case 2a, we first create a funnel F, whose apex is j_i's start position, and whose left and right portal points are the left and right endpoints of j_i's land segment, respectively. We then apply the steps of the SSFA, setting F as the initial funnel, setting j_i's start position as the start position, and setting $\mathbf{t}$ as the goal position. If the second vertex that would be added to the local path is not an endpoint of j_i's land segment, then we include an additional step that first adds the intersection of the line segment from F's apex to this vertex with j_i's land segment to V. Case 2b is handled in the same way, except that the initial funnel F is now *mirrored* in j_i's land segment. That is, the apex of F is set to the reflection of j_i's start position in L, and F's left and right portal points are set to the right and left endpoints of j_i's land segment, respectively. In Example 6.2 below, we explain the steps of the approach for Case 2b, using the example of Figure 30.

Now, let us consider the third subcase, in which the endpoints of e lie on opposite sides of L. In this case, the path vertex v that will come after the vertex on j_i's land segment can lie on either side of L with respect to j_i's start position. If it turns out that j_i's start position and v will lie on opposite sides of L, then in order to find the vertex on j_i's land segment, we may need to compute the intersection of the line segment from j_i's start position to v with j_i's land segment. If, on the other hand, j_i's start position and v will lie on the same side of L, then we may need to compute the intersection of the line segment from the reflection of j_i's start position to v with j_i's land segment. Since the position of v is not known to us yet, we solve this case by creating two funnels. The first funnel F_1 is defined as in Case 2a, and the second funnel F_2 is defined as in Case 2b. We then apply the steps of the SSFA, while maintaining both funnels at the same time. Once one of these funnels *collapses* (i.e. if either the third or forth event, described in Section 3.2.2, gets triggered for one of the funnels), then instead of restarting the algorithm, we destroy this funnel, and continue the algorithm with the other funnel. In Example 6.3 below, we explain the steps of this approach, using the example of Figure 31.

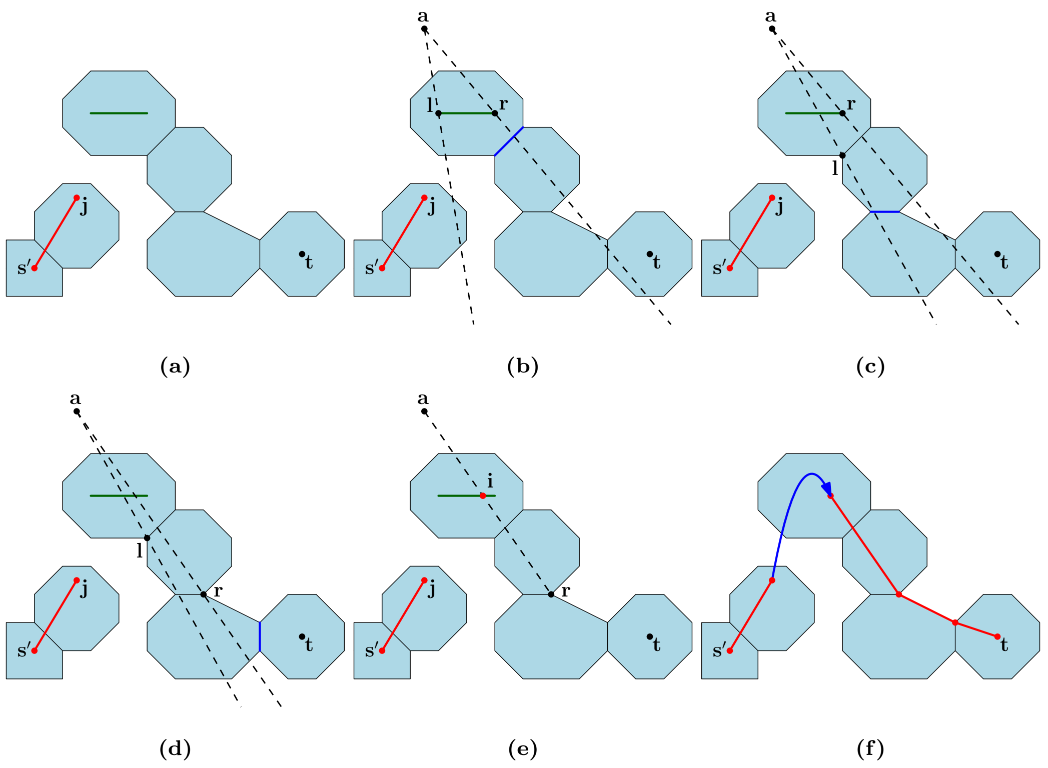


Figure 30: Visualization of Example 6.2.

Example 6.2. Let us consider Figure 30a. In this image, we see a sequence of two corridors, in which the first polygon of the first corridor contains the start position $\mathbf{s'}$, and in which the last polygon of the second corridor contains the target position $\mathbf{t}$. There is one jump link, whose start position $\mathbf{j}$ lies inside the last polygon of the first corridor, and whose land segment (the green line segment) lies inside the first polygon of the second corridor. The red line segment is the local path from $\mathbf{s}'$ to $\mathbf{j}$ that is computed in earlier steps of the algorithm. Our goal is to find a local path from $\mathbf{j}$ to $\mathbf{t}$, whose second vertex is a point on the land segment. The endpoints of the first polygon edge that we have to cross (the dark-blue edge in Figure 30b) both lie on the same side of the land segment as $\mathbf{j}$. Therefore, we create a funnel F, whose apex $\mathbf{a}$ is the reflection of the start position of the jump link in the line that passes through the endpoints of the land segment. The left portal point $\mathbf{l}$ and the right portal point $\mathbf{r}$ of F are set to the right and left endpoints of the land segment, respectively (Figure 30b). We then apply the steps of the SSFA, as they are described in Section 3.2.2. In the first iteration of the SSFA, the algorithm checks the orientations of the endpoints of the first polygon edge with respect to F. The left endpoint of this edge lies inside F, and the right endpoint lies to the right of F. Therefore, $\mathbf{l}$ is moved to the left endpoint of the edge (Figure 30c). In the second iteration, the orientations of the endpoints of the second polygon edge (the dark-blue edge in Figure 30c) are considered. The left endpoint of this edge lies to the left of F, and the right endpoint lies inside F. Therefore, $\mathbf{r}$ is moved to the right endpoint of the edge (Figure 30d). The endpoints of the third edge (the dark-blue edge in Figure 30d) both lie to the right of F. Therefore, $\mathbf{r}$ is the point that would be added as a vertex to the local path in the third iteration of the SSFA. However, we modify the SSFA to include an intermediate step that adds the intersection $\mathbf{i}$ of the line segment from $\mathbf{a}$ to $\mathbf{r}$ with the land segment as a vertex to the local path, before $\mathbf{r}$ is added to the local path (Figure 30e). Then, the algorithm is restarted at $\mathbf{r}$. Figure 30f shows the final local path after the remaining steps of the algorithm are complete. We have drawn the line segment in this path that connects the second and third vertices as an arc, since this line segment represents the trajectory of the jump from the first corridor to the second one.

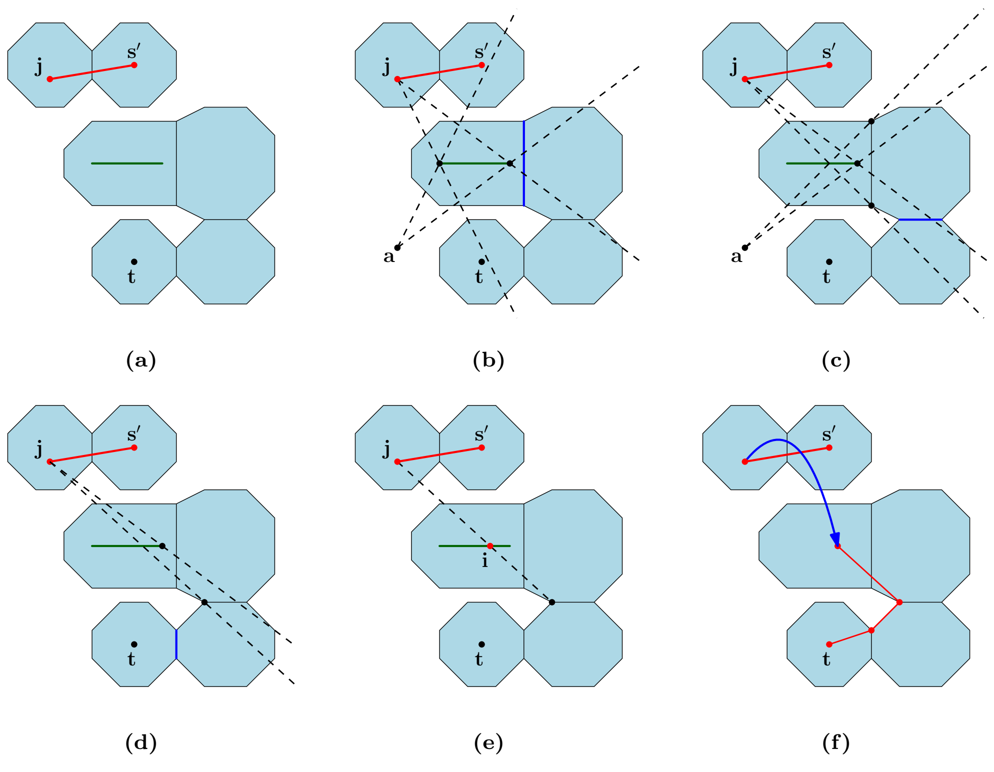


Figure 31: Visualization of Example 6.3.

Example 6.3. Let us consider Figure 31a, which has a similar setup as Figure 30a. Our goal is to find a local path from $\mathbf{j}$ to $\mathbf{t}$, whose second vertex is a point on the land segment. The endpoints of the first polygon edge that we have to cross (the dark-blue edge in Figure 31b) lie on opposite sides of the land segment. Therefore, we create two funnels F_1 and F_2.

The apex of F_1 is set to $\mathbf{j}$, and its left and right portal points are set to the left and right endpoints of the land segment, respectively. The apex of F_2 is set to the point $\mathbf{a}$, which is the reflection of the start position of the jump link in the line that passes through the endpoints of the land segment. The left and right portal points of F_2 are set to the right and left endpoints of the land segment, respectively (Figure 31b). We then apply the steps of the SSFA, while maintaining both funnels at the same time. In the first iteration, we consider the orientations of the endpoints of the first polygon edge with respect to F_1 and F_2. The left endpoint of this edge lies inside F_1, and the right endpoint lies inside F_2. Therefore, we move the left portal point of F_1 to the left endpoint of this edge, and we move the right portal point of F_2 to the right endpoint of this edge (Figure 31c). In the second iteration, we consider the orientations of the second polygon edge (the dark-blue edge in Figure 31c) with respect to F_1 and F_2. Since both endpoints lie to the left of F_2, we destroy F_2, and continue the algorithm with F_1. Since the left endpoint lies inside F_1, we move the left portal point of F_1 to the left endpoint of the edge (Figure 31d). The endpoints of the third edge (the dark-blue edge in Figure 31d) both lie to the left of F_1. Therefore, we compute the intersection $\mathbf{i}$ of the line segment from the apex of F_1 to the left portal point with the land segment, and we add it as a vertex to the local path. Then, we add the left portal point as a vertex to the local path, and we restart the algorithm at the left portal point (Figure 31e). Figure 31f shows the final local path after the remaining steps of the algorithm are complete.

6.2.2 Selecting Jump Links

In this section, we show how we determine which jump links in J will be incorporated into the local path. In the previous section, we have shown how these jump links are used by our local planning algorithm to find a local path from the start position $\mathbf{s}'$ to the goal position $\mathbf{g}'$. For convenience, we will treat the jump links in J as two-dimensional objects in the remainder of this section. When we speak of start positions or land segments of jump links, we will thus refer to their projections onto $\mathbb{R}^2$.

