Automatic Generation of Jump Links in Arbitrary 3D Environments for Navigation Meshes

Diplomarbeit

zur Erlangung des akademischen Grades
Diplominformatikerin

Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik

eingereicht von:	Sara Budde
geboren am:	22.10.1981
in:	Iserlohn

Gutachter:	Prof. Dr.-Ing. Peter Eisert
	Prof. Dr.-Ing. David Strippgen

eingereicht am: ..... verteidigt am: .....

Abstract

This thesis presents a robust solution for the generation of jump link data for virtual agents in arbitrary three-dimensional environments. This jump link data will be integrated into the navigation mesh data structure by designing a smart data model that fits the logic of the navigation mesh as well as the dynamic nature of the jump data. The developed methods will be shown to be fast, real time capable and consistently delivering a dense network of jumps even in very complex environments.

Contents

1 Introduction

This thesis will present a solution for enabling virtual artificial intelligence (AI) agents to perform accurate jumps in complex three-dimensional environments. The solution will automatically generate jump links which can then be used by the AI agents to perform accurate jumps. Using these jump links in conjunction with a navigation mesh will enable the AI agents to use jumps in their navigation queries to find shorter paths or to reach locations that are only accessible by jumping. To put it in a nutshell, we want to enable an AI agent to jump from one roof to another to prevent him from running five floors down, across the alley and five floors up again.

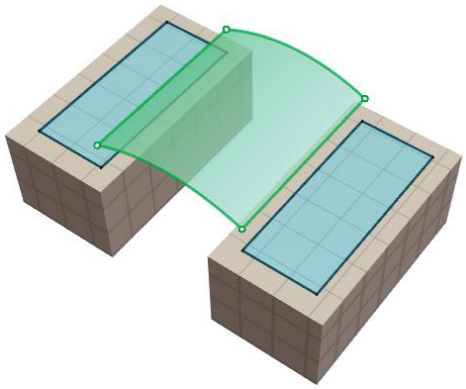


Figure 1: *Screenshot of two parts of a navigation mesh connected by a jump link.*

Figure 1 shows a simple example of a jump link. Visualized in blue are two separated parts of walkable surface which result from two separate boxes

floating below them. The green curved quadrangle represents a jump link which connects the two walkable surfaces.

Artificial Intelligence in the field of virtual agents has developed rapidly in the past years and efficient pathfinding algorithms and data structures have been extensively studied in the academic world. Navigation meshes are one of the state-of-the-art data models that are being used for virtual agent navigation and there is academic literature as well as industry standard implementations on the topic of automatic generation of navigation meshes from polygonal worlds. However, these navigation meshes only cover the walkable surface of an environment. By not only automatically generating the jump information but also encoding it in a manner that integrates well into the existing data structure of navigation meshes, this thesis aims at enhancing the AI's terrain reasoning capabilities to enable him to move beyond the walkable surface of his environment.

As an example, figure 2 presents a complex three-dimensional world from the game *Counter-Strike: Source* [Val13]. The navigation mesh is colored in blue and the curved quadrangles in green, orange and yellow are jump links generated by the solution presented in this thesis. The density of the jump links makes clear that the number of jump links found by our methods add quite some navigational information to the knowledge base of a virtual agent. Placing context information like jump links into the virtual world is called world annotation.

The jump links can also be utilized for player control support. That is when a virtual character is controlled by a human, the AI data is used to anticipate certain complex movements the player intends to initiate, thus enabling the player to trigger a much wider range of actions with a very limited control interface [Spl13]. This usage of world annotation to assist the player in the

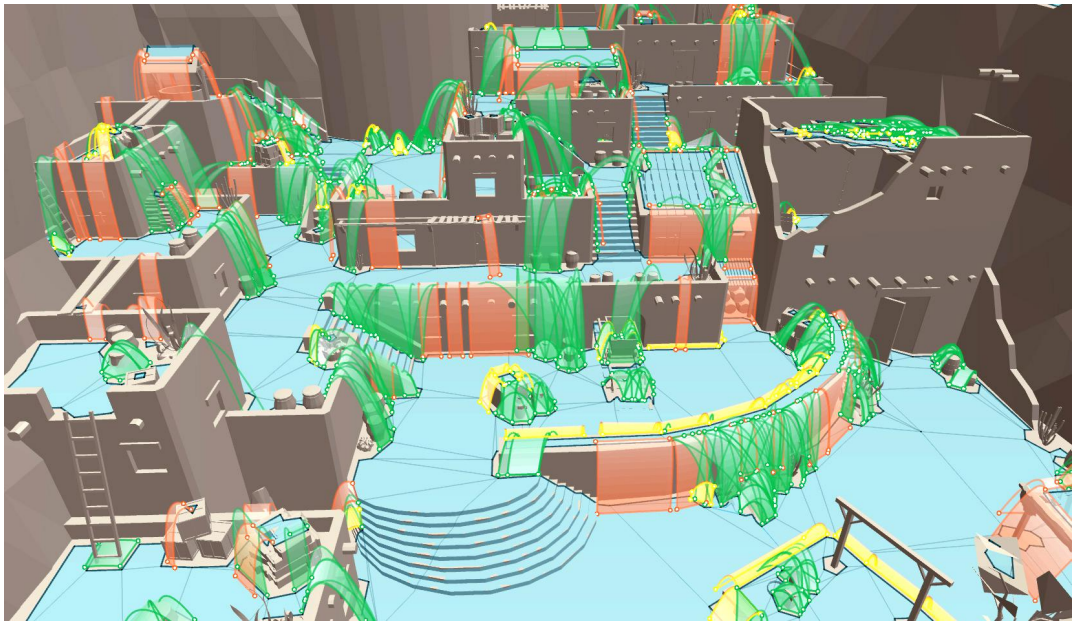


Figure 2: *Screenshot of the jump links found by this thesis methods for the test environment "cs_desperados".*

execution of complex movements in close interaction with the geometry gets more and more popular in the entertainment industry.

Advanced movement technology is already a major part of the product value and the unique selling points of some multiple million dollar entertainment titles. An exemplary product is *Brink* with its "SMART" acrobatic movement system which is one of the aspects of the game that set it apart. "SMART is Brink's most successful innovation."[Pea11]

The quest to give agents the ability to jump realistically is around for some time and automating the process of jump link generation [Far06] and eventually realizing a real-time solution is one of the next vital steps in extending an agent's ability to navigate through virtual worlds.

Section 2 will describe the fundamental data structure and existing methods to create jump links. The details of the jump link generation presented in this thesis will be defined in section 3. Section 4 will analyze the search space for jump links and the deduced tests will be described in section 5 and 6. The results of the jump link generation will be evaluated in section 7.

2 State of the Art

2.1 Navigation Mesh

In the next sections we will have a look at different approaches to generate jump annotations, which have mostly been realized for video games. But to begin with, we will describe the fundamental data structure that this thesis will build upon: the navigation mesh.

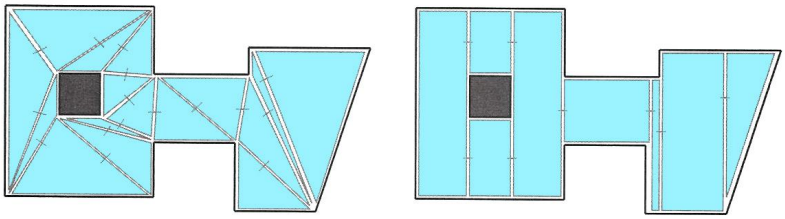


Figure 3: *Schematic view of two navigation meshes. From [Toz04] colored afterwards.*

Navigation meshes were first described in [Sno00] and are now the favored search space representation for pathfinding [MS08] and terrain reasoning [Toz08]. A navigation mesh is a graph with nodes, that represent the traversable surfaces of the level geometry, and edges, which describe the traversability from one traversable surface to another [Sno00; Toz04]. The traversable surfaces are convex polygons, so that in these surfaces there can be maneuvered without any pathfinding look ups apart from dynamic objects [Sno00; Toz04].

In figure 3 by Snook, which has been colored afterwards, two different navigation meshes are visualized. The left navigation mesh, colored in blue, only consists of triangles, whereas the navigation mesh nodes on the right

mainly consists of quadrangles. The navigation mesh edges in figure 3 are presented as small line-segments across shared outlines of the navigation mesh polygons.

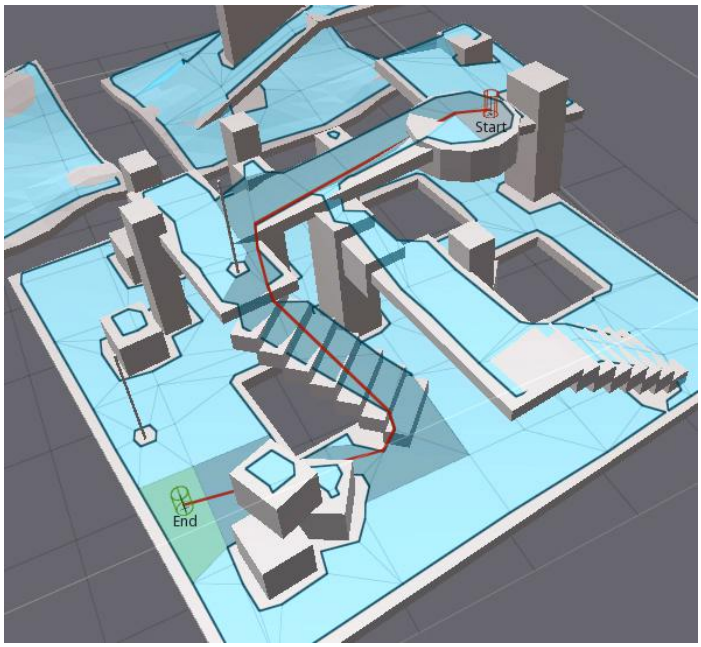


Figure 4: *An example of geometry with its navigation mesh. From [Mon12].*

The navigation mesh covers the whole walkable surface except for a little margin to the walls, to obstructions and to the outline of the geometry as shown in figure 4 and figure 6. The agent obviously has a width and that width has to be considered in the navigation mesh, preventing the agent from partially standing inside of an object. Since the agent's origin is usually in the middle of his feet and the navigation mesh should be constructed in such a way that the agent can stand at every point inside of a navigation mesh polygon, the navigation mesh margin has to measure half the agent's

width. Figure 5 shows a navigation mesh in blue consisting of two polygons. The gray circle symbolizes the agent with the cross "×" marking his origin. The dimensions of the agent extend exactly to the outer edge of the geometry in figure 5. If the agent was positioned outside of the navigation mesh, his extends would exceed the geometry or he would partially be included in an object. Only if the agent, represented by his origin, is positioned at any point inside of the navigation mesh, it is guaranteed that he does not collide with anything else than the floor.

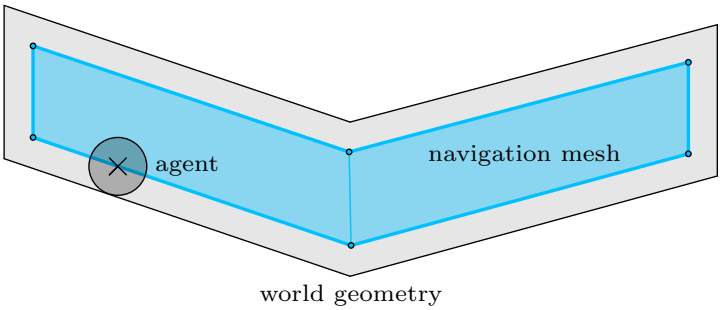


Figure 5: *Sketch of an agent standing inside a navigation mesh.*

There are several different methods to create navigation meshes, like Movement-Based Expansion, voxelization and subdivision routines methods [Axe08; Toz02]. Even several products exist that automatically generate navigation meshes for arbitrary world geometry like "Xaitment" [xai] and "Recast" [Mon12]. The primary use of navigation meshes is of course pathfinding, which is a well researched subject in the context of artificial intelligence. Pathfinding algorithms like Dijkstra's algorithm, A* search algorithm and hierarchical Path-Finding A* algorithm are well-studied. The resulting paths will usually be post processed by path smoothing algorithms.