There may be multiple jump links in J that connect a pair of consecutive corridors in C. For each pair of consecutive corridors, we want to select the jump link that will end up giving us a short local path from $\mathbf{s}'$ to $\mathbf{g}'$. In theory, we could calculate exactly which choice of jump links will yield the shortest path that can be obtained by our local planning algorithm. We could run the algorithm on every possible combination of jump links $\{j_0, j_1, \ldots, j_{n-1}\}$, in which each j_i connects C_i to C_{i+1}, and select the combination that yields the shortest path from $\mathbf{s}'$ to $\mathbf{g}'$. However, if C contains many corridors, and if there are many different jump links that connect the pairs of consecutive corridors, then this approach is very computationally expensive. Therefore, instead of considering different combinations of jump links in the selection process, we will choose the jump links on an 'individual basis'. We will apply the steps of the local planning algorithm, as they are described in Section 6.2.1, and select the next jump link whenever it is first required by the algorithm. The choice of the next jump will only depend the previous choices, and not future ones.

Let us first explain how we select the first jump link j_0, connecting the first corridor C_0 to the second corridor C_1. Recall, from Section 6.2.1, that our local planning algorithm will first compute a local path from $\mathbf{s}'$ to j_0's start position, and then compute a local path from j_0's start position to the next target position $\mathbf{t}$, which is either the goal position, or the start position of the next jump link. Let $J_0 \subset J$ be the set of jump links that connect C_0 to C_1, and let j be a jump link in J_0. Then the utility of j can be expressed as the length of the resulting local path from $\mathbf{s}'$ to $\mathbf{t}$, when j is used as a jump link by the local planning algorithm. If we know the value of $\mathbf{t}$, we can calculate the exact length of this path, by plugging j into the local planning algorithm, and letting it compute the path from $\mathbf{s}'$ to $\mathbf{t}$. However, since applying this algorithm on every jump link in J_0 is expensive, we will use a heuristic method for evaluating the jump links in J_0. For each jump link $j \in J_0$, we will estimate the length of the local path from $\mathbf{s}'$ to j's start position, and we will estimate the length of the local path from j's start position to $\mathbf{t}$. We will then add our estimates together to obtain an estimate of the total length of the path from $\mathbf{s}'$ to $\mathbf{t}$, when j is used as a jump link. The jump link $j_0 \in J$ that has the lowest estimated path length, will then be selected to be incorporated into the local path.

As we mentioned above, selecting jump links is a process that is integrated into our local planning algorithm. We will apply the steps of this algorithm, and select the first jump link whenever it is first required. From Section 6.2.1, recall that the local planning algorithm first applies the SSFA on the first corridor C_0. The SSFA maintains a funnel F, while iterating over the shared edges between the consecutive polygons in C_0. When there are no more edges to be processed, the algorithm first requires access to the first jump link in the path, and it is at this moment that we will select it. Let $j \in J_0$ be a jump link connecting C_0 to C_1. We will first explain how we estimate the length of the local path from $\mathbf{s}'$ to j's start position. To do this, we will use the current state of the funnel F. Notice that the local path from $\mathbf{s}'$ to the start position of any jump link in J_0 will always pass through the apex $\mathbf{a}$ of F. Therefore, it will be sufficient to only estimate the length of the path from $\mathbf{a}$ to j's start position. Our estimate of the length of this path is the length of the shortest polygonal chain p that connects $\mathbf{a}$ to j's start position, and that passes through the line segment s, connecting the left portal point $\mathbf{l}$ and the right portal point $\mathbf{r}$ of F. If s is reduced to a single point (which means that $\mathbf{l} = \mathbf{a}$ and $\mathbf{r} = \mathbf{a}$), then we define p to be the line segment connecting $\mathbf{a}$ to j's start position. Notice that p is not always a valid path (i.e. it does not always lie completely within C_0). However, if p is a valid path, then it is always the shortest valid path from $\mathbf{a}$ to j's start position. Figure 32 shows some examples.

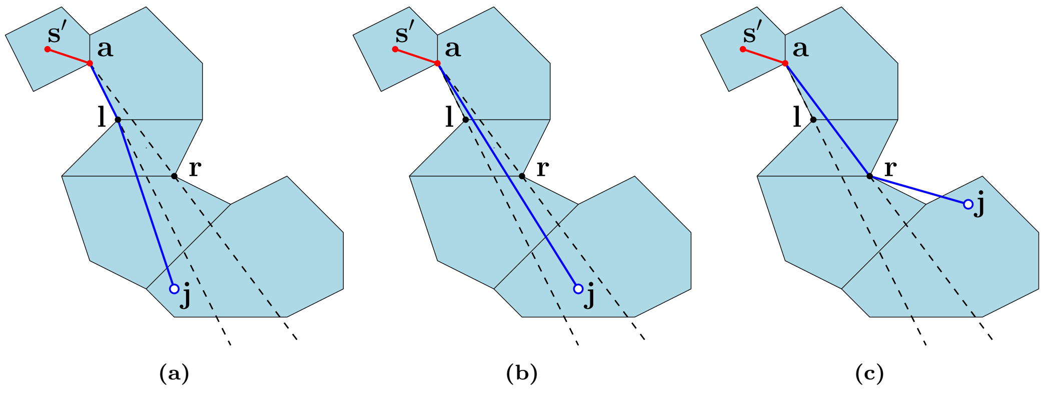


Figure 32: Examples of estimating the length of the local path from the apex $\mathbf{a}$ of a funnel F to the start position $\mathbf{j}$ of a jump link. Our estimate of the length of this path is the length of the shortest polygonal chain p that connects $\mathbf{a}$ to j's start position, and that passes through the line segment connecting the left portal point $\mathbf{l}$ and the right portal point $\mathbf{r}$ of F. (a) $\mathbf{j}$ lies to the left of F. Therefore, p is the polygonal chain that is defined by the points $\{\mathbf{a}, \mathbf{l}, \mathbf{j}\}$. (b) $\mathbf{j}$ lies inside F. Therefore, p is the line segment connecting $\mathbf{a}$ to $\mathbf{j}$. (c) $\mathbf{j}$ lies to the right of F. Therefore, p is the polygonal chain that is defined by the points $\{\mathbf{a}, \mathbf{r}, \mathbf{j}\}$. Notice that p is not a valid path; it does not lie completely within the corridor.

Let us now explain how we estimate the length of the path from j's start position to the target position $\mathbf{t}$. Recall that, if C_1 is the last corridor in C, then $\mathbf{t} = \mathbf{g}'$. If C_1 is not the last corridor in C, then $\mathbf{t}$ is the start position of the next jump link in the path (the one connecting C_1 to C_2). Since we have not selected the next jump link yet, we will (temporarily) set $\mathbf{t}$ to centroid of the last polygon in C_1 in this case, because this position is expected to lie relatively close to the start position of the next jump link. To estimate the length of the path from j's start position to $\mathbf{t}$, we construct a *backwards funnel* B, by running the SSFA *backwards* on C_1. That is, we apply the SSFA on the polygons of C_1 in the reversed order, setting $\mathbf{t}$ as the start position. When there are no more polygon edges to be processed, we set B to the current state of the funnel. Notice that the path from the start position of any jump link in J_0 to $\mathbf{t}$ always passes through the apex $\mathbf{a}$ of B. Therefore, it will be sufficient to only estimate the length of the path from j's start position to $\mathbf{a}$. Our estimate of the length of this path is the length of the shortest polygonal chain that connects

j's start position to $\mathbf{t}$, that passes through j's land segment, and that passes through the line segment connecting the portal points of B (Figure 33). Notice that we can use B to evaluate all the jump links in J_0. Therefore, we only have to compute it once when selecting the first jump link.

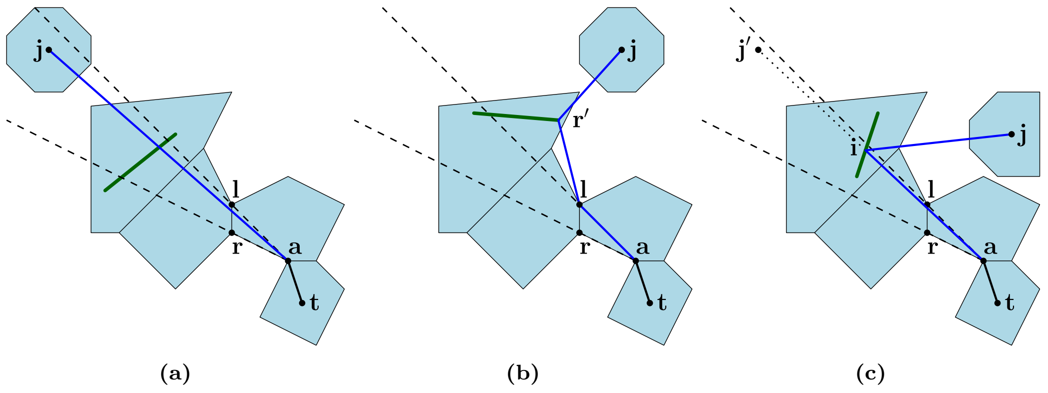


Figure 33: Examples of estimating the length of the local path from the start position $\mathbf{j}$ of a jump link to the apex $\mathbf{a}$ of backwards funnel B. Our estimate of the length of this path is the length of the shortest polygonal chain p that connects $\mathbf{j}$ to $\mathbf{t}$, that passes through the land segment of the jump link (the green line segment), and that passes through the line segment connecting the left portal point $\mathbf{l}$ and the right portal point $\mathbf{r}$ of B. (a) In this example, p is the line segment connecting $\mathbf{j}$ to $\mathbf{t}$. (b) In this example, p is the chain defined by the points $\{\mathbf{j}, \mathbf{r}', \mathbf{l}, \mathbf{a}\}$, where $\mathbf{r}'$ is the right endpoint of the land segment. (c) In this example, p is the chain defined by the points $\{\mathbf{j}, \mathbf{i}, \mathbf{a}\}$, where $\mathbf{i}$ is the intersection of the land segment with the line segment connecting $\mathbf{a}$ to $\mathbf{j}'$, where $\mathbf{j}'$ is the reflection of $\mathbf{j}$ in the line that passes through the endpoints of the land segment.

Once we have evaluated all the jump links in J_0, we select the link $j_0 \in J$ with the lowest estimated path length, and plug it into the local planning algorithm. To select the second jump link j_1, connecting C_1 to C_2, we use a similar strategy as described above. We continue to run the local planning algorithm, until it first requires access to j_1. Recall that the algorithm first creates at least one new funnel F_1. The apex of this funnel is either set to the start position of j_0, or to the reflection of j's start position in the line that passes through the endpoints of j's land segment, and its portal points are set to the endpoints of j's land segment. In one special case (Case 2c in Section 6.2.1), the algorithm creates an additional funnel F_2, which is the mirrored version of F_1. The algorithm then applies the steps of the SSFA on C_1, while maintaining F_1 (and potentially F_2). When there are no more polygon edges to be processed, the algorithm first requires access to the second jump link in the path, and it is at this moment that we will select it.

To select the second jump link, we use a similar approach as described above. For each jump link j connecting C_1 to C_2, we first estimate the length of the path from j_0's start position to j's start position. If there is only one funnel, then we estimate the length of this path in the same way as before. That is, our estimate is the length of the shortest polygonal chain that connects the apex of the current funnel to j's start position, and that passes through the line segment connecting the portal points of the funnel. If there are two funnels however, then our estimation is slightly different. In this case, the two funnels always share precisely one portal point that lies on the endpoint of j_0's land segment. Let $\mathbf{p}_1$ and $\mathbf{p}_2$ be the two *other* portal points. That is, $\mathbf{p}_1$ is the portal point of F_1 that is not a portal point of F_2, and $\mathbf{p}_2$ is the portal point of F_2 that is not a portal point of F_1. Our estimate of the length of the path from j_0's start position to j's start position is then defined as the length of the shortest polygonal chain that connects j_0's start position to j's start position, that passes through j's land segment, and that passes through the line segment connecting $\mathbf{p}_1$ to $\mathbf{p}_2$ (Figure 34). Estimating the length of the path from j's start position to the next target position is done in the exact same way as before. That is, we first compute a backwards funnel by running the SSFA backwards on C_2, and compute the shortest polygonal chain that connects j's start position to the apex of the backwards funnel, that passes through j's land segment, and that passes through the line segment connecting the portal points of the backwards funnel.