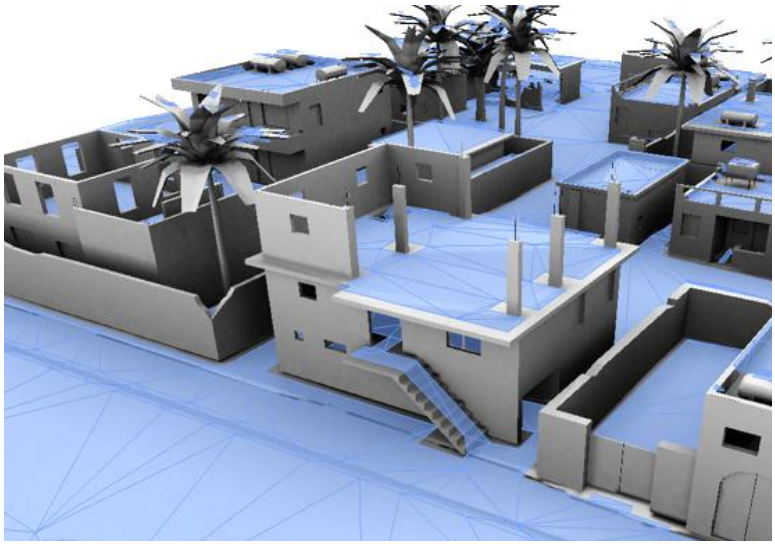
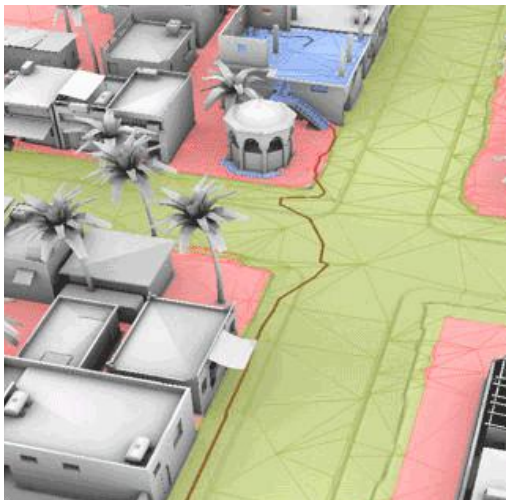
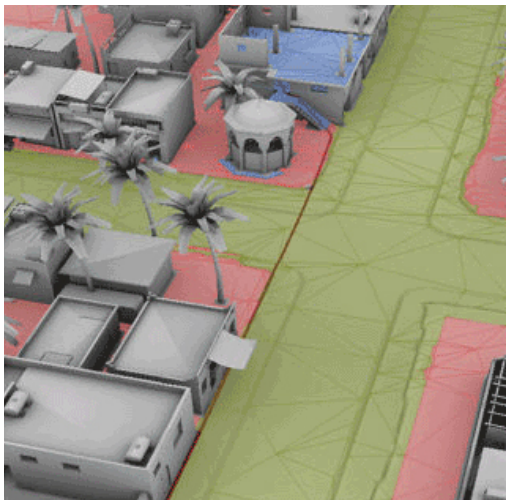


Figure 6: *A navigation mesh example. From [xai].*

Path smoothing will be relevant in section 4.7 that addresses the integration and storage of jump links in a navigation mesh. Thus, the principle of path smoothing will be described here. A path returned by one of the pathfinding algorithms is usually edged and not straight and smooth. Figure 7(a) shows an unsmoothed path colored in a dark red from the house at the top with the blue navigation mesh to the street at the bottom of the figure. Especially where the path crosses the intersecting road, the angularity of the path is visible. This angularity results from the structure of the calculated movement through the polygons of the navigation mesh, which can go through the middle of a polygon or the middle of a shared edge [Pat12]. Path smoothing is a post-processing that modifies the path so that it appeals more realistic and does not always lead through the middle of a polygon or the middle of a shared edge. This is done by applying *smooth straight-line movement*, adding *smooth turns* and *legal turns* [Pin01]. For instance figure 7 visualizes the path smoothing.



(a) *A path before path smoothing.*



(b) *A path after path smoothing.*

Figure 7: *An example of path smoothing. From [xai].*

2.2 Manual Annotation

Manual annotation in the context of jump capabilities for agents means that designers place the jump links in locations where they think they are appropriate. Basically the start and the landing point are defined by the designer, normally with some restriction to the placement, depending on the number of jump trajectory animations and their flexibility. This way of placing jump links is very time consuming and has to be redone in the event that the environment around the jump link is changed. It is a robust technique for a controlled and small environment, but it is not a practical solution in the sense of a realistic representation of human jump capabilities. Because this would involve a lot more jump links than manual annotation can deliver in an acceptable work time. Figure 8 shows an example of how manual jump links are placed in the *CryEngine 3* development kit [Cry].

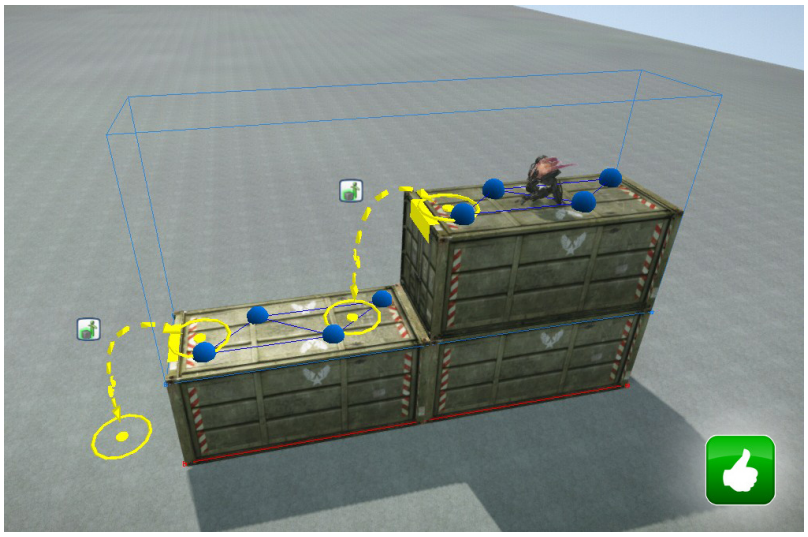


Figure 8: *Screenshot of a jump object (yellow) in the CryEngine 3 development kit. From [Cry]*

2.3 Movement-Based Expansion Method

The movement-based expansion method works like a recursive flood fill algorithm, which can be used to generate navigation data as well as jump links. Beginning with a given point, a movement is simulated and if the movement is positively executed, the new position is added to the movement data. Jumps can be simulated the same way and if they are positively performed, a jump link can be added to the navigation data. [Axe08; Smi02]

This approach is rather theoretical because the size of state-of-the-art worlds and the amount of different jumps, like a long jump, a jump down and a jump up, result in an extremely long computation time [Axe08]. Since jump links are generated for every possible location, a high number of jumps has to be stored resulting in a high memory requirement or a post processing to determine useful jumps. Therefore, this method is more theoretical than

practical while all further introduced approaches have been implemented and have been used in video game production.

2.4 Jumps for *Quake III Arena* Bots

Quake III Arena was developed by *id Software* and first published 1999 [Wik13]. This video game is a multiplayer first person shooter which can be played with and against other players as well as virtual players, also called bots. *Quake III Arena* has implemented an Area Awareness System (AAS) that includes the search space representation for pathfinding queries. The AAS consists of areas which describe the walkable surface and the surface where agents can swim. These areas are connected by reachabilities. Additionally, the AAS contains information about other entities of the game [Wav01]. This system contains more information than just the navigation information like a navigation mesh.

There are different tests for generating reachabilities like jumping onto a barrier, walking off a ledge and jumping over a gap. Reachabilities are connecting different areas. So for a jump onto a barrier, two borders of areas are needed which lie in one perfectly vertical plane and are vertically overlapping each other. Between such area borders, a jump onto a barrier is generated if the borders have a certain distance and the jump volume is not blocked by any obstacle. The tests for walking off a ledge and jumping over a gap are similar. In order to check two edges of different areas for jump reachabilities, first, the closest points of these edges are determined. If there is a line-segment closest to the other edge, the middle of this line-segment is used as the closest point. A jump between these points is simulated to check whether the jump is possible and unobstructed. After this has positively been tested, a jump reachability is generated as shown in figure 9.

Figure 9: *Screenshot of a jump reachability in Quake III Arena. From [Wav01].*

In *Quake III Arena* different areas are connected by jumps with point to point reachabilities. In general, point to point connections are inflexible in the post processing of path smoothing because the takeoff and landing points are fixed and cannot be adjusted to the smoothed path. Applied to figure 9, an agent who was supposed to move from the left bottom of the red area to the top right corner of the pink area would use a path up to the takeoff point, then jump and continue up to the goal instead of using the shortest way to the goal by jumping diagonally from on area to the other.

The test method used for jump reachabilities does not contain any contingency alternatives for obstructions. This means that in case of an obstruction, no reachability is generated even if a jump between the non-closest points of the areas would be possible. Also for jumping onto a barrier and walking off a ledge, only a perfectly vertical gap can be overcome while even along small slants no reachabilities are found. Furthermore, jumps down into an area are not possible because in this situation edges exactly vertical over each other do not exist. Overall, reachabilities are only generated in defined situations.

2.5 Smooth Movement Across Random Terrain in *Brink*

In 2011 *Splash Damage* published *Brink*, a first person shooter with a acrobatic movement system, which is called Smooth Movement Across Random Terrain (SMART) [Spl13]. SMART enables the players to perform different kinds of jumps depending on the geometry in front of them and where the player wants to go. Therefore, annotations are stored within the virtual world [Hal12].

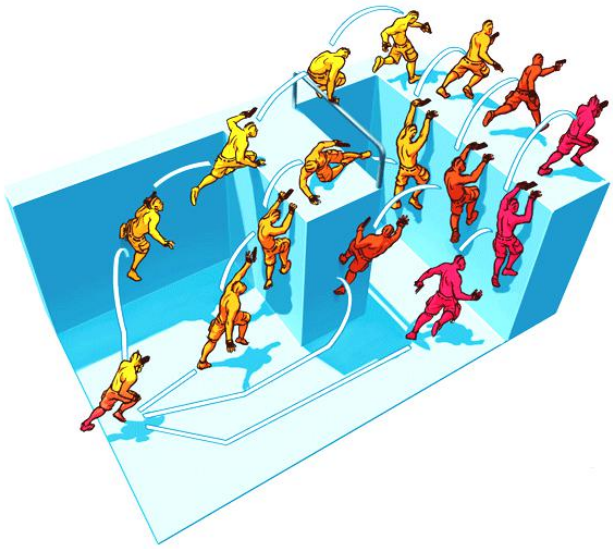


Figure 10: *Different movement capabilities supported by SMART. From [Bri12].*

In figure 10 the different movement possibilities are visualized. The movements on the left path are a wall hop and a vault while the path in the middle

comprises a pullup, a slide and another pullup. The other path in the middle consists of the actions wall hop and pullup and on the rightmost path a simple pullup is performed.