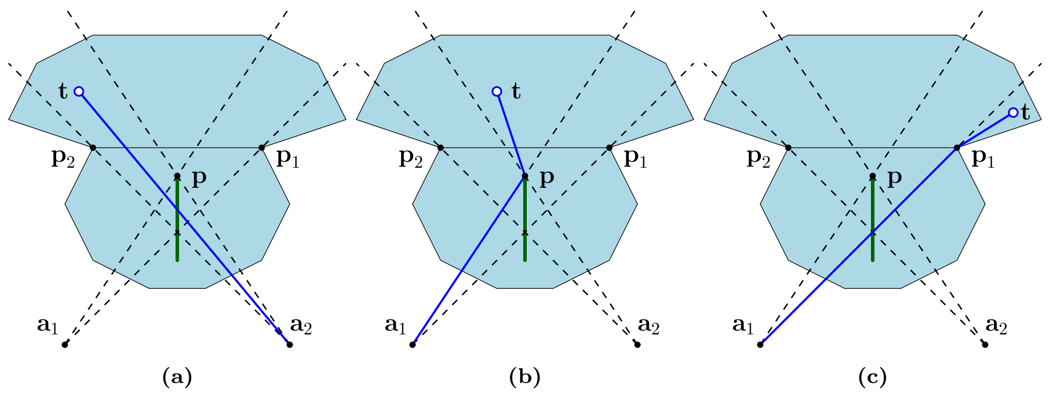


Figure 34: Examples of estimating the length of the local path from the start position $\mathbf{j}$ of a jump link to the target position $\mathbf{t}$. The land segment of the jump link is the green line segment. In these examples, there are two funnels F_1 and F_2. The apex $\mathbf{a}_1$ of F_1 is $\mathbf{j}$, and the apex $\mathbf{a}_2$ of F_2 is the reflection of $\mathbf{j}$ in the line that passes through the endpoints of the land segment (or vice versa). Notice that the two funnels share the portal point $\mathbf{p}$, which is an endpoint of the land segment. The other two portal points are denoted by $\mathbf{p}_1$ and $\mathbf{p}_2$. Our estimate of the length of the path from $\mathbf{j}$ to $\mathbf{t}$ is the length of the shortest polygonal chain that connects $\mathbf{j}$ to $\mathbf{t}$, that passes through the land segment, and that passes through the line segment connecting $\mathbf{p}_1$ and $\mathbf{p}_2$. (a) $\mathbf{j}$ lies inside F_2. Therefore, the length of p is equal to the length of the line segment from $\mathbf{a}_2$ to $\mathbf{t}$. (b) $\mathbf{t}$ lies 'between' F_1 and F_2. That is, $\mathbf{t}$ lies to the left of F_1, and lies to the right of F_2. Therefore, the length of p is equal to the length of the polygonal chain defined by the points $\{\mathbf{a}_1, \mathbf{p}, \mathbf{t}\}$. Notice that the length of this chain is equal to the length of the chain defined by the points $\{\mathbf{a}_2, \mathbf{p}, \mathbf{t}\}$. (c) $\mathbf{t}$ lies to the right of F_1. Therefore, the length of p is equal to the length of the polygonal chain defined by the points $\{\mathbf{a}_1, \mathbf{p}_1, \mathbf{t}\}$.

Once we have evaluated all the jump links connecting C_1 to C_2, we select the link j_1 with the lowest estimated path length, and plug it into the local planning algorithm. We then continue following the same approach to select the remaining jump links in the path.

6.3 Path Following

In this section, we discuss how an agent can use a local path to navigate itself towards its goal position. Our implementation does not include any algorithms that make an agent follow a path, so the concepts presented in this section are purely theoretical. The main purpose of this section is to present ideas on how the jump links and the paths that are generated with our methods could be used in a practical application, such as a computer game. In addition, we will expose one of the main problems of our method, for which we have not found a proper solution yet.

When an agent receives a request to move to a goal position, a path leading to the goal position must be computed, and the agent must follow this path without colliding with other agents and dynamic obstacles. The path-planning algorithms that we presented in Sections 6.1 and 6.2 can be used to compute a local path from the agent's current position to the goal position. Given this local path, we can compute the *desired velocity* of the agent, by subtracting the agent's current xy-position from the position of the next vertex in the local path. The desired velocity can be used as input for a steering model, such as the RVO method [31]. Such a steering model can compute a two-dimensional steering vector, which can be used as the horizontal velocity of the agent in the current update. The steering vector will make sure that the agent will move in roughly the same direction as its desired velocity, while avoiding collisions with other agents, and staying within the boundaries of the navigation mesh. Physics or raycasts can be used to make sure that agent remains grounded on the surface of the environment.

When the agent reaches the start position of a jump link j (i.e. when its xy-position aligns with the position of the vertex in the local path that represents j's start position), it is time for the agent to perform a jump. In practice, this means that the agent must move along a trajectory, while displaying a jump animation. A problem that we have yet been able to solve is how this trajectory should be defined. The land position of the jump is represented by the next vertex of the local path, which can be mapped to a position $\mathbf{l} \in \mathbb{R}^3$ on j's land segment. Using the jump parameters that were used for j's construction, we could define a trajectory in the direction of $\mathbf{l}$, starting at the agent's current position. However, it will not be guaranteed that this trajectory will have a land position on the surface of the environment. An alternative approach would be to first find a land position on the surface of the environment, and then to 'fit' a trajectory between the agent's current position and this land position. Since j's land segment lies inside a polygon of the navigation mesh, it is guaranteed that there is a walkable surface that lies above or below this land segment (assuming that the environment contains no small holes). Therefore, we can use a raycast to find a land position on the surface of the environment that either lies directly above or below the point $\mathbf{l}$. We can then fit a jump trajectory between the agent's current position and the land position that we found with the raycast. For example, we could fix the value of the jump height h that was used during the construction of j, and use Equation 1 (Definition 4.3) to compute a new jump distance d', yielding a trajectory that connects the agent's current position to the land position. One disadvantage of this approach is that the resulting trajectory is not guaranteed to be collision free, so there is a chance that the agent will penetrate an object while traversing the trajectory. The maximum penetration depth depends on the voxel size and the value of ϵ that was used during the construction of the land segments. Therefore, the maximum depth can be reduced by decreasing these values. However, decreasing the voxel size leads to longer computation times for generating the navigation mesh and the jump links. Decreasing the value of ϵ leads to a larger amount of land segments, which results in larger storage requirements, and longer path-planning times.

As part of our future work, we intend to extend our implementation by adding agents that actually follow the paths that are computed with our methods. To calculate the animation trajectories for the jumps in the path, we plan to use the second approach described above. We can then test how often the agents collide with objects when jumping, and how the collisions effect the perceived realism of the simulation. We expect that these collisions will be rare, and that the visual effects will be tolerable. However, if this turns out not to be the case, then we will reconsider the steps of our methods, and try to come up with better solutions.

7 Implementation

In this section, we discuss our implementation of the methods presented in Sections 5 and 6. Since these methods have been designed specifically for navigation meshes that are created with Recast, we have decided to integrate our implementation into the Recast Navigation software. Recast Navigation is an open-source software package that is written in C++ [20]. It includes the Recast Demo program, which can be used to build and test navigation meshes. This program allows the user to load a Wavefront OBJ file, defining the geometry of a virtual environment. The user can then choose between different interfaces, called *samples*, which can be used to generate different types of navigation meshes for the environment. The *Solo Mesh Sample*, for example, generates a navigation mesh for the entire environment at once. The *Tile Mesh Sample*, on the other hand, decomposes the environment into tiles, and generates a navigation mesh for each tile separately. Each sample has its own set of tools, which are called *sample tools*. These tools can be used to test and modify the navigation mesh that is created with the sample. We have implemented our methods as a single sample tool for the Solo Mesh Sample. This sample tool, which we call the *Jump-Link Tool*, contains two sub-tools, which are called the *Generation Tool* and the *Path-Planning Tool*. The Generation Tool is an interface that enables users to extend the navigation mesh with jump links, and the Path-Planning Tool is an interface that enables users to find paths on the extended navigation mesh.

7.1 The Generation Tool

The Generation Tool is an interface that enables users to extend a navigation mesh with jump links. It contains several sliders, which can be used to set the parameters of the algorithms for generating the jump links. There are sliders for selecting the jump parameters (i.e. the jump height and the jump distance), for selecting the maximum fall distance, and for selecting the value of ϵ that is used by the Zhao-Saalfeld algorithm (Section 5.5). Once these parameters are set, the user can generate jump links in two different ways: manually, and automatically. Generating jump links manually consists of selecting a sample point with the mouse. The user can click on a position in the environment, which calls a function that finds the walkable voxel that corresponds to the click position (if there is such a voxel). A sample point is then created at the bottom-left corner of the top face of this voxel, and the algorithms described in Sections 5.4 and 5.5 are called on this sample point. To generate jump links automatically, the user must choose a step size $d \in \mathbb{N}$, and press a button. This calls a function that applies the generation algorithms on the bottom-left corners of the top faces of all the walkable voxels, whose horizontal grid coordinates are divisible by d (Section 5.3). The Generation Tool allows the user to create different types of jump links with different parameters. That is, after the user has generated a set of jump links for one configuration of the parameters, he is allowed to change the parameters, and expand his set by generating additional jump links, using the new parameters. This is useful for applications in which agents can perform different types of jumps.

The jump links that are generated with the Generation Tool are visualized as follows. The start positions of the jump links are indicated by horizontal circles, which are centered at the corresponding sample points. The land segments of the jump links are drawn as line segments. For each land segment, two parabolic arcs are drawn between the start position of the jump link and the endpoints of the land segment, representing the trajectories of the corresponding jumps (Figure 35). To prevent clutter, the user has the option to turn off the rendering of some of these elements. For example, he can choose to turn off the rendering of the jump arcs, so that only the start positions and the land segments will be drawn.

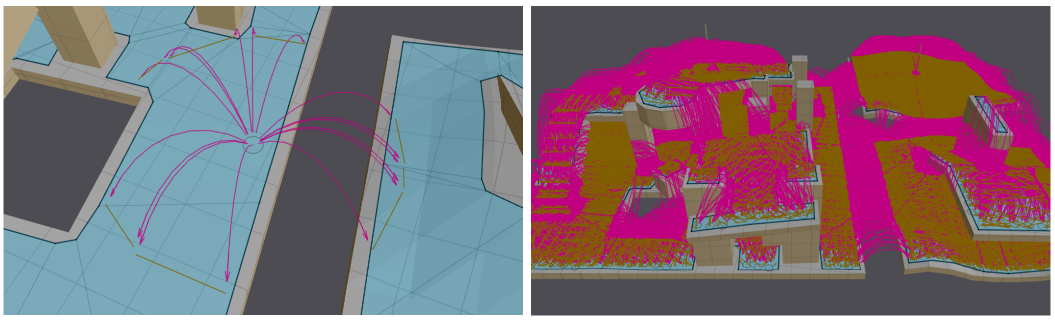

(a) (b)

Figure 35: (a) A set of jump links corresponding to a single sample point. The start position of these links (the sample point) is indicated by the gray circle. The land segments are drawn as yellow line segments. For each land segment, two pink parabolic arcs are drawn between the start position and the endpoints of the line segment, representing the trajectories of the corresponding jumps. (b) The total set of jump links, corresponding the sample points from Figure 18.