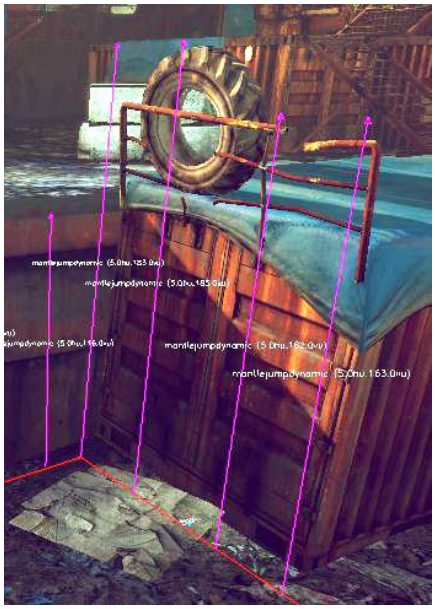
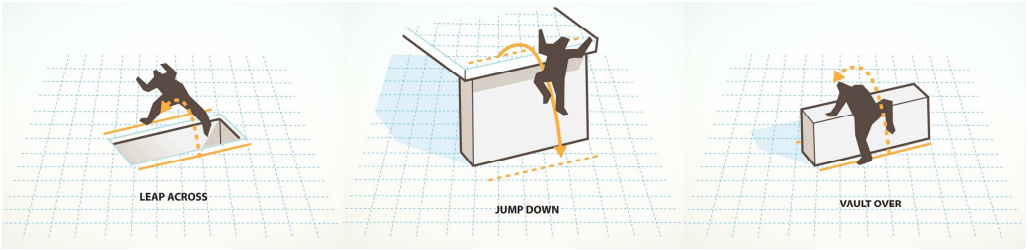


Figure 11: *Screenshot with pullup reachabilities of Brink. From [Hal12].*

These movements can only be performed where the corresponding reachabilities are set in the virtual world. These point to point connecting annotations are shown in figure 11. As described earlier for the jump reachabilities of Quake III Arena, point to point annotations are inflexible for path smoothing. For all reachabilities other than slides will be searched where edges of the walkable surface overlap vertically [Hal12]. This means that no reachabilities will be found if the edges do not overlap perfectly in the vertical dimension.

2.6 Jump Annotations for *Killzone 3*

Killzone 3 is a first person shooter developed by *Guerrilla Games* and published in February 2011 [Son10], which uses automatically generated annotations. Among others, leap, jump down and vault annotations have been created as visualized in figure 12. In further considerations we will focus on leap and jump down annotations because vaulting actions depend on finding some cover over which a vault is possible and this thesis does not regard cover detection.



(a) *leap over a gap* ***(b)*** *jump down a ledge* ***(c)*** *vault over cover*

Figure 12: *Different jump capabilities of Killzone 3. From [Mon11b].*

The implementation consists of two tests, one for leaping and one for jumping down. Each outline in a navigation mesh is tested for possible jumps down while for leap annotations every outline pair is tested if a jump is possible. The collision test is based on a voxel representation of the world geometry. Figure 13 shows a jump volume which is tested for obstructions. If there is no obstruction, the outlines are connected by a jump link, also called off-mesh connection.

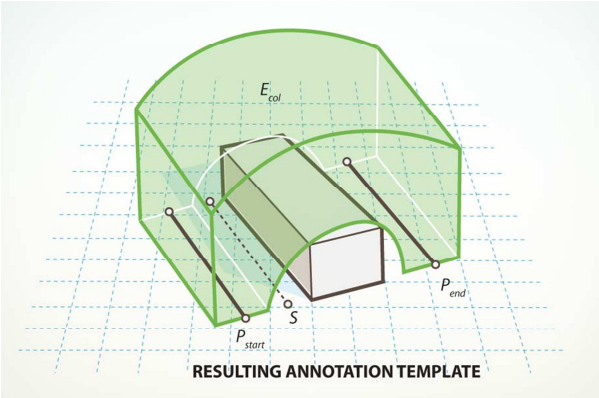


Figure 13: *Schematic of a jump collision volume. From [Mon11b]*

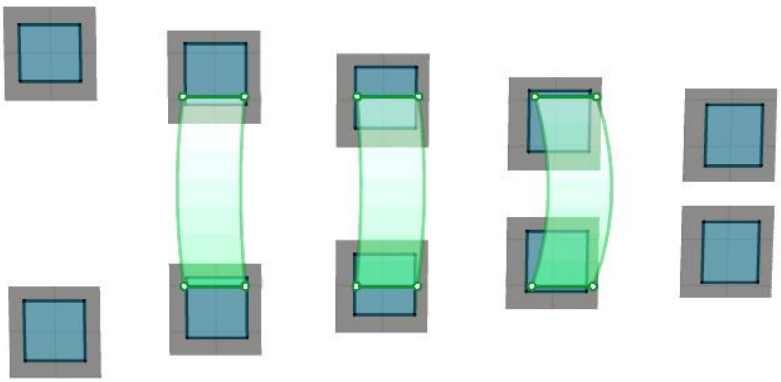
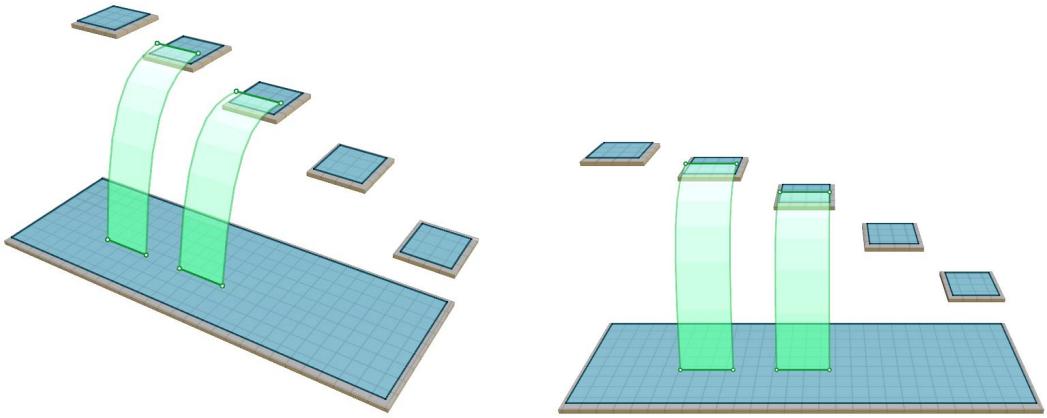


Figure 14: *Issues with leap annotations caused by only using a single trajectory (top view).*

The test to find leap annotations places one fixed trajectory in the middle between the two outline edges which we test. The trajectory can be stretched and compressed a bit to adjust it to the actual situation of the navigation mesh outlines, but only a small adjustment is possible. Therefore, if the gap between the navigation mesh outlines is much smaller than the width of the fixed trajectory, the takeoff points and the landing points are shifted away from the gap and are lying somewhere in the two areas. However, if the gap between these outlines is small and also the areas are small, no off-mesh connection will be found because the jump with the fixed trajectory is wider than the whole areas with the gap. In figure 14 this issue is visualized, showing five situations with two walkable areas in blue and the jump links between them in green. In the first situation from the left, the gap between the walkable surfaces is too large for a jump link to be generated. In situations two to four, the generated jump links do not adapt to the shrinking gap, instead their starting points and ending points move towards the outer edge of the walkable areas. In the last situation on the right, the gap is even smaller, resulting in the tested jump trajectory to start and finish outside the blue navigation mesh polygon so that no jump is found.

The off-mesh connections for jumps down are also tested with a fixed trajectory, meaning that the jumps down must have a certain height and neither higher nor lower jump annotations can be generated. In figure 15 this aspect is visualized. On the left of the figure no jump is possible because it is higher than the maximum falling depth. In the middle, jump annotations are found, but on the right of the figure, where the platforms are nearest, no off-mesh connections are produced. This comes for example into play when a jump down is tested along a ramp as shown in figure 16. Only a slim jump on the higher part is found, and along the rest of the ramp the smaller jumps are not found.



(a) *perspective view* ***(b)*** *front view*

Figure 15: *Issues with jump down links caused by using only a single trajectory and a small landing height tolerance.*

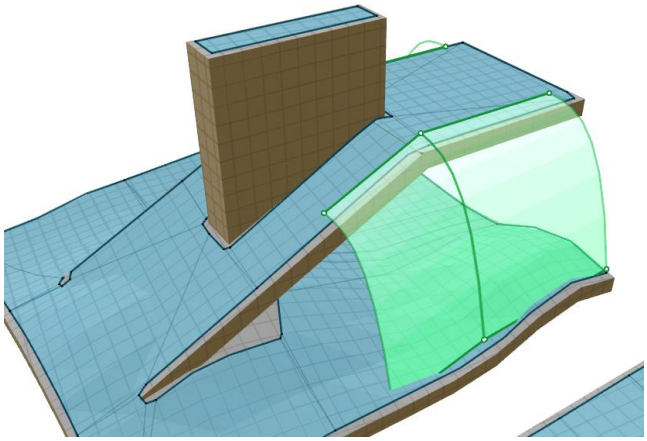


Figure 16: *Restrictive jump down test at the example of a ramp.*

This restriction to two tests through a fixed trajectory results from practical issues in the game development pipeline. For each performable jump an animation is needed and the animation and jump action have to be integrated into the product. This involves several different professions and a lot of work, which limits the amount of complexity that is profitable for a single product [Mon13]. The *Killzone 3* implementation automatically generates jump annotations with a reasonably reduced search space. The generated jump links are no point to point connections, so that jumps can be flexibly modified during the path smoothing process. The off-mesh connections are constructed between areas on nearly the same height and between areas below each other; in all other cases no jump annotations will be found. Moreover, if there are any obstacles obstructing the jump volume, no other jump trajectories are tested. For example, if a higher jump would jump over the obstacle, this would not be found.

Figure 17 shows an environment including off-mesh connections which have been generated with the technology implemented in Killzone 3. The next section will outline the differences to the above introduced methods for jump link generation and define the details of automatic jump link generation which this thesis studies.

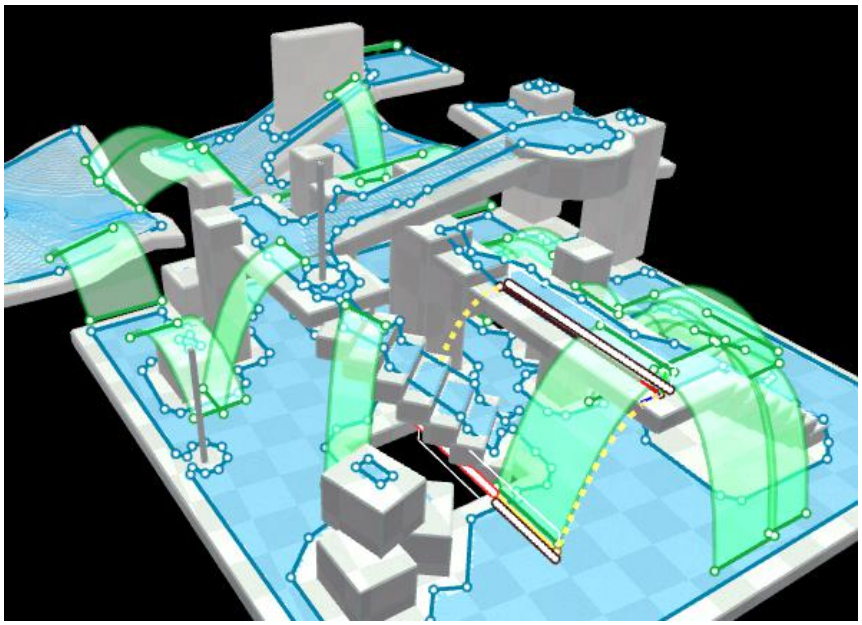


Figure 17: *Off-mesh connections found with the technology of Killzone 3. From [Mon11a].*

3 Problem Definition

3.1 Automated Jump Link Generation

This thesis will present a solution for automatic jump link generation, empowering the agent with new movement capabilities. There are several kind of movements that can be categorized as jumps. In this thesis though, we will focus on jumps that are performed in a usual jump stance, which means that during a jump from a takeoff point to a landing point nothing is touched in contrast to vaulting over something and somersaulting.