When the user is satisfied with his set of jump links, he can 'bake' the jump links into the navigation mesh by pressing a button. The jump links are then stored within the class of the Jump-Link Tool, so that they can be accessed by the Path-Planning Tool. Each jump link is stored as a single object, containing the start position and the land segment. The configuration of the jump height that was used when the link was generated is also stored within this object (this is necessary for path visualization, which we will discuss in the next section). For each polygon of the navigation mesh, we create a separate list of jump links, containing all the links whose start positions lie inside the polygon. These lists are subdivided into different sublists, such that the land segments of all the jump links that are contained within one sublist lie within the same polygon of the navigation mesh. This enables our global planning algorithm to access the list of polygons that are connected to a given polygon by jump links in constant time. It also enables our local planning algorithm to access the list of jump links, whose start positions lie inside a polygon P_i, and whose land segments lie inside another polygon P_j, in $O(n)$ time, where n is the number of polygons that are connected to P_i by jump links.

Our code for generating the jump links is a direct implementation of the methods described in Section 5. However, our implementation does not include the construction of the dilated heightfield (Section 5.2). Instead of using the dilated heightfield in the algorithm from Section 5.4.2, we use the heightfield that is generated by Recast (Section 3.1). Even though the navigation mesh is build for cylindrical agents, the jump links generated with our prototype are thus only suited for point-based agents. We leave the implementation of the construction of the dilated heightfield for future work.

7.2 The Path-Planning Tool

The Path-Planning Tool is an interface that enables users to find paths on a navigation mesh that is extended with jump links. It contains one slider, which can be used to select the jump-cost parameter used by our global path-planning algorithm (Section 6.1). The user can select a start position and a goal position by clicking on the navigation mesh. Once these positions are selected, a path is computed between the two positions. The code that computes this path is a direct implementation of the algorithms described in Section 6.1 and 6.2, so we will not discuss it further.

When a path between the two selected positions is found, we visualize it as follows. We visualize the global path by giving the polygons in the sequence of corridors defined by the global path a different color than usual. We visualize the local path as a series of dots, line segments, and parabolic arcs. The dots represent the vertices of the local path, the line segments represent the walkable parts of the path, and the parabolic arcs represent the jumps in the path (Figure 36).

Even though the vertices of the local path are defined as two-dimensional points, we visualize them as three-dimensional points. Recall, from Section 6.2, that there are three types of vertices in the local path (not considering the vertices representing the start and the goal position); there are path vertices that represent vertices of polygons of the navigation mesh (these vertices represent the 'corners' in the path); there are vertices that represent the start positions of the jumps in the path; and there are vertices that represent the land positions of the jumps in the path. For each corner in the path (i.e. a path vertex that represents a vertex of a polygon), we draw a dot at the 3D-position of the polygon vertex from which the corner was derived. For each path vertex that represents the start position of a jump, we draw a dot at the sample point from which the corresponding jump link was derived. Finally, for each path vertex that represents the land position of a jump, we draw a dot on the land segment from which the vertex was derived. We find the height coordinate of this dot by using linear interpolation on the heights of the endpoints of the land segment.

We connect the dots that represent the vertices of the local path with line segments and parabolic arcs. Specifically, we connect a pair of dots representing two consecutive path vertices with a parabolic arc, if the path vertices were derived from the same jump link. Otherwise, we connect the dots with a line segment. Let $\mathbf{p}_i \in \mathbb{R}^3$ and $\mathbf{p}_{i+1} \in \mathbb{R}^3$ be two dots that represent two consecutive path vertices that were derived from a jump link j. Then $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$ respectively represent the start and the land position of a jump defined by the parameters h and d, where h is the jump height, and d is the jump distance used during the construction of j. However, due to some of the approximations made during the

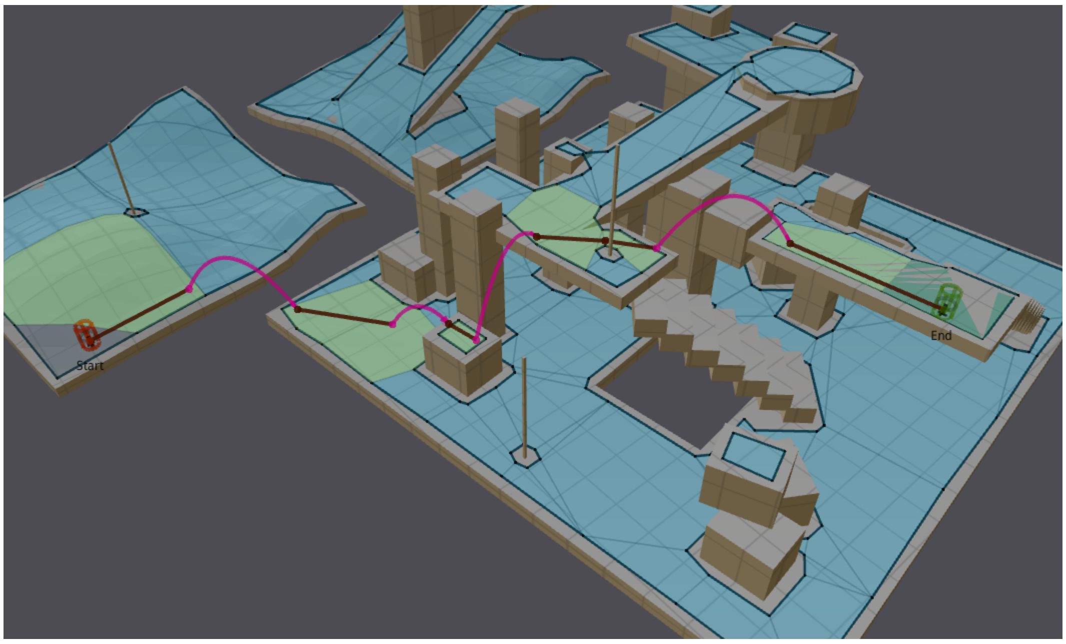


Figure 36: Visualization of a path. The start and the goal position are indicated by the red and green cylinders, respectively. The polygon that contains the start position is drawn in dark red, and the polygon that contains the goal position is drawn in dark-green. The other polygons in the corridors are drawn in light green. The local path is indicated by a series of dots, line segments, and parabolic arcs. The red line segments represent the walkable parts of the path, and the pink arcs represent the jumps in the path.

construction of j, $\mathbf{p}_{i+1}$ does not always lie on the trajectory of a jump that starts at $\mathbf{p}_i$, and that is defined by h and d. Therefore, to define a parabolic arc that connects $\mathbf{p}_i$ to $\mathbf{p}_{i+1}$, we must sometimes use different parameters. To find these parameters, we leave the jump height h as it was during the construction of j, and we use Equation 1 (Definition 4.3) to compute a jump distance d', such that the resulting trajectory passes through $\mathbf{p}_{i+1}$. Then, using h and the new jump distance d', we can define and draw a parabolic arc connecting $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$. However, since the value of d' can slightly deviate from d, this arc is not guaranteed to be collision free. During testing, we have seen some cases where the arcs in the visualized paths partially intersect with the geometry of the environment. However, we have found these cases to be extremely rare.

8 Experiments and Results

To test the implementation of our methods, we conducted experiments on a variety of different environments. The purpose of these experiments was to investigate how our method behaves under different parameter settings. Specifically, we wanted to know which parameter settings are needed to obtain sets of jump links that have relatively low construction times and memory usages, while being large enough to support high-quality paths. Our methods involve many different parameters, all of which influence the construction time of the jump links and the quality of the paths that incorporate them. However, the configuration of most of these parameters will be dictated by the application in which the jump links are required. For example, the configuration of the parameters for generating the navigation mesh and the configuration of the jump parameters will all depend on the size and the locomotion abilities of the agents. Therefore, we focused our attention on the step size that is used for generating the sample points (Section 5.3), which is one of the few parameters that does not depend on the agent type. Using different step sizes, we measured the performance of our algorithm for generating the jump links; we measured the memory usage of the jump links, and we measured the usability of the jump links for generating paths. In Section 8.1, we will discuss the experiments for measuring the construction time and memory usage of the jump links. Then, in Section 8.2, we will discuss the experiments that involve path planning. Finally, in Section 8.3, we draw conclusions from the results of our experiments, and recommend a range of step sizes that is expected to yield satisfactory results in most applications.

Test Environments: We conducted experiments on six different environments: `Dungeon`, `Navtest`, `House`, `Ai Test Area`, `Multiplayer Map`, and `Single Player Map`. `Navtest` and `Dungeon` are test levels that are included within the Recast Navigation software package. The `House` environment is a level that we obtained from the Sketchup 3D Warehouse [25]. The `Ai Test Area` is a level that is used by Guerrilla Games for testing A.I. systems. `Multiplayer Map` and `Single Player Map` are levels from the Killzone games [15]. A detailed description of the test environments can be found in Appendix A.1.

Experimental Setup: All of our experiments were conducted on a laptop with an Intel Core i7-4702MQ processor (2.2 GHz), an nVidia GeForce 820M graphics card and 8GB of RAM. The operating system was Windows 10 Home (64x bits). Only a single core was used during the experiments.

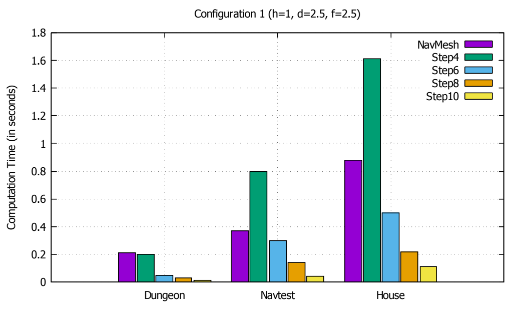
8.1 Generating Jump Links

We tested our algorithm for generating jump links on each of our six test environments. For each environment, we pre-computed a navigation mesh with Recast. We used a voxel size of $0.25 \times 0.25 \times 0.25$ meters. The agent height was is set to 1.8 meters, and the agent width was set to 0.5 meter (representing a humanoid computer-game character). The remaining parameters of Recast were left to their default values. For each environment, we created a set of jump links, using several different configurations of the jump parameters. We used the following three configurations of the jump height h, the jump distance d, and the maximum fall distance f (in meters):

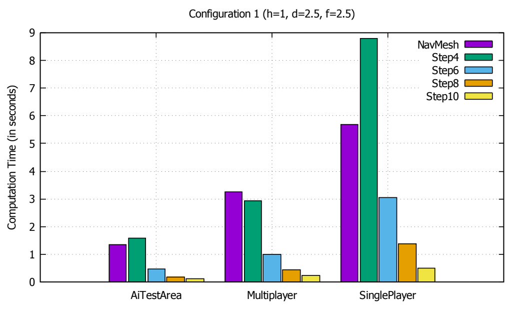
1. $h = 1$, $d = 2.5$, $f = 2.5$.
2. $h = 3$, $d = 7.5$, $f = 7.5$.
3. $h = 5$, $d = 12.5$, $f = 12.5$.

Our choices for the three configurations are relatively arbitrary. Their main purpose is to demonstrate how the performance of our algorithm scales to jumps of different sizes. For each of these configurations, we tested our approach using nine different step sizes: 1, 2, 4, 6, 8, 10, 12, 14 and 16. This translates to having a sample point every 0.25, 0.5, 1, 1.5, 2, 2.5, 3, 3.5 and 4 meters, respectively. During all of the experiments, we fixed the value of ϵ for the Zhao-Saalfeld algorithm to 0.25 meters, which is the size of one voxel.

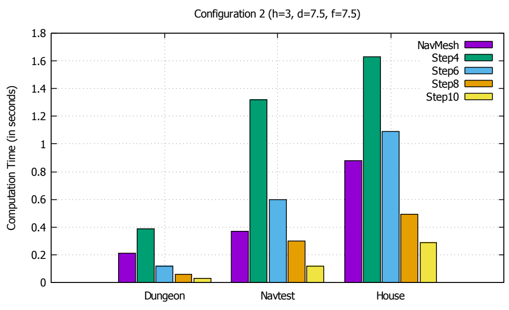
In total, we thus tested our approach on 27 different parameter configurations. For each test environment, and for each of the 27 configurations, we measured the total computation time of the entire pipeline, which includes the generation of the sample points, the generation the jump links, and the procedure that stores the jump links in the class of the Jump Link Tool (Section 7.1). The construction times of the navigation meshes were left out of the equation. We averaged each result over 10 runs, to compensate for any fluctuations due to the operating system. The test results of these experiments can be found in Appendix A.2. From the results, we can see that using a step size of 1 is very expensive in terms of performance. In the `Single Player Map`, which is our largest test environment, generating the jump links with this step size took roughly a minute for the smallest jumps, and roughly 2.5 minutes for the largest jumps. For some applications, these running times may be acceptable. However, for a level designer who quickly wants to test the navigability of his environment, 1 or 2 minutes is a long time to wait. Using a step size of 4 or larger yields more acceptable computation times in this scenario (under 15 seconds for each experiment). In Figure 37, we have plotted the computation times for step sizes 4 to 10. The construction times of



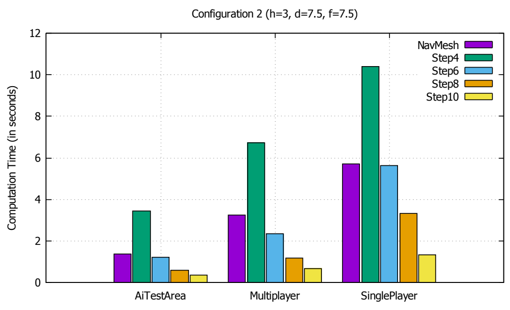
(a)



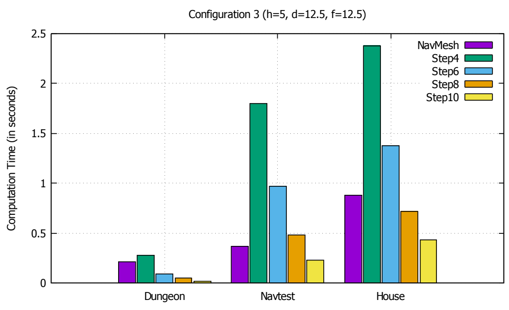
(b)



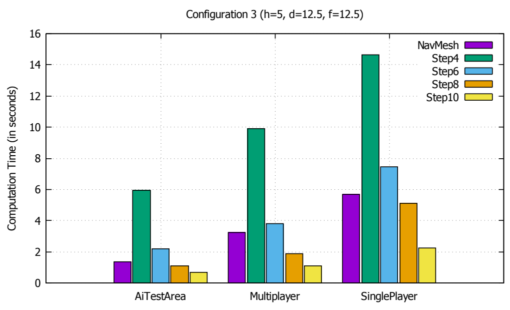
(c)



(d)



(e)

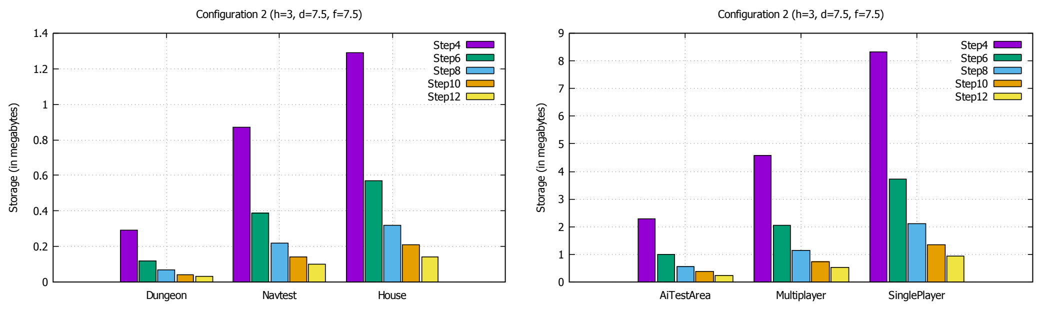


(f)

Figure 37: Test results of our experiments for measuring the performance of our algorithm for generating jump links. The computation times are plotted for step sizes 4, 6, 8, and 10 (the green, blue, yellow and orange bars, respectively). The purple bars show the construction times of the navigation meshes. Figures (a) and (b) show the computation times for the first configuration of the jump parameters ($h = 1$, $d = f = 2.5$). Figures (c) and (d) show the computation times for the second configuration ($h = 3$, $d = f = 7.5$). Figures (e) and (f) show the computation times for the third configuration ($h = 5$, $d = f = 12.5$).

the navigation meshes are also included in this figure, so a comparison can be made. We can see that generating jump links with a step size of 4 is generally more expensive than constructing the navigation mesh, and that generating jump links with a step size of 8 is generally cheaper than constructing the navigation mesh. If we use the construction time of the navigation mesh as reference point to an 'acceptable' amount of computation time, then we can conclude from Figure 37 that using a step size of 6 or more will generally lead to satisfactory results.

For each environment, and for each of the 27 configurations, we have also measured the memory usage of the resulting set of jump links. The test results of these experiments can be found in Appendix A.3. For our smallest environments (`Dungeon`, `Navtest` and `House`), the jump links cost less than 10 MB, starting at a step size of 2 or more, and cost less than 1 MB, starting at a step size of 6 or more. For our largest environments (`Ai Test Area`, `Single Player Map` and `Multiplayer Map`), the jump links cost less than 10 MB, starting at a step size of 6 or more, and cost less than 1 MB, starting at a step size of 14 or more. Figure 38 shows a plot of the memory costs of the jump links for step sizes 4 to 12, using the second configuration of the jump parameters ($h = 3$, $d = 7.5$, $f = 7.5$). Whether these costs should be considered 'acceptable' or not depends on the application.



(a) **(b)**

Figure 38: Test results of our experiments for measuring the memory usage of the jump links. The storage costs are plotted (in megabytes) for step sizes 4, 6, 8, 10 and 12 (the purple, green, blue yellow and orange bars, respectively). Figure (a) shows the storage costs for our smallest test environments, and Figure (b) shows the storage costs of our largest environments.

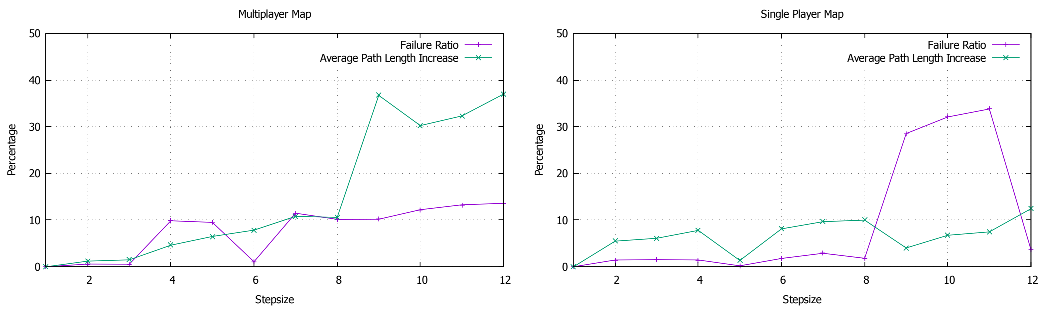
8.2 Path Planning

In the previous section, we saw that the configuration of the step size heavily influences the construction time and memory usage of the jump links. Increasing the step size leads to fewer jump links, which means lower construction times and less memory usage. However, having fewer jump links also means that it is less likely that a path between two positions in the environment can be found. Therefore, increasing the step size also lowers the usability of the resulting set of jump links.

In this section, we discuss an experiment that we conducted to measure the influence of the configuration of the step size on the usefulness of the resulting set of jump links. We performed this experiment on our two largest test environments, which are the `Multiplayer Map` and the `Single Player Map`. These environments are rich of obstacles and gaps, and therefore contain many paths that involve jumps. To generate the navigation meshes for these environments, we used the same parameter settings as we used in the experiments from Section 8.1. To generate jump links, we used the second configuration of the jump parameters that we used in the experiments from Section 8.1 ($h = 3$, $d = 7.5$ and $f = 7.5$). We chose this configuration of the jump parameters, because the resulting jumps are large enough to cover most of the gaps and obstacles in the environments.

In our experiment, we first generated a set of jump links, using a step size of 1. Since this step size translates to generating a sample point on *every* walkable voxel, the resulting set of jump links is the largest set that can be generated under the current settings of the jump parameters. In addition, the resulting set contains all the jump links that can possibly be generated under these parameters, regardless of which step size is used. Therefore, we can use this set of jump links as a reference for measuring the usefulness of sets of jump links that are generated with different step sizes. That is, given a set of jump links that is generated with an arbitrary step size, we can measure its usefulness by planning several paths with this set, and comparing these paths to the paths that would be computed, if we would have used the set of jump links that was generated with step size 1.

Using the set of jump links generated with step size 1, we let our path-planning algorithms solve 10.000 random path queries on each of our test environments. If a path could not be found, or if the resulting path did not contain any jumps, then we discarded the path query, and we generated a new one. For each query, we measured the length of the resulting path. Next, we deleted the set of jump links, and generated new sets using different step sizes. For each of those sets, we then solved the same 10.000 path queries that we had generated before, and counted the number of times that a path could not be found. For the path queries that were solved successfully, we measured the lengths the resulting paths, and compared them to the lengths of the original paths.



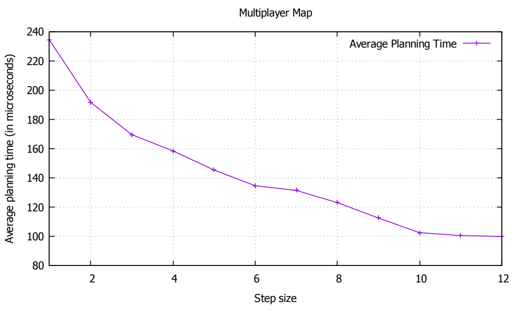
(a) **(b)**

Figure 39: Test results of our experiment for evaluating the usefulness of sets of jump links that are generated with different step sizes. Figure (a) shows the results for the `Multiplayer Map`, and Figure (b) shows the results for the `Single Player Map`. The purple curves show the ratio of path queries where a path could not be found. The green curves shows the average relative increase in path length of the queries that were solved successfully.

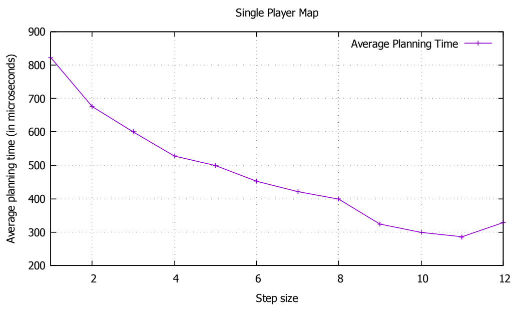
Figure 39 shows the results of our experiment. The purple curves show the ratio of failed path queries, and the green curves show the average relative increase in path length for the queries that were solved successfully. To give an example on how the figure should be read, consider the values that are plotted for step size 10 in Figure 39a. For this step size, the failure ratio is $\approx 12\%$, which means that in roughly 12% of the queries, no path could be found. The average relative increase in path length is $\approx 30\%$, which means that the path length of a successful path query was, on average, 30% longer than the original path that was computed with the first set of jump links (the set that was generated with a step size of 1).

The results of our experiment are heavily fluctuating, which is not what we expected (at least not at this scale). We expected the failure rate and the average path length to increase steadily with the step size, but we can see in Figure 39 that the curves include several spikes. In the `Single Player Map`, for example, the number of failed path queries drops from 3376 for step size 11, to 369 for step size 12. We can explain the sudden drops and increases in the curves by looking at the layout of our test environments. The environments contain several 'global routes' that are followed by most of the paths. If one of these routes becomes unavailable, due to the lack of jump links, then this can cause a large number of path queries to fail. Apparently, the set of jump links that was generated with a step size of 12 in the `Single Player Map` contains a few 'important' jump links that are incorporated in a large number of paths. Similar conclusions can be drawn about the other spikes in the curves of Figure 39. This shows that the quantity of the sample points is not the only factor that determines the usefulness of a set of jump links. What is equally, or perhaps more important, is the location of the sample points. As for a recommendation for a step size, we can conclude that using a step size of 8 or lower leads to good results in both environments. For this range of step sizes, the failure ratio and the average increase in path length is generally less than 10% in both environments. However, since our results are so heavily fluctuating, more experiments will be needed before we draw this conclusion with confidence.