There exists an enormous number of jumps that could be generated, but we are only interested in jumps which enable the agent to move along new paths. Of course we want to find jumps that allow the agent to access areas, which are unreachable without jumps, such as two unconnected islands as visualized in figure 18(a). Figure 18(b) shows a passage from one platform to another one, but there is no direct contact between these platforms. A jump link between these platforms is intended to provide a shortcut between them, but there also exist unwanted jumps, like jumps that start and land in the same free square. In these cases a straight walkable way from the takeoff point to the landing point exists, making those jumps unnecessary. Also redundant jump links, like several jump links between one pair of takeoff and landing point, are undesirable. These jumps are not wanted because they do not enable the agent with new and advantageous movement capabilities. However, we also want to find jumps that provide an alternative path like the jump alongside a bridge shown in figure 18(c). Obviously the jump alongside the bridge would take longer than walking over the bridge, but in case the narrow bridge is occupied by another agent, an alternative path like a jump alongside the bridge would be of advantage. Alternative paths

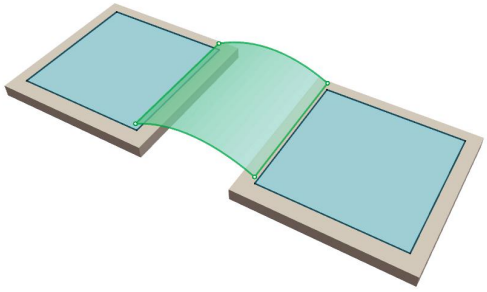
thereby allow the agent to react properly to dynamic obstacles like other agents in the world geometry. In a nutshell, the solution presented by this thesis is supposed to find useful jumps that provide the agent with new and alternative movement capabilities.

The solution should be completely automated, so that it can be easily integrated in the process of the navigation mesh generation. Since the navigation mesh generation is often used in an iterative process by designers, we want a consistent jump link generation, so that local changes in the world geometry and in the navigation mesh lead to other jump links adapted to the new local environment. The solution will use tunable variables, which can be defined individually for each application, ensuring that the user will get optimal results for his scenario. Apart from defining these tunable variables, there are no inputs required by the user, because the generation of the jump links will be completely automated.

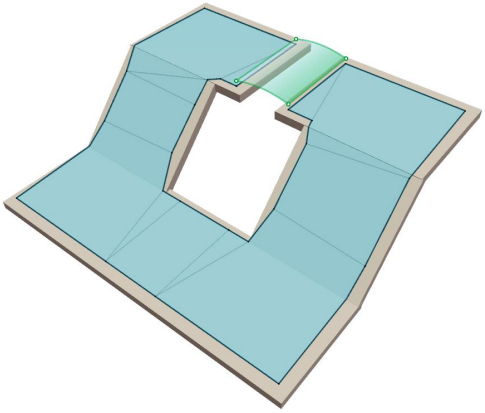
We want a solution with a reasonable computation time to make even real time application possible. However, the presented solution will be designed to find almost every possible useful jump. The number of found jumps will be examined in this thesis. Basically, we want a method that tests for specific jumps between points and does not simply apply a brute force search for every possible jump. This will clearly help to differ between useful jumps and jump links we do not want to find. Thereby, we will get a reasonable generation time for the jump links.

3.2 Jump Links with Variable Jump Trajectories

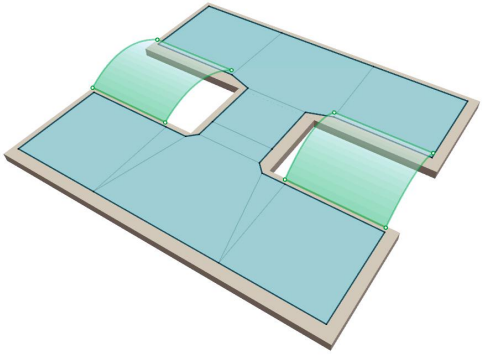
This thesis deals with the generation of jump links, which are a data structure storing jump information. However, we do not want to generate jump links



(a) *unconnected islands*



(b) *shortcut*



(c) *alternative paths*

Figure 18: *Different scenarios of wanted jump connections. Only the jumps that are exemplary for the shown scenario are rendered for clarity. The methods presented in this thesis actually find several more jumps.*

that connect only two points, resulting in a jump link that represents exactly one jump, like the annotation generation presented in section 2 for *Quake III Arena* and *Brink*. Instead, we want a jump link to connect two line-segments, so that between these edges both straight and diagonal jumps are possible. This means that a jump link is not directly associated to one specific jump but rather represents a set of jumps.

The generation of jump links should not require that the world geometry fits a specific pattern to find a jump. On the contrary, jump links should be found easily and without any restrictions to the world geometry. Therefore, the solution needs to support a lot of different jump trajectories with different jump heights and jump distances. That way, a suitable jump trajectory can be chosen, depending on the world geometry. Moreover, we can test several trajectories between a pair of takeoff and landing points, thereby avoiding obstructions by the means of alternative jump trajectories. To make this possible, the solution has to support a lot of variable trajectories, which is one main difference to the jump link generation processes of *Quake III Arena*, *Brink* and *Killzone 3* presented in section 2, which are based on one or a very small number of jump trajectories.

Because we do not have a fixed number of trajectories, our solution does not simply have to test whether a jump is possible, but it has to be checked which of the many jumps between a certain takeoff point and a chosen landing point is possible. This means that the solution does not test blindly, but has a specific test that checks for jumps between takeoff line-segments and landing line-segments.

3.3 Integration of the Jump Links into the Navigation Mesh

There are two majorly different approaches to enable an agent to jump precisely. One is testing the jump possibilities in real time during the pathfinding

of the agent. The developers of *Brink* stated that these tests were not real time capable for 16 players [Hal12]. Considering that players do not have a path planning phase and only need the test for control support when they initiate a jump, the real time jump tests are definitely not possible for AI agents. The other approach is to preprocess the jump possibilities in the environment prior to the individual path finding queries of the AI agents. Important to note is that the jump link preprocessing itself can be real time capable but not if all jump links are calculated for every pathfinding query. Instead, the environment is reexamined in real time when it changes, so the jump link data is already processed when the pathfinding starts and can be used in multiple queries.

One criterion for a jump link should clearly be that the agent can stand at the takeoff point and the landing point of a jump. Consequently, all takeoff and landing points have to be inside a navigation mesh of the world geometry. Therefore, the solution uses a navigation mesh as well as the world geometry as data input. Many navigation mesh generations are designed to work for arbitrary worlds. For a simple integration of the automated jump link generation we do not want to constrain these arbitrary worlds, for which navigation meshes are generated. Basically, we want a robust solution that can handle arbitrary worlds.

The tool "Recast"[Mon12] is a robust solution for the generation of three-dimensional navigation meshes. It is based on voxelization and will be used to create the navigation mesh for the presented solution. Furthermore, voxel data, created for the navigation mesh generation, will be used for the collision tests within the jump link generation. More precisely, the jump link generation will use a Boolean collision test which tests for obstruction in a certain distance over a two-dimensional curve placed in the three-dimensional world.

For the jump link data to be easily accessible by pathfinding algorithms, we store the jump links in conjunction with the navigation mesh. Thus, the output of the solution is a navigation mesh which has been extended by the information of the jump links. Different kinds of movement like walking, crawling and crouching are often encoded in the navigation mesh [BSL04; Fun09], which means doing the same with jump data follows a proven concept. The storage of the jump links will be discussed in more detail in section 4.7. In the next section though, we will first examine the navigation mesh and where the wanted jumps' takeoff and landing points are located.

4 Analysis of the Jump Problem Space

4.1 Introduction

In the previous section we have already discussed that we want to find jumps representing connections among separate navigation mesh parts, shortcuts between logical points and alternative paths. This section will analyse the space where jumps could be performed.

First, we will classify jumps according to their takeoff and landing positions and thereafter we will discuss the wanted jump trajectory. Combining these information, we will deduce where the solution has to search for the jumps we want to find.

4.2 Jump Classification

As described in section 3.3, we use a navigation mesh as the fundamental data structure for the walkable areas of the world geometry. Obviously jumps should take off and land somewhere the agent is able to walk, not to mention stand. Therefore, takeoff and landing positions have to lie inside of the navigation mesh.

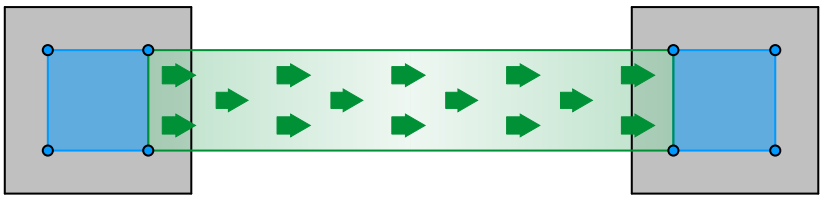
A navigation mesh is a graph with polygons as nodes and shared polygon edges as graph edges. Points in a navigation mesh can lie either inside of a polygon or on the outline of the polygon. In the latter case, the point has to lie on one edge of the polygon. The positions of the takeoff point and the landing point of a jump define to which of the following classes the jump belongs, namely to

- Jump from Edge onto Edge,
- Jump from Edge into Polygon,
- Jump from Polygon onto Edge and
- Jump from Polygon into Polygon.

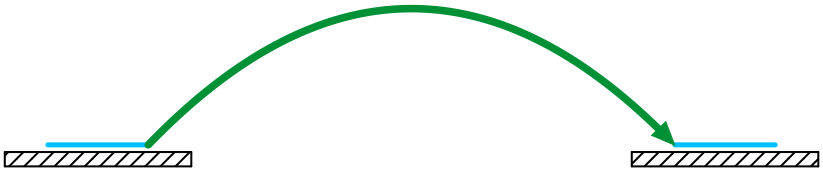
These classes will now be described in more detail and with examples.

Jump from Edge onto Edge

The class *jump from edge onto edge* contains jumps whose takeoff and landing points are lying on edges of the navigation mesh polygons. Figure 19 shows a schematic example of a jump from an edge of one navigation mesh polygon onto an edge of another polygon.



(a) *top view*



(b) *side view*

Figure 19: *Schematics of a jump from edge onto edge.*

An example of a jump from one edge to another edge is the far jump from

a roof ledge to the roof ledge on the other side of the street or a jump from one wooden post to another.

Jump from Edge into Polygon

Jumps of this class start from an edge like in the class *jump from edge onto edge*, but they end inside of a navigation mesh polygon as visualized in figure 20.

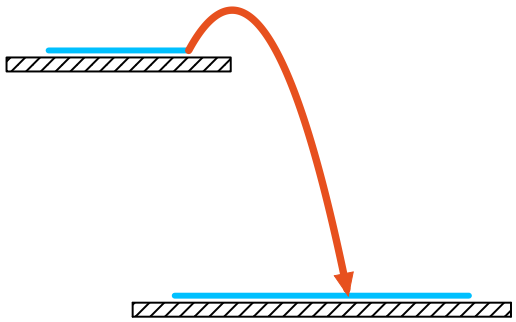


Figure 20: *Schematic of a jump from edge into polygon.*

For instance, a hop from a windowsill onto a jumping blanket or from the vaulting horse onto a mat are jumps of this class. Other examples are jumps from tree branches or bridges onto the ground beneath them.