To give the reader an idea of the performance of our path-planning method, we have measured the average computation times of the path queries used in our experiments. The results are shown in Figure 40. To compute the graphs in this figure, we only used queries that were solved successfully.



(a)



(b)

Figure 40: Average path-planning times in our experiment for evaluating the usefulness of sets of jumps links that are generated with different step sizes. Figure (a) shows the average planning times for the `Multiplayer Map`. Figure (b) shows the average planning times for the `Single Player Map`.

8.3 Conclusion

From our experiments, we conclude that using a step size that ranges between 6 and 8 can be expected to yield satisfactory results in most applications. Using these step sizes, sets of jump links can be constructed in a matter of seconds, even for very large environments. These sets can be expected to take up between 1 and 10 MB in storage space for environments having similar sizes as our test environments. Our experiments from Section 8.2 have shown that these sets are also large enough to support many of the paths through an environment that require jumps. Our path-planning algorithms can compute such paths in a few hundred microseconds, depending on the size of the environment. We believe that these results look promising, especially since some of the steps of our method were implemented rather naively, and provide a lot of room for optimizations.

9 Conclusion

In this thesis, we have presented an automated approach for generating jump links for arbitrary virtual environments. Our method operates on a voxel representation of the environment and its walkable space, which is a modified version of the voxel representation that is used by Recast's algorithms for generating navigation meshes. Our approach is based on sampling positions from the top faces of walkable voxels, using a fixed step size. For each sample point, we compute a set of land voxels, which are walkable voxels that an agent can jump to from the sample point. Given this set of land voxels, we generate a set of line segments, and create a jump link for each line segment in this set. Each of our jump links thus consists of a sample point and a line segment, where the line segment represents a set of land positions of valid jumps that start at the sample point. Since the voxels from which the jump links are derived can be coupled to a navigation mesh that is created with Recast, we can integrate these links into the navigation-mesh data structure. We have presented a modified version of Detour's path-planning method, which can be used to find paths on a navigation mesh that is extended with our jump links. This method effectively utilizes the point-to-segment structure of the links, to find relatively short paths that include jumps.

We have integrated our methods into the Recast Navigation software, and conducted experiments to test this implementation. Our results show that under certain configurations of the step size, our approach is capable of producing commodious sets of jump links for large and complex environments in a respectable amount of time. We think that the paths that can be generated with these links look promising, but more work will be needed to determine how useful these paths are in practice. Due to the approximations made during the construction of the jump links, the start and land positions of the jumps in the path do not always align with the geometry of the environment. Therefore, it may be difficult to find an animation trajectory for an agent that is performing a jump in its path. In Section 6.3, we have argued how an animation trajectory can be computed, given the start and land positions of a jump. However, the solution that we proposed is not guaranteed to produce a collision-free trajectory. Even though we expect that collisions during these jumps will be rare, and that the visual effects will be tolerable, we cannot know this for certain until more experiments have been conducted.

9.1 Future work

Even though our current implementation seems to produce promising results, there is still a lot more to be done before our techniques can be effectively used in practice. As for now, our implementation only includes algorithms for generating jump links and for planning paths that utilize these links. However, we have yet to implement any agents that actually follow the paths that are computed with our methods. In Section 6.3, we presented ideas on how a path-following algorithm could operate on these paths. However, the approach that we discussed has the disadvantage that agents may collide with objects while performing the jumps in the path. In the future, we intend to extend our implementation with this path-following method, and perform experiments to determine if the results are acceptable, or if more work is needed. If this path-following method is to be used effectively in practice, then it should include a collision avoidance strategy, and support nonholonomic agents.

If we succeed in implementing a working path-following algorithm, then we believe that our implementation can form a useful extension to the Recast Navigation software. We think that it would be particulary useful for computer games with relatively small environments, containing only static geometry. However, if our approach is to be used in games that involve huge open worlds with dynamic geometry, then more work would be needed. In this type of game, the environment is often subdivided into different tiles. A separate navigation mesh is then created for each tile, and the navigation meshes of neighboring tiles are stitched together to form a single mesh. In order to avoid long computation times, the navigation mesh is usually only generated in the tiles around the position of the player's avatar. When the player moves across the map, new sections of navigation mesh are generated at runtime, while other sections are removed. If the geometry of the environment changes in one of the tiles, then the navigation mesh can be recomputed locally in that specific tile, and can be reattached to the surrounding navigation mesh. The Recast Navigation software supports such a tile-based approach, but our implementation of the jump-link generation does not. Since each tile has its own heightfield, our current implementation is only capable of generating jump links that are contained within a single tile, and is not able to produce jumps links between neighboring tiles.

If our methods are to be used for the type of game described above, then our algorithm for generating jump links should be able to operate at interactive rates. Even though our experiments show that the running time of our current algorithm is relative low, we expect that the method is not fast enough to satisfy this real-time requirement. One way of optimizing our method could be to prune the areas in which we search for jump links. Currently, our method searches for jump links everywhere, which causes it to generate many 'redundant' links. A jump link can be considered redundant, if an agent that is standing at the start position of the link can walk in a "straight line" to any point on the land segment of the link. If the jump cost that is used by the global planning algorithm is set high enough, then such a jump link will never be incorporated into a path, and will just take up unnecessary space. We expect that we can speed up the jump-link generation significantly, by preventing the redundant jump links from being generated. The observation can be made that a jump link is only useful, if it represents jumps that pass over bounding edges of the navigation mesh. Therefore, it would be sufficient to only search for jump links near the boundaries of the navigation mesh. This would mean that we would only choose our sample points at positions from which an agent can jump over a boundary edge, and that we would only search for land voxels in the direction of that boundary edge. Notice that we can easily adapt our algorithm for finding land voxels to search within a specific range of directions, instead of searching in every direction. Choosing our sample points near the bounding edges of the navigation mesh is a less trivial problem, and we have yet to come up with an efficient solution.

We believe, provided that the problems mentioned in this section can be solved efficiently, that our software can be a useful tool for any application involving jumping agents, whose navigation model is based on Recast.

Acknowledgements

This research has been supported by the COMMIT/ project [2]. I would like to thank my internal supervisor Roland Geraerts of the University of Utrecht for his guidance while I was writing this thesis. I would also like to thank the A.I. team at Guerrilla Games for their hospitality and support during my time at the company. I would especially like to thank my external supervisor Arjen Beij for helping me to come up with ideas, and for teaching me to become a better coder. Finally, I would like to thank my parents and my girlfriend for providing mental support during this project.

References

[1] Choi, M. G., Lee, J., & Shin, S. Y. (2003). Planning biped locomotion using motion capture data and probabilistic roadmaps. *ACM Transactions on Graphics (TOG)*, 22(2), 182-203.

[2] COMMIT/ project. `http://www.commit-nl.nl/`, 2016.

[3] CryEngine Documentation on Off-Mesh Navigation, from the Multi-Layer Navigation section, `http://docs.cryengine.com/display/SDKDOC2/Off-mesh+Navigation`.

[4] Cui, X., & Shi, H. (2012). An overview of pathfinding in navigation mesh. IJCSNS, 12(12), 48.

[5] Douglas, D. H., & Peucker, T. K. (1973). Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. Cartographica: The International Journal for Geographic Information and Geovisualization, 10(2), 112-122.

[6] Geisler, B. (2004). Integrated machine learning for behavior modeling in video games. *Challenges in game artificial intelligence: papers from the 2004 AAAI workshop. AAAI Press, Menlo Park*, 54-62.

[7] Guerrilla Games: `http://www.guerrilla-games.com/`.

[8] Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on, 4*(2), 100-107.

[9] Haumont, D., Debeir, O., & Sillion, F. (2003, September). Volumetric cell-and-portal generation. Computer Graphics Forum (Vol. 22, No. 3, pp. 303-312). Blackwell Publishing, Inc.

[10] Hershberger, J., & Snoeyink, J. (1994). Computing minimum length paths of a given homotopy class. Computational geometry, 4(2), 63-97.

[11] Horizon: Zero Dawn. Wikipedia page: `https://en.wikipedia.org/wiki/Horizon_Zero_Dawn`.

[12] Kallmann, M., & Kapadia, M. (2014, July). Navigation meshes and real-time dynamic planning for virtual worlds. *In ACM SIGGRAPH 2014 Courses* (p. 3). ACM.

[13] Kang, S. J., Kim, Y., & Kim, C. H. (2010). Live path: adaptive agent navigation in the interactive virtual world. *The Visual Computer*, 26(6-8), 467-476.

[14] Kapadia, M., & Badler, N. I. (2013). Navigation and steering for autonomous virtual humans. *Wiley Interdisciplinary Reviews: Cognitive Science*, 4(3), 263-272.

[15] Killzone (series). Wikipedia page: `https://en.wikipedia.org/wiki/Killzone_%28series%29`

[16] Latombe, J. C. (2012). Robot motion planning (Vol. 124). Springer Science & Business Media.

[17] Levine, S., Lee, Y., Koltun, V., & Popovic, Z. (2011). Space-time planning with parameterized locomotion controllers. *ACM Transactions on Graphics (TOG)*, 30(3), 23.

[18] Lopez, T., Lamarche, F., & Li, T. Y. (2012). Space-time planning in changing environments: using dynamic objects for accessibility. *Computer Animation and Virtual Worlds*, 23(2), 87-99.

[19] Mononen, M. (2010). Simple Stupid Funnel Algorithm. `http://digestingduck.blogspot.nl/2010/03/simple-stupid-funnel-algorithm.html`.

[20] Mononen, M. (2011). Recast Navigation. Website: `https://github.com/memononen/recastnavigation`.

[21] Mononen, M. (2011). Automatic Annotations in Killzone 3 and Beyond. *Paris Game/AI Conference*. Link to slides and demo: `http://digestingduck.blogspot.nl/2011/07/paris-gameai-conference-2011-slides-and.html`.

[22] Pratt, S. (2010 - 2015). Study: Navigation Mesh Generation, `http://www.critterai.org/projects/nmgen_study/`.

[23] Pratt, S. (2010 - 2015). Study: Navigation Mesh Generation; Chapter: Detail Mesh Generation, `http://www.critterai.org/projects/nmgen_study/detailgen.html`.

[24] Shi, W., & Cheung, C. (2006). Performance evaluation of line simplification algorithms for vector generalization. The Cartographic Journal, 43(1), 27-44.

[25] Sketchup 3D Warehouse, `https://3dwarehouse.sketchup.com/`.

[26] Snook, G. (2000). Simplified 3D movement and pathfinding using navigation meshes. *Game Programming Gems*, 1, 288-304.

[27] Soille, P. (2013). Morphological image analysis: principles and applications. Springer Science & Business Media.

[28] Sunshine-Hill, B. (2014). Automatic Traversal Analysis. *In Paris Game/AI Conference*. `http://www.gdcvault.com/play/1020571/Environmentally-Conscious-AI-Improving-Spatial`.

[29] Unity documentation on creating an off-mesh link, from the Navigation and Pathfinding section, `http://docs.unity3d.com/Manual/nav-CreateOffMeshLink.html`.

[30] Unity documentation on building off-mesh links automatically, from the Navigation and Pathfinding section, `http://docs.unity3d.com/Manual/nav-BuildingOffMeshLinksAutomatically.html`.

[31] Van Den Berg, J., Guy, S. J., Lin, M., & Manocha, D. (2011). Reciprocal n-body collision avoidance. In Robotics research (pp. 3-19). Springer Berlin Heidelberg.