Jump from Polygon onto Edge

This class contains the reverse jumps of the class *jump from edge into polygon*. These jumps take off inside of a polygon and end on an edge of another navigation mesh polygon.

Figure 21 schematically shows such a jump. Examples could be a jump from a balcony onto a windowsill or from the ground onto the edge of a box.

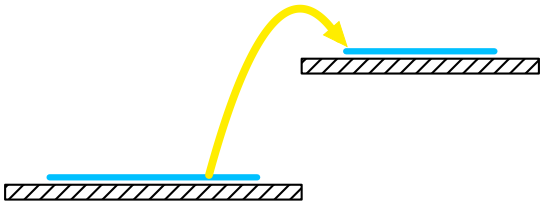


Figure 21: *Schematic of a jump from polygon onto edge.*

Jump from Polygon into Polygon

Figure 22: *Schematic of a jump from polygon into polygon.*

Jumps of this category take off and land inside of a navigation mesh polygon. Such a jump is visualized in figure 22. Examples include jumps from a trampoline onto a mat or a hop in the children's game hopscotch from one square to another.

4.3 Definition of the Optimal Jump

In this section we will take a closer look at what kind of jumps we want to find and how these jumps look like. In section 3 we described three kinds of jumps we want to find, one of which was the shortcut between two platforms.

Between these platforms, there are a lot of different jumps possible, which is shown in figure 23. The jump possibilities vary in their height as well as their length. This means that between these two platforms there are jumps that differ both in takeoff and landing point (green and blue), jumps that differ either in their takeoff point (green and yellow) or in their landing point (yellow and blue) as well as jumps with the same takeoff and landing points but a different height (green and purple).

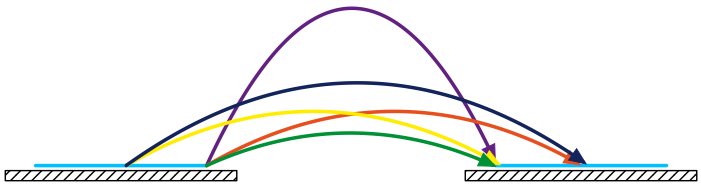


Figure 23: *Schematic of the two platforms and several totally different jumps between them.*

If these platforms were far away from each other there would obviously just be the flat jump from the ledge of one platform to the border of the other platform (green) possible. In the case that the takeoff point and the landing point are wooden posts, we do not wish an unnecessarily high jump between these positions, and a jump down out of a window to get back into the house should clearly be as short as possible. In these examples we can see that the wanted and most natural jump is always as short as possible and as flat as possible. The desired jump connects the closest points of two positions with the shortest jump trajectory that connects these points. We call this preferred jump the optimal jump.

There do exist situations though, in which the optimal jump is not the best jump. For example if an agent jumps from one roof to another roof which is

significantly lower and close in order to climb down a ladder at the far end of this second roof, we would expect the agent to jump, roll and run further towards the ladder. In these cases, when the further path of the agent has the same direction as the jump, a far jump could be more believable than the short optimal jump. Nevertheless, the optimal jump is not unbelievable because after the jump, the agent simply has to walk across the rest of the roof. On the other hand, in all situations where the jump direction does not match with the further path of the agent, jumps that are further than the optimal jump would be very unbelievable, because the whole distance covered by the jump has to be walked back. Therefore, the optimal jump is not always the most believable, but never an unbelievable jump.

4.4 Jump Trajectories and their Lookup Table

Up to now we have only specified that the jump trajectory connecting two positions should be short and flat. In this section the trajectory will be described more formally. An agent who performs a jump uses his takeoff velocity and angle to land at his desired location. During the flight of a jump, the trajectory is influenced by the gravitational acceleration g and the air drag. A long jump can be represented by two different parabolas, which are merged at their maximum [Mül04]. We ignore the air drag, because its impact heavily depends on the posture of the agent which we do not want to constraint. Furthermore we want our data model to support bidirectional jumps in a certain tolerance to reduce unnecessary data complexity. Therefore, we use the trajectory of a projectile, which is defined by

$$trajectory_{v_0,\theta}(x) = \frac{-g}{2 \cdot (v_0)^2 \cdot \cos^2(\theta)} \cdot x^2 + \tan(\theta) \cdot x + y_0 \,,$$

where v_0 is the launch velocity or in our case the takeoff velocity, θ is the launch angle and y_0 corresponds to the height difference between takeoff and

landing location [Fen03].

In section 3.2 we stated that we want to have a specific test. This means that the test will check if a jump is possible between two points. In conclusion, the potential takeoff and landing points are provided to be tested. Therefore, all trajectories $Trajectories_{(x',y')}$, which connect the takeoff point $(0, 0)$ and the landing point (x', y') are given by

$$Trajectories_{(x',y')} = \left\{ (v_0, \theta) \middle| \theta = \arctan\left(\frac{v_0^2 \pm \sqrt{v_0^4 - g\left(g\,(x')^2 + 2y'v_0^2\right)}}{gx'} \right) \right\} .$$

The previous equation of the set definition only has a solution if the root term $\sqrt{v_0^4 - g\left(g\,(x')^2 + 2y'v_0^2\right)}$ is a real number. This means that the launch velocity v_0 has to be great enough to reach the landing point. For each launch velocity v_0 that is great enough, there exist two launch angles θ. The main difference between these two jump trajectories is the maximum jump height. So if the flatter jump trajectory is blocked by an obstruction, we can still find a valid jump based on the other jump trajectory. For Instance in figure 24 the flatter trajectory (red) is obstructed by some kind of railing, but the higher jump trajectory (green) can overcome this obstruction and represents an alternative jump trajectory to the flatter jump trajectory. This is the reason why we want alternative trajectories to be tested in our solution, which is done with the jump trajectory lookup table. The lookup table stores all sets of trajectories that connect one specific takeoff point with one specific landing point in a list and is organized in a way that allows access to a certain set of alternative trajectories in constant time.

Considering jump trajectories as two-dimensional curves, the horizontal axis corresponds to the jump width while the jump height is assigned to the

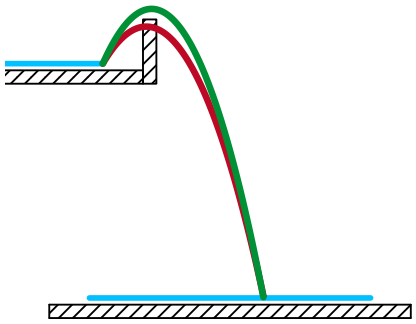


Figure 24: *The minimal jump is obstructed (red), but an alternative jump trajectories (green) can still provide a valid jump.*

vertical axis. We assume that the continuous space is discretised along both axes by user specified step sizes. Therefore, every cell of the jump trajectory lookup table represents a jump width- and a jump height interval. Each of the table's cells is filled with all jump trajectories that pass through the cell $JTLT_{(x',y')}$, which is given by

$$JTLT_{(x',y')} = \bigcup_{\substack{x' \leq x < x' + \Delta x \\ y' \leq y < y' + \Delta y}} Trajectories_{(x,y)} .$$

Figure 25 shows the jump trajectory lookup table with all its cells. Only 1.3% of the jump trajectories that are stored in the table with the later described default settings are visualized as curves over the cells represented by the grid intervals.

For a given takeoff point and a given landing point we can calculate the jump width and jump height. Based on this calculated jump width and -height, we can get the correlating lookup table cell which gives us the set of

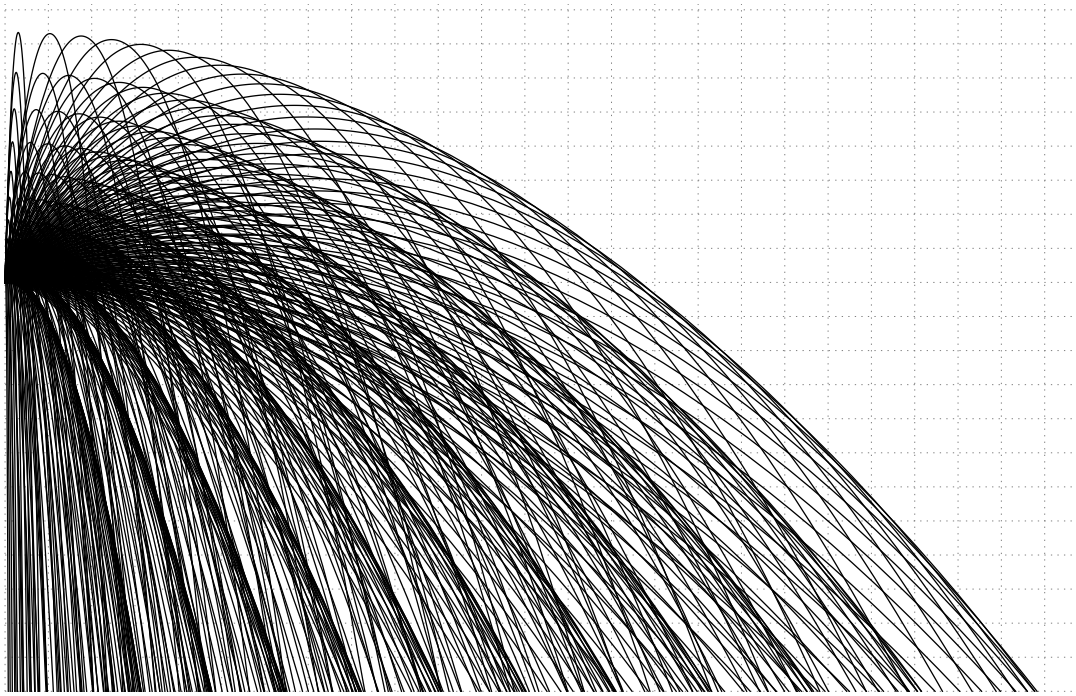


Figure 25: *Jump trajectories drawn over a grid as visualization of the lookup table.*

jump trajectories, which connect the takeoff point and the landing point. If the collision test fails for one jump trajectory of this set, we can test another one of the same set. The jump trajectories, which are stored in one cell of the lookup table, are ordered according to their arc length, which enables us to first test flatter jumps that have a shorter arc length. The arc length of the trajectory $trajectory_{v_0,\theta}$ is described by

$$\int_0^{x_0} \sqrt{1 + \left(trajectory'_{v_0,\theta}(x)\right)^2}\, dx \ ,$$

with the trajectory's $trajectory_{v_0,\theta}$ takeoff point is $(0, y_0)$ and the landing point is $(x_0, 0)$.

The jump trajectory look up table by itself is not just defined by discretisation of the jump width and jump height, but also by the trajectories that are stored in the lookup table. The trajectory's defining variables are the takeoff velocity and the takeoff angle. For our implementation we choose to store all trajectories between a minimal and a maximal velocity and a minimal and a maximal angle in the jump trajectory lookup table. With our default values of the tunable variables we nearly store 600 000 different trajectories in the jump trajectory lookup table. Still, the lookup table gives us fast lookups of desired trajectories for the cost of a rather small memory consumption. In a worst case we have to check all trajectories stored in one cell, which means a linear run time depending on the number of trajectories per cell.

4.5 Study of the Search Space

In section 4.2 we classified jumps based on the position of their takeoff and landing points and in the previous sections we concluded that we want to find optimal jumps. This section will combine this information to reduce the search space of our solution.

Depending on the position of the two polygons to each other, the optimal jump's takeoff point and landing point either lie on the edge of a polygon or inside of the polygon. But as we can see in the example in figure 26, none of the shown optimal jumps has both its takeoff and landing point inside of a polygon. For a jump down the shortest jump always has a takeoff point on the edge of a surface. The same applies to long jumps, but while their shortest jump also has its landing point on the edge of a surface, the jump down ends inside of a polygon if the lower surface extends under the takeoff surface. For a jump up the shortest jump always ends on the edge of a surface. If the takeoff point lies inside a polygon or on the edge it depends on if the landing polygon lies above and overlaps with the takeoff polygon

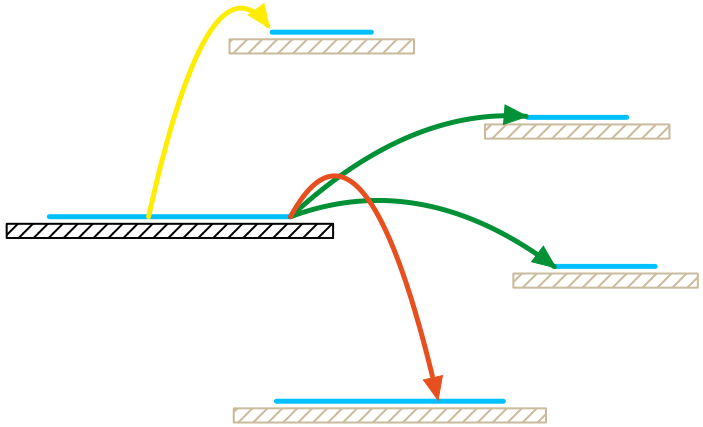


Figure 26: *Sketch of pairs of polygons and the optimal jumps from the start polygon on the left towards the other polygons on the right.*

or if it does not overlap.

To summarize, all jumps we are looking for have a takeoff point or a landing point on an edge. Figure 27 shows that there are two kinds of navigation mesh polygon edges namely those edges that are connecting two polygons and edges which only belong to one polygon. We call them inner and outer edges, with outer edges being the ones that do not connect polygons. A jump should connect positions of different height or positions on unconnected polygons and not two positions between which a straight ground connection already exists. Therefore, the edges that are relevant for the jump test are the outer edges.

In this section we reduced the search space by excluding jumps from polygon into polygon and through the constraint that jumps do not start or land in connection with the inner edges. Outer edges of the navigation mesh

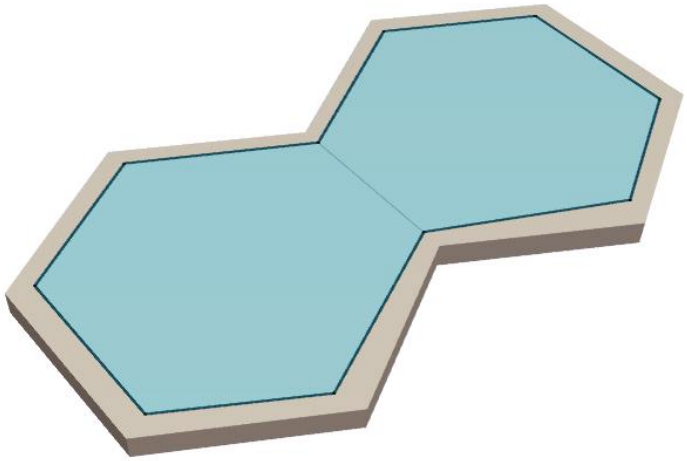


Figure 27: *Screenshot of a navigation mesh to illustrate the difference between shared edges (thin blue line) and outer edges (thick blue lines).*

will simply be referred to as edges in the following. In the next section we will take a closer look at the remaining three classes of jumps and if we can reduce the search space even more.

4.6 Symmetry and Reversibility of Jumps

In the previous section we have seen that wanted jumps belong to one of the three classes *jump from edge onto edge*, *jump from edge into polygon* and *jump from polygon onto edge*. In this section we will discuss the symmetry of jumps and the reversibility of jumps.

In reality a jump trajectory is never symmetrical, owing to the air drag [Mül04]. In our solution though, we ignore the air drag and use trajectories of a projectile as jump trajectories [Fen03]. These trajectories are parabolas, which means that their curve is symmetrical. Furthermore, in reality most

jumps are also doable the other way around, which is visualized in figure 28. Long jumps can mostly be reversed and if a jump up is possible the jump down definitely can be done. However, if we assume that the agent can jump or fall deeper than he can jump up, a possible jump down does not necessarily mean that the jump up is possible as well.

(a) *downwards jump* ***(b)*** *upwards jump*

Figure 28: *Reversibility of jumps.*

As we have seen in figure 26, jumps from a polygon onto the edge of another polygon are always jumps up. Those jumps up whose landing polygon does not lie above the takeoff polygon, as well as jumps down and far jumps all start from the edge of a polygon. We also know that jumps from edge into polygon are always the shortest jumps down. In reality we would not always jump down from a point that is $30cm$ (half the agent's width) away from the edge we want jump down. Instead we would walk off the ledge and make a controlled fall. This is a specialized move that does not allow to jump over obstacles, to vault over obstacles would require another specialized test for close interaction with the obstructing geometry. By jumping from the edges of the navigation mesh, we keep the obstacle avoidance for the takeoff and the complete flight trajectory in one algorithm. This means that the class *jump from edge into polygon* cannot be reduced without loosing wanted jumps. However, we know that a jump up is only possible where a jump down is possible, too. Consequently, we can reduce the search space for the class

jump from polygon onto edge to all those positions where a jump of the class *jump from edge into polygon* has been found.

As a result we fuse the test for the class *jump from edge into polygon* and *jump from polygon onto edge* and call it the *Jump into Polygon Test*, while the test for jumps of the class *jump from edge onto edge* is called *Jump onto Edge Test*. These tests are implemented for jump links and in the next section we will discuss the difference between a jump and a jump link.

4.7 Smart Jump Data Model

In the previous sections we talked about what kind of jumps we want to find and where believable jumps are positioned on a navigation mesh. We assumed that a jump has one takeoff point connected by one specific trajectory. Such a jump has no variability but is exactly one jump, meaning that there is only one way to execute it. Such a model with one takeoff point, one landing point and one trajectory is inflexible in the application, but a way to circumvent this inflexibility is to store many of these jumps. Thereby, we would get a lot of parallel jumps as well as many diagonal jumps across these parallel jumps in one region. This would result in the desired level of detail, but it would also require a lot of memory. This section will demonstrate the advantages of jump links as a smarter way of storing the desired detail level of jumps.

To begin with, we will look at jump links which contain jumps from one edge of the navigation mesh to another. The basic idea is that a jump link represents the set of all possible jumps between two edges. Because there is an endless number of possible takeoff and landing points on a pair of edges

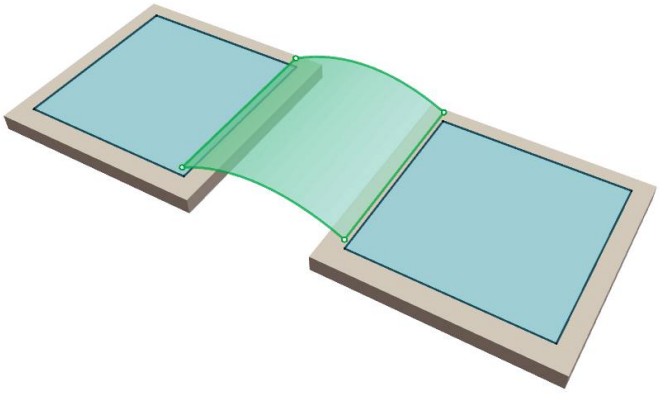


Figure 29: *Schematic of a jump link connecting two edges of different navigation mesh polygons.*

with an endless number of potential jumps in-between them, each having its own trajectory, a jump link does not contain any trajectories. Instead, a jump link connects two edges of a navigation mesh, which means that there are jumps possible between these edges. Figure 29 shows a jump link that connects the navigation mesh polygon on the left side of the figure with the one on the right side.

The jump link can be stored like a shared edge connecting the two navigation mesh polygons, meaning that the jump link is an edge in the navigation graph used for pathfinding. Such jump links can be used by any high level pathfinding algorithm and they allow path smoothing algorithms to determine the actual jump an agent performs along its path. Figure 30 shows an unsmoothed path (purple) as determined by the basic pathfinding, while in figure 30, one can see the same pathfinding query after the path has been smoothed (pink), resulting in the jump being adjusted to the actual path the agent will take.

The adjustment of the trajectory requires the trajectory along which the

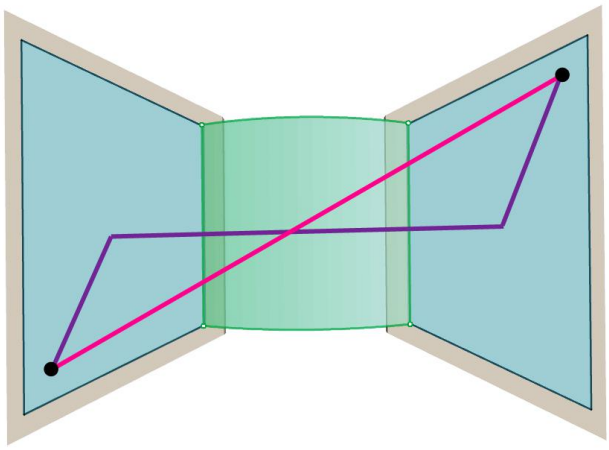


Figure 30: *An unsmoothed path (purple) and a smoothed path (pink), both traveling through a jump link (green area).*

agent actually jumps to be determined during the path smoothing process. That is because the takeoff point on the takeoff edge and the landing point on the landing edge depend on the current pathfinding query. As long as the takeoff point and the landing point are not fixed, the trajectory cannot be determined. Consequently, the trajectory has to be computed during the path smoothing and cannot be preprocessed. The trajectory along which the agent will jump can be determined with the jump trajectory lookup table. Therefore, the lookup table is not only put in action in the jump link generation process, but also during run time to get the actually performed jump trajectory for a pathfinding query. This is possible because the jump trajectory lookup table grants us access to the desired jump trajectory in linear time depending on the number of trajectories connecting the takeoff point with the landing point.

Therefore, the jump link is a data model which by definition has the desired flexibility in jumps. In comparison to storing single jumps in a desired level

of detail, jump links require much less memory. A jump link in conjunction with the jump trajectory lookup table represents a huge set of jumps. This set count of jumps is restricted by the sampling density of the jump trajectory lookup table and the precision of the determination of the takeoff and landing points in the path smoothing process. Since no trajectories are stored, we only have to store the two edges which are connected by the jump link. As will be explained in detail later, a jump link needs to be able to only apply to a part of an edge. Thus both edges will be stored together with two variables between zero and one denoting where on the edge the link begins and ends. A screenshot of such a partial jump link can be seen in figure 31. To summarize, a jump edge is stored in the navigation graph connecting two nodes. This jump edge also includes a pointer to a jump link, which holds the information about which two outline edges of the two connected nodes (navigation mesh polygons) are actually linked and which part of them.