[32] Wikipedia: A* search algorithm, `http://en.wikipedia.org/wiki/A*_search_algorithm`.

[33] Zhao, Z., & Saalfeld, A. (1997). Linear-time sleeve-fitting polyline simplification algorithms. In Proceedings of AutoCarto (Vol. 13, pp. 214-223).

A Experiments and Results

This appendix provides a description of the environments on which we have conducted our experiments (Appendix A.1), and it contains the test results of our experiments for measuring the construction time and memory usage of the jump links (Appendix A.2 and A.3, respectively).

A.1 Environments

Figures 41 to 46 show the geometry of our eight test environments and their corresponding navigation meshes. The navigation meshes were generated with Recast, using the parameters mentioned in Section 8.1. In the captions of the figures, we provide some information about the environments. Specifically, we state the number of triangles that the environment consist of, the horizontal measurements of the axis-aligned bounding box of the geometry of the environment, and the computation time of the construction of the navigation mesh.

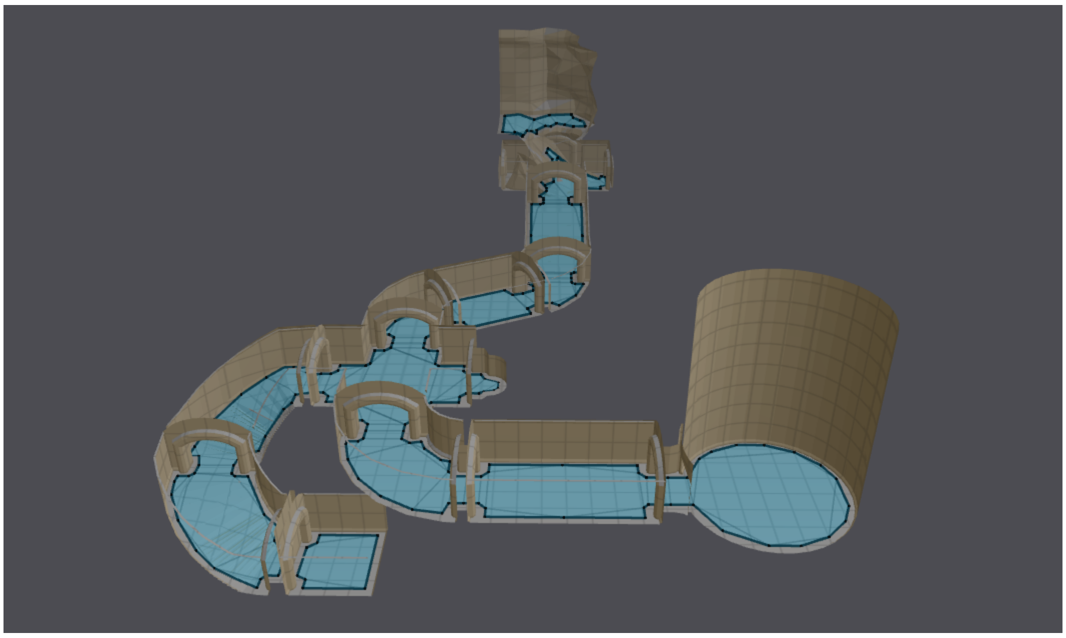


Figure 41: Screenshot of the `Dungeon` environment. This environment is a test level that is included in the Recast Navigation software package. It represents a dungeon containing several rooms. The environment is composed of 10133 triangles, and spans a horizontal area of approximately 76 × 99 meters. Constructing the navigation mesh took 209 milliseconds.

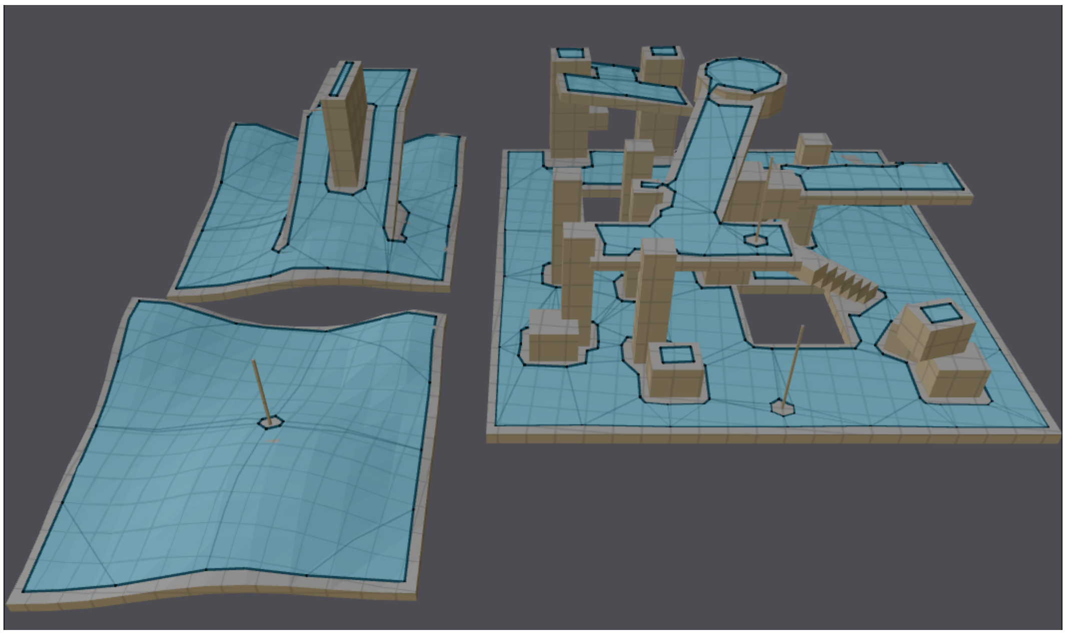


Figure 42: Screenshot of the `Navtest` environment. This environment is a test level that is included in the Recast Navigation software package. It consists of three platforms, containing some objects, some stairs and a few slopes. The environment is composed of 1612 triangles, and spans a horizontal area of approximately 92 × 77 meters. Constructing the navigation mesh took 368 milliseconds.

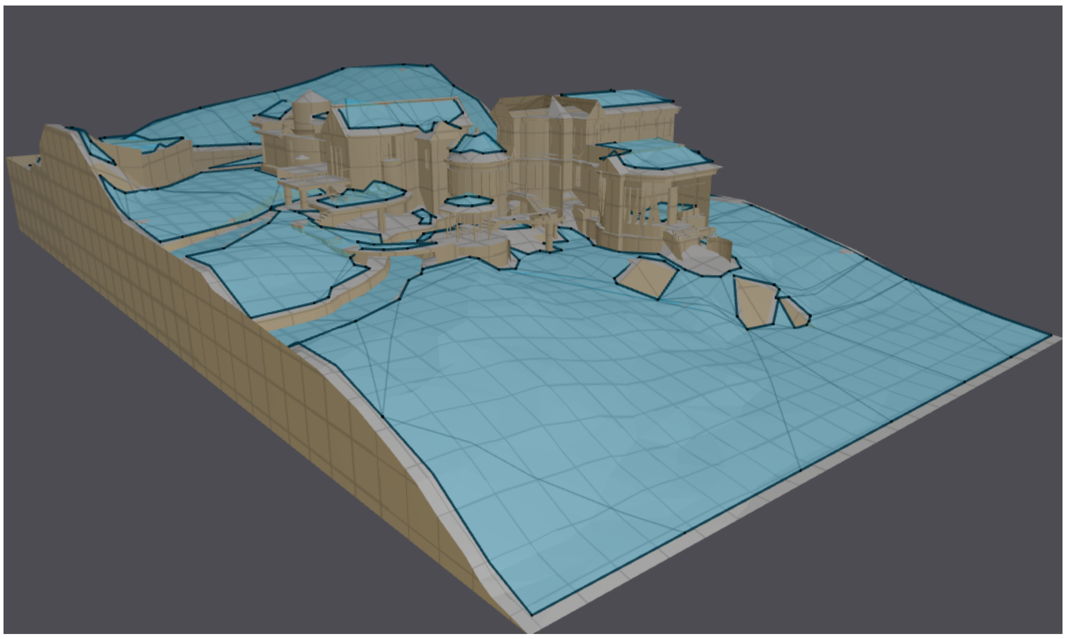


Figure 43: Screenshot of the `House` environment. It contains a house with a few terraces. The ground around the house contains several gaps, representing a river. The environment is composed of 119352 triangles, and spans a horizontal area of approximately 91 × 53 meters. Constructing the navigation mesh took 876 milliseconds.

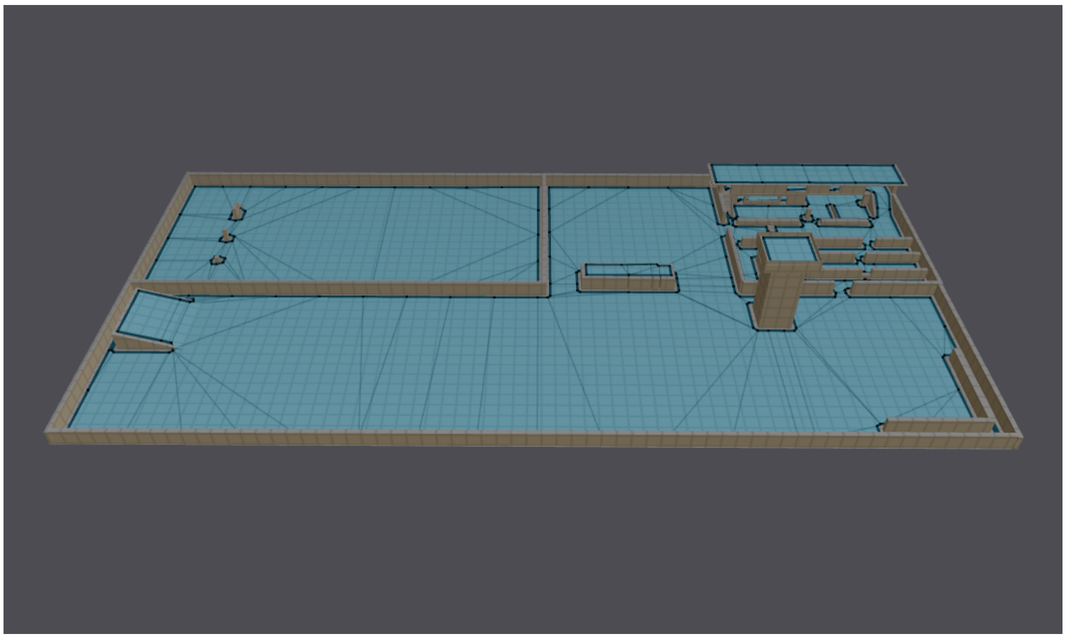


Figure 44: Screenshot of the `Ai Test Area` environment. This environment is used by Guerrilla Games for testing A.I. systems. It contains some walls, a tower, a small building, and a slope. The environment is composed of 1541 triangles, and spans a horizontal area of approximately 165 $\times$ 80 meters. Constructing the navigation mesh took 1356 milliseconds.