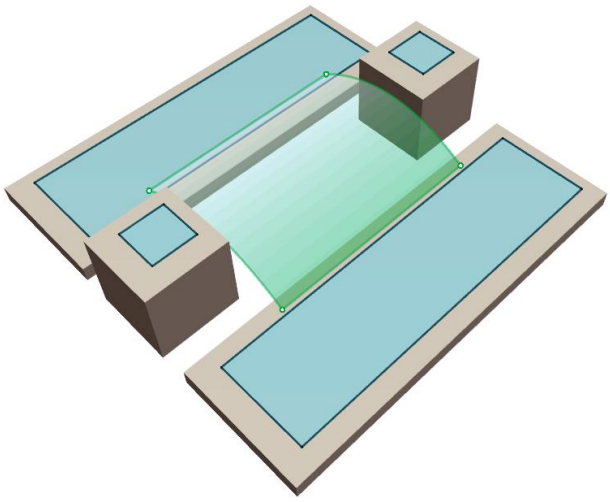


Figure 31: *Obstructions on the left and right side leading to a partial jump link. (Only the discussed jump link is visualized.)*

We defined jump links to connect two edges of the navigation mesh. This works fine for jumps of the class *jump from edge onto edge*, but jumps of the class *jump from edge into polygon* do not land on an edge of the navigation mesh. For jumps of this class a virtual edge is created along the landing points of the downward jump. This virtual edge then serves as a landing edge. An example of a virtual edge is visualized in figure 32. In the situation that a jump down lands in several navigation mesh polygons, there will be several jump link edges created in the navigation graph to account for the different connections to the multiple navigation mesh polygons that the jump down lands in. Still, there is only one jump link with one virtual edge stored, because how the agent jumps through this jump link is up to the path smoothing and does not interfere with high level pathfinding. Apart from the virtual edges there is no difference between the jump links for jumps of the class *jump from edge into polygon* and the jump links for jumps of the class *jump from edge onto edge*.

With this concept of virtual edges we are able to store jump links for jumps of all three different classes. As we have seen in section 4.6, jumps of the class *jump from polygon onto edge* (upward jumps) are only possible where jumps of the class *jump from edge into polygon* (downward jumps) have been found. In case that the complete jump down link can also be jumped up, the link will be stored as a bidirectional jump link instead of two separate directional ones. The *Jump into Polygon Test*, which checks for such downward and upward jumps, will be described elaborately in the next section.

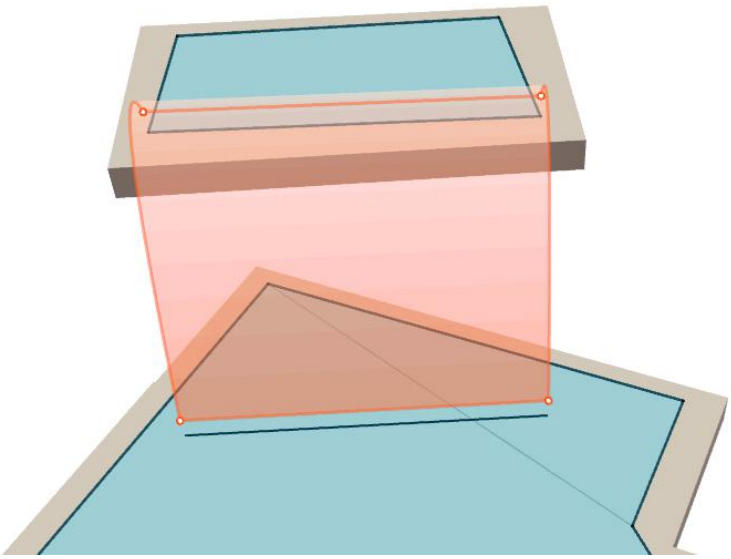


Figure 32: *Screenshot of a virtual edge (blue line at the lower end of the jump link) for a jump from an edge into polygons.*

5 Jump into Polygon Test

5.1 Introduction

The *Jump into Polygon Test* is one of the two jump tests mentioned in the section 4.6. There we have discussed that jump ups are only possible at locations where a jump down is possible. Therefore, the *Jump into Polygon Test* first checks if a jump down from a navigation mesh edge is possible, and only if a jump down has been found, a test for the jump up is done. The jumps found by the *Jump into Polygon Test* are of the class *jump from edge into polygon* and of the class *jump from polygon onto edge*. Intuitively this test finds jumps from a balcony onto the ground, from a fire escape down into the alley and from the balcony onto a windowsill.

The *Jump into Polygon Test* consists mainly of the determination of the landing points and afterwards a collision test of the jump space after which the jump links are created. All these steps will be described in more detail in this section. But first we will take a closer look why the *Jump into Polygon Test* is really necessary.

5.2 Determination of the Landing Points

The Jump into Polygon Test has a given takeoff edge, but it has no fixed landing points. As mentioned earlier in section 4.3 the wanted jump down trajectory is the one which represents a minimal jump. With this trajectory the landing points will be determined. The minimal trajectory will be placed at the takeoff edge and where this minimal jump down trajectory collides with the navigation mesh is where the landing point will be.

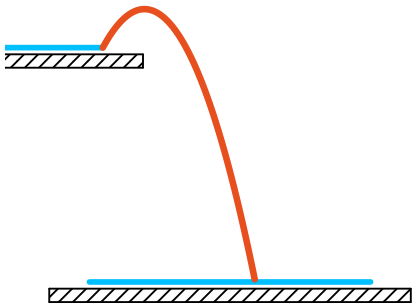


Figure 33: *Side view of a jump down (orange) with geometry (hatched) and navigation mesh (blue).*

The navigation mesh is constructed so that the agent can stand at every point inside of that mesh. This means that the navigation mesh never touches an obstruction. To ensure this, there is a gap between the navigation mesh and any obstruction with the extent of half the agent's width.

As described in section 4.6 we want a jump that takes off at the edge of a navigation mesh and not the ledge of the geometry underneath. For our minimal jump we make use of that gap between the navigation mesh and the ledge of the obstruction and deduce that the minimal jump's width at the height of the takeoff has to measure at least an agent's width so that the jumping agent does not collide with whatever he was standing on. If we consider the jump trajectory as a two-dimensional curve, we let the coordinate origin be the takeoff point. This means that the minimal jump does not drop under a positive jump height before the jump width is at least the agent's width. This condition has to be met so the agent gets over the obstruction edge, which is illustrated in figure 33. The jump that has the shortest trajectory to the floor is the fastest jump down and defines the minimal jump that we are looking for. We measure which trajectory is the

shortest when hitting the maximum falling depth and this trajectory is our minimal jump trajectory. Combining this with the definition of the jump trajectory from section 4.4, the resulting minimal trajectory for our default settings is $trajectory_{2.7\frac{m}{s},61.6°}$. This minimal jump trajectory is visualized in figure 34.

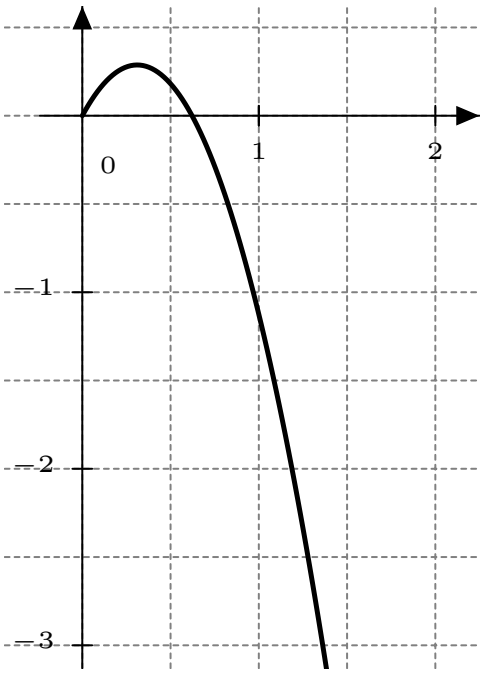


Figure 34: *The minimal jump trajectory with a velocity of* $2.7\frac{m}{s}$ *and an angle of* $61.6°$
$\frac{-g}{2\cdot\left(2.7\frac{m}{s}\right)^2\cdot\cos^2(61.6°)}\cdot x^2 + \tan(61.6°)\cdot x$

Similar to a navigation mesh where a polygon represents an area in which the agent can move freely, the wanted jump links should be represented by a curved area over which the agent is able to jump. This especially means that it is possible for the agent to jump along the outer border of a jump link,

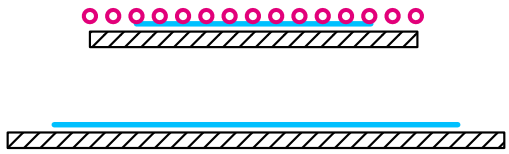


Figure 35: *Schematic front view of the extended sampling at the right and left end of the takeoff edge. The takeoff sample points are the pink circles.*

which implies that space to the left and the right of the jump links also has to be tested for collisions. For this purpose we extend the takeoff line segment at both ending points by half of an agent's width, as visualized in figure 35.

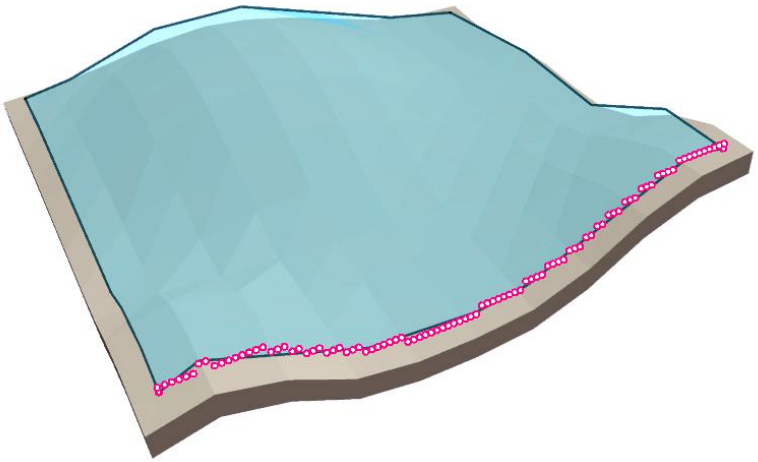


Figure 36: *Perspective view of a jump down sample point distribution (pink circles) for the takeoff edge.*

To determine the landing points a tunable number of sample points will be equally distributed along the extended takeoff line segment. For each

takeoff sample point a landing point will be placed where the minimal jump trajectory first collides with the navigation mesh or the world geometry. If there is no collision, this landing sample point is marked with "No Collision", meaning that it is an invalid landing point. Otherwise the landing point is marked with "Navigation Mesh" or "World Geometry" according to the type of the first collision. In this way it is ensured that each takeoff point with a valid landing point is connected by the minimal jump trajectory, provided that there is no obstruction (see figure 36). The markings and their relevance will be explained later in section 5.4.

5.3 Jump Collision Volume

The jump collision volume is a volume in which all jumps along an edge are performed. Therefore, this volume has to be tested for collisions to find out where jumps are possible and where not. Before we turn to the whole jump collision volume, we will have a look at the volume of one single jump. Every jump is defined by a jump trajectory, so it is obvious that the single jump collision volume is also defined by that same trajectory. As a result, the volume of a single jump looks like a tube, which is illustrated in figure 37.

The jump collision volume along an edge e is given by

$$\text{jump collision volume}(e) = \bigcup_{\text{all jumps } j \text{ along this edge } e} \text{single jump collision volume}\,(j)\ ,$$

where j is one jump from the takeoff edge e. Because we do not only want to have a specific number of jumps per edge but rather want to be able to jump from every point of the edge, we test the jump collision volume for obstruction and not the single jump collision volume.