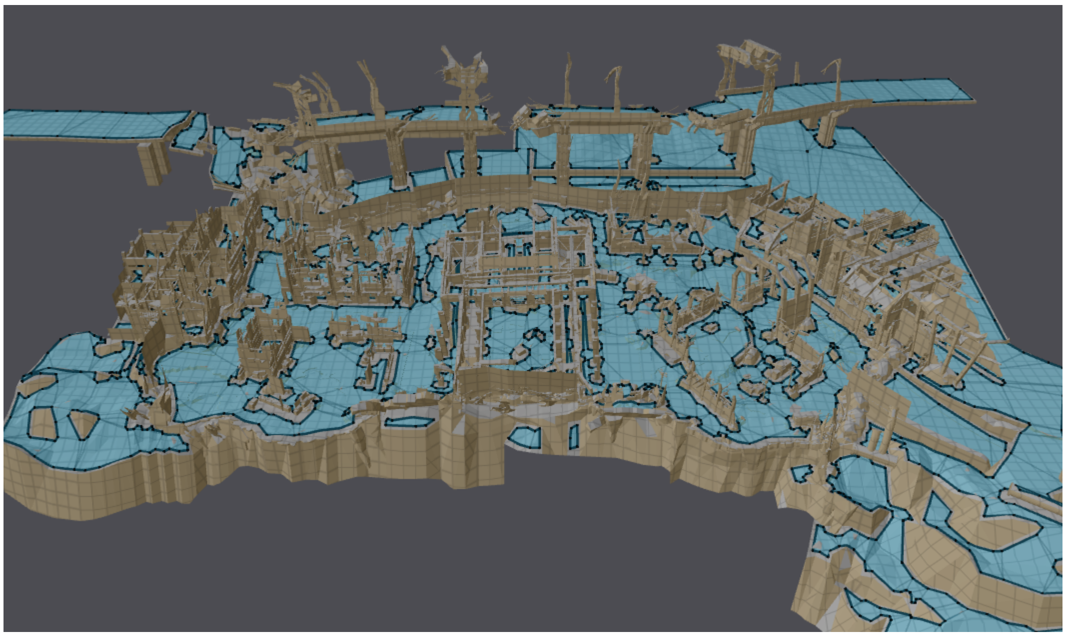


Figure 45: Screenshot of the `Multiplayer Map`. It represents the ruins of an urban area. The environment contains many buildings, some of which have been completely destroyed, while others have remained fairly intact. In the back, there are some remains of a destroyed elevated high way. The environment is composed of 259009 triangles, and spans a horizontal area of approximately 271 $\times$ 178 meters. Constructing the navigation mesh took 3245 milliseconds.

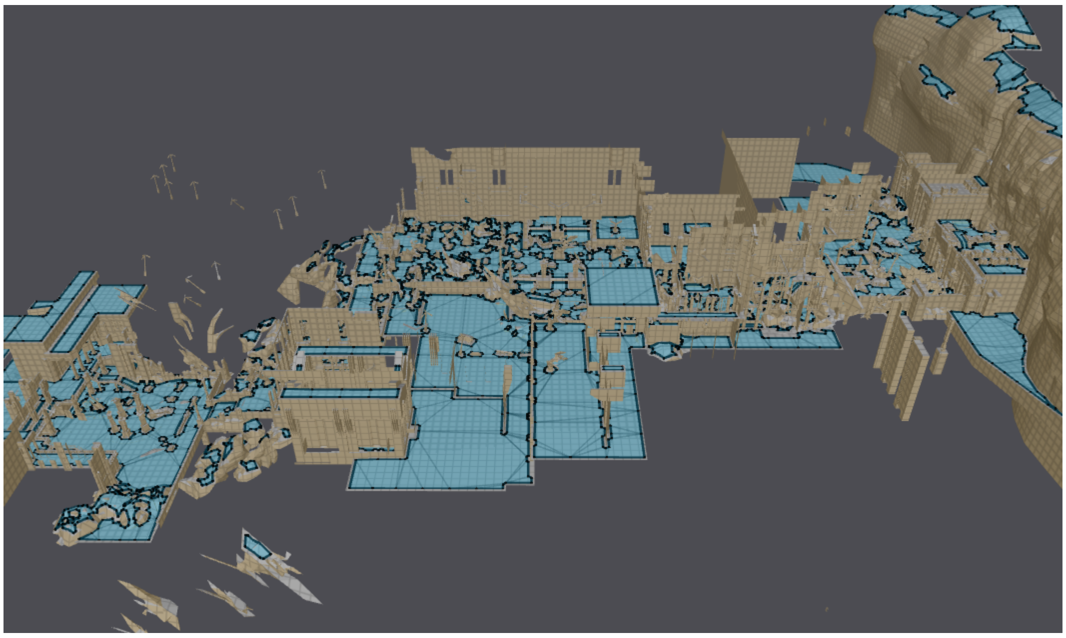


Figure 46: Screenshot of the `Single Player Map`. This environment is our largest test environment. It is composed of 216853 triangles, and spans a horizontal area of approximately 389×462 meters. Constructing the navigation mesh took 5689 milliseconds.

A.2 Test Results: Performance

Tables 2 to 6 contain the test results of the experiments for measuring the performance of our algorithm for generating jump links (Section 8.1). A separate table is provided for each of the test environments. The columns of these tables represent the different step sizes, and the rows represent the three different configurations of the jump height h, the jump distance d, and the maximum fall distance f. These configurations are defined as follows:

1. $h = 1$, $d = 2.5$, $f = 2.5$.
2. $h = 3$, $d = 7.5$, $f = 7.5$.
3. $h = 5$, $d = 12.5$, $f = 12.5$.

Table 1: Test results for the `Dungeon` environment. The columns of represent the different step sizes, and the rows represent the three different configurations of the jump parameters. The computation times are shown in seconds. The reason why the third configuration has lower average computation times than the second configuration is that the jumps defined by the third configuration are too large to fit inside the dungeon. Therefore, the algorithm for finding land voxels will detect collision with solid voxels sooner, causing it to terminate early.

	1	2	4	6	8	10	12	14	16
Config 3	3.66	1.07	0.28	0.09	0.05	0.02	0.02	0.01	0.01
Config 2	4.48	1.26	0.39	0.12	0.06	0.03	0.02	0.01	0.01
Config 1	2.37	0.96	0.20	0.05	0.03	0.01	0.01	0.00	0.00

Table 2: Test results for the `Navtest` environment. The columns of represent the different step sizes, and the rows represent the three different configurations of the jump parameters. The computation times are shown in seconds.

	1	2	4	6	8	10	12	14	16
Config 3	25.69	5.65	1.80	0.97	0.48	0.23	0.15	0.11	0.08
Config 2	14.28	3.61	1.32	0.60	0.30	0.12	0.08	0.06	0.04
Config 1	7.47	3.01	0.80	0.30	0.14	0.04	0.02	0.01	0.01

Table 3: Test results for the `House` environment. The columns of represent the different step sizes, and the rows represent the three different configurations of the jump parameters. The computation times are shown in seconds.

	1	2	4	6	8	10	12	14	16
Config 3	38.29	8.50	2.38	1.38	0.72	0.43	0.28	0.19	0.14
Config 2	21.34	5.69	1.63	1.09	0.49	0.29	0.19	0.12	0.09
Config 1	10.97	3.49	1.61	0.50	0.22	0.11	0.08	0.04	0.03

Table 4: Test results for the `Ai Test Area`. The columns of represent the different step sizes, and the rows represent the three different configurations of the jump parameters. The computation times are shown in seconds.

	1	2	4	6	8	10	12	14	16
Config 3	69.63	19.37	5.96	2.18	1.12	0.69	0.41	0.30	0.21
Config 2	39.99	14.13	3.45	1.21	0.58	0.36	0.20	0.14	0.10
Config 1	29.23	10.79	1.58	0.47	0.19	0.11	0.06	0.04	0.03

Table 5: Test results for the `Multiplayer Map`. The columns of represent the different step sizes, and the rows represent the three different configurations of the jump parameters. The computation times are shown in seconds.

	1	2	4	6	8	10	12	14	16
Config 3	111.78	28.62	9.92	3.78	1.88	1.12	0.72	0.50	0.37
Config 2	59.31	20.44	6.74	2.34	1.16	0.68	0.42	0.27	0.21
Config 1	45.73	17.15	2.93	0.99	0.44	0.24	0.16	0.10	0.07

Table 6: Test results for the `Single Player Map`. The columns of represent the different step sizes, and the rows represent the three different configurations of the jump parameters. The computation times are shown in seconds.

	1	2	4	6	8	10	12	14	16
Config 3	207.85	49.34	14.65	7.44	5.11	2.22	1.51	1.07	0.74
Config 2	119.25	31.37	10.38	5.62	3.32	1.32	0.88	0.60	0.42
Config 1	69.60	23.53	8.80	3.04	1.37	0.51	0.34	0.23	0.16

A.3 Test Results: Memory

Tables 7 to 12 contain the test results of the experiments for measuring the memory usage of the jump links (Section 8.1). A separate table is provided for each of the test environments. The columns of these tables represent the different step sizes, and the rows represent the three different configurations of the jump height h, the jump distance d, and the maximum fall distance f. These configurations are defined as follows:

1. $h = 1$, $d = 2.5$, $f = 2.5$.
2. $h = 3$, $d = 7.5$, $f = 7.5$.
3. $h = 5$, $d = 12.5$, $f = 12.5$.

Table 7: Test results for the `Dungeon` environment. The columns of represent the different step sizes, and the rows represent the three different configurations of the jump parameters. The memory usage of the jump links is shown in megabytes. The reason why the third configuration yields lower memory usage than the second configuration is that the jumps defined by the third configuration are too large to fit inside the dungeon. Therefore, less jump links will be generated for the third configuration.

	1	2	4	6	8	10	12	14	16
Config 3	2.47	0.62	0.16	0.07	0.04	0.02	0.02	0.01	0.01
Config 2	4.39	1.11	0.29	0.12	0.07	0.04	0.03	0.02	0.02
Config 1	1.97	0.49	0.13	0.06	0.03	0.02	0.01	0.01	0.01

Table 8: Test results for the `Navtest` environment. The columns of represent the different step sizes, and the rows represent the three different configurations of the jump parameters. The memory usage of the jump links is shown in megabytes.

	1	2	4	6	8	10	12	14	16
Config 3	20.35	5.09	1.27	0.57	0.32	0.21	0.15	0.11	0.08
Config 2	13.87	3.48	0.87	0.39	0.22	0.14	0.10	0.07	0.05
Config 1	4.11	1.03	0.26	0.12	0.07	0.04	0.03	0.02	0.02

Table 9: Test results for the `House` environment. The columns of represent the different step sizes, and the rows represent the three different configurations of the jump parameters. The memory usage of the jump links is shown in megabytes.

	1	2	4	6	8	10	12	14	16
Config 3	28.39	7.09	1.79	0.80	0.44	0.28	0.20	0.14	0.10
Config 2	20.43	5.11	1.29	0.57	0.32	0.21	0.14	0.10	0.08
Config 1	5.31	1.33	0.34	0.15	0.08	0.05	0.04	0.03	0.02

Table 10: Test results for the `Ai Test Area`. The columns of represent the different step sizes, and the rows represent the three different configurations of the jump parameters. The memory usage of the jump links is shown in megabytes.

	1	2	4	6	8	10	12	14	16
Config 3	64.96	16.18	4.06	1.78	1.02	0.66	0.44	0.34	0.25
Config 2	36.47	9.09	2.28	1.00	0.57	0.37	0.25	0.19	0.14
Config 1	8.70	2.17	0.54	0.24	0.14	0.09	0.06	0.04	0.03

Table 11: Test results for the `Multiplayer Map`. The columns of represent the different step sizes, and the rows represent the three different configurations of the jump parameters. The memory usage of the jump links is shown in megabytes.

	1	2	4	6	8	10	12	14	16
Config 3	99.23	24.85	6.28	2.80	1.58	1.03	0.71	0.52	0.40
Config 2	72.56	18.16	4.58	2.05	1.16	0.75	0.52	0.38	0.30
Config 1	22.23	5.58	1.41	0.63	0.36	0.23	0.16	0.12	0.09

Table 12: Test results for the `Single Player Map`. The columns of represent the different step sizes, and the rows represent the three different configurations of the jump parameters. The memory usage of the jump links is shown in megabytes.

	1	2	4	6	8	10	12	14	16
Config 3	186.62	46.89	11.83	5.28	3.01	1.90	1.33	1.00	0.74
Config 2	131.18	32.96	8.32	3.72	2.11	1.34	0.94	0.70	0.52
Config 1	39.49	9.92	2.52	1.13	0.64	0.40	0.29	0.21	0.16