We represent the volume by a set of jump collision volume slices. Figure 38 shows how these slices are distributed between the takeoff edge and their

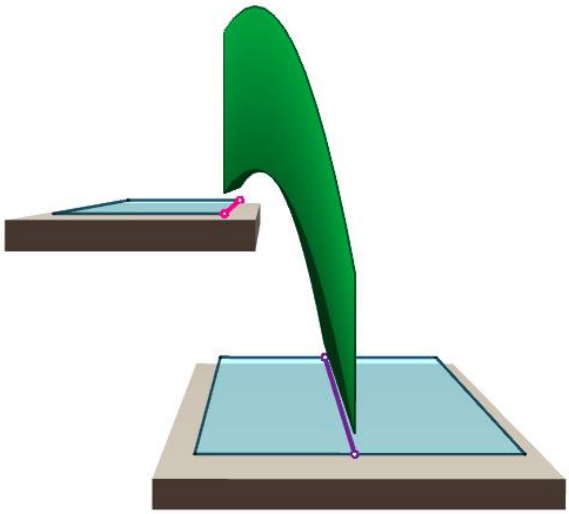


Figure 37: *Profile of a single jump collision volume.*

jump down landing points. Each slice will be individually tested for collisions and have its own collision value. This collision value of one slice can either be free or obstructed. A slice is defined by a takeoff point, a landing point and the set of jump trajectories connecting these two points. The set of jump trajectories is provided by the corresponding cell in the jump trajectory lookup table. Obviously this cell contains the minimal jump trajectory, because the landing points were constructed by this trajectory. The jump trajectory lookup table also provides alternative jump trajectories, which connect the takeoff point with the landing point. The *Jump into Polygon Test* uses only those cells of the lookup table, which include the minimal jump trajectory. The set of trajectories returned by the jump trajectory lookup table is used to construct a slice. A slice is obstructed if all collision tests for all trajectories detect an obstruction, which means there is no possible jump. As stated in section 5.2, the number of takeoff and landing sample points

is tunable, and this directly relates to the density of the slices and thus the precision of the collision test. Meaning if the density of the slices is too low, it is possible that a very small obstruction exists between slices which would then be falsely tested as unobstructed. Therefore, it is important to have a sufficient sampling density. In figure 38, collision free slices are visualized in green.

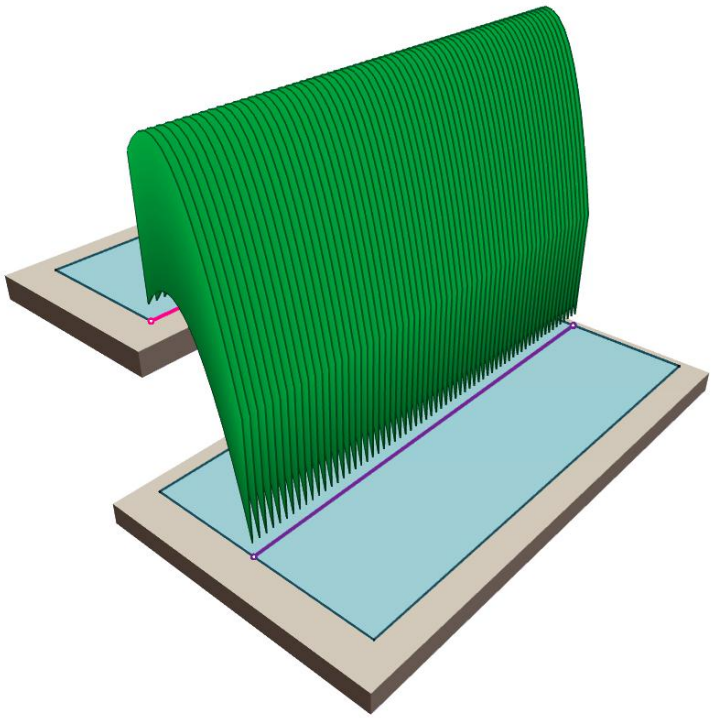


Figure 38: *Perspective view of a jump down with all its slices.*

One collision test is performed with one trajectory collision area. This area is located between one takeoff point and one landing point and defined by a jump trajectory that connects these points. More precisely a trajectory collision area is the two-dimensional area between the jump trajectory and a

vertically shifted version of that. The lower jump trajectory starts where the feet of agent would be when initiating the takeoff. The top jump trajectory starts at the height of the top of the agent's head, which means it is vertically shifted by the agent's height. A trajectory collision area is illustrated in figure 39. To test these jump collision areas of one slice the implementation uses a collision test from [Mon11a]. A practical method to accelerate a collision test is first to check if a bounding box is free of collisions and only test the detail geometry for collisions if the bounding box is obstructed. This concept is also valid for our implementation, which means that the slices only have to be tested if a bounding box of the jump collision volume along an edge is obstructed.

As we explained in section 5.2, all takeoff points are connected to their landing points by the minimal jump trajectory, provided that there is no obstruction. Consequently the collision volume, that is used to test for obstruction, is only composed of slices which are based on the minimal jump trajectory. Thus the slices are an adequate representation of the jump collision volume. In the next section we will discuss how obstructions of the minimal jump trajectory are handled.

5.4 Handling Obstructions

As described in section 5.3, slices are tested for obstruction and a slice is defined by a takeoff point, a landing point and the set of jump trajectories connecting these two points. The result of the collision test is either that the slice is not obstructed or that it is obstructed. In the following figures this will always be represented by the colour of the slices. Green slices are free of collision, while red slices are obstructed (see figure 40).

If a slice is obstructed, it means that all collision tests of all available jump trajectories for this slice were tested positively for obstruction. As described

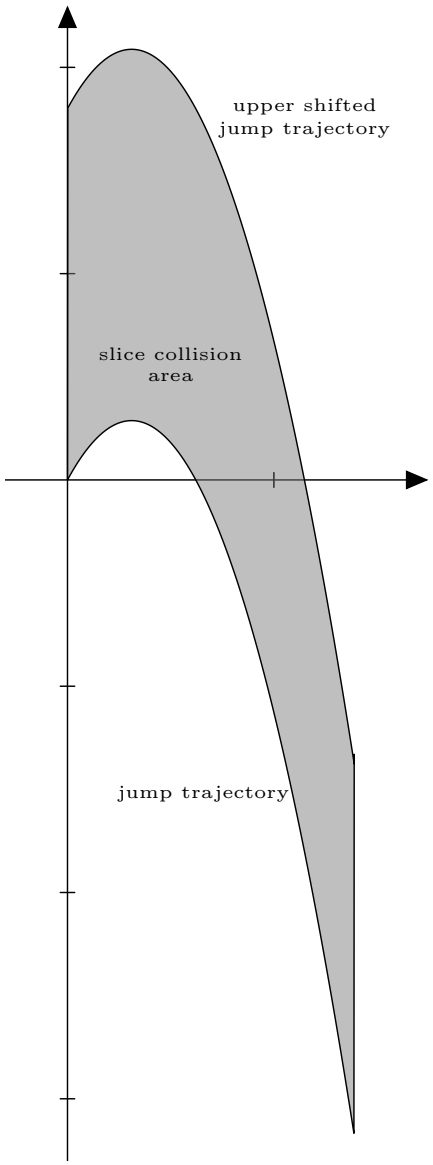


Figure 39: *A slice collision area (gray) defined by a jump trajectory.*

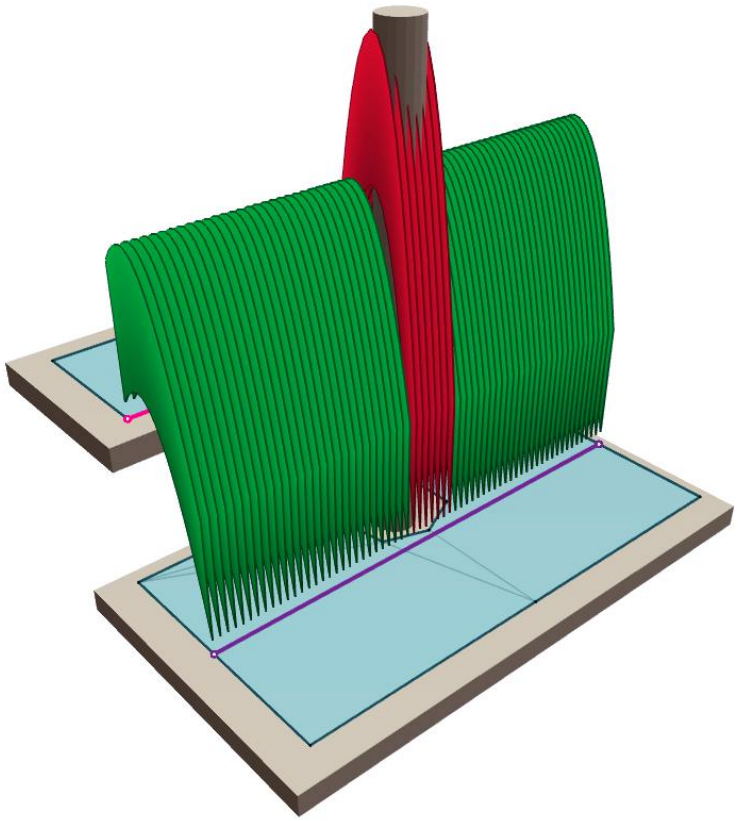


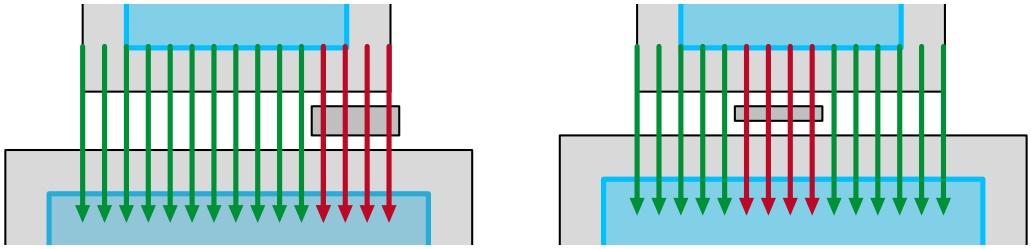
Figure 40: *Perspective view of a jump down with its unobstructed slices (green) and its obstructed slices (red). The last tested jump collision area of every slice is rendered, which means for blocked slices that always the longest jump trajectory is rendered in red.*

in section 4.4, the available trajectories of one slice are provided by the jump trajectory lookup table. If at least one jump trajectory area is tested collision free, the slice is marked green because a valid jump has been found. Because the trajectories in the lookup table are sorted by their arc length, the first obstruction-free jump that is found is always the best to be found.

Following the above, we have to handle two different types of obstructions in the *Jump into Polygon Test*. First, an obstruction of all jump trajectory areas, leaving the slice obstructed. And secondly, an obstruction which is just partial, meaning at least one unobstructed alternative jump trajectory was found. If an alternative jump trajectory is chosen because the minimal jump trajectory does not connect the takeoff point and the landing point free of obstruction, it is possible that the alternative jump trajectory is dissimilar to its neighbours. We will now take a closer look at how to handle these types of obstruction.

First, we will focus on the total obstruction of a slice. In the vast majority of cases, if there is one obstructed slice, then there are also neighboring obstructed slices. Their position at the takeoff edge can be at the end (see figure 41(a)) or in the middle of the edge (see figure 41(b)) and of course there can be multiple obstructions, which will be handled separately. Therefore, we will focus on the handling of one obstruction.

If the obstructed slices are at the end of the edge we obviously want to cut them off and shorten the length of the takeoff edge we are looking at. As a result we will get a jump link that does not cover the complete outline edge of the navigation polygon. In the case that there is an obstruction in the middle of the edge, the agent cannot jump straight down here. Therefore, we want jump links at both ends of the edge but no jump link in the middle, so we cut the area with the obstructed slices out of the takeoff edge. The



(a) *Side obstruction leading to edge shortening.*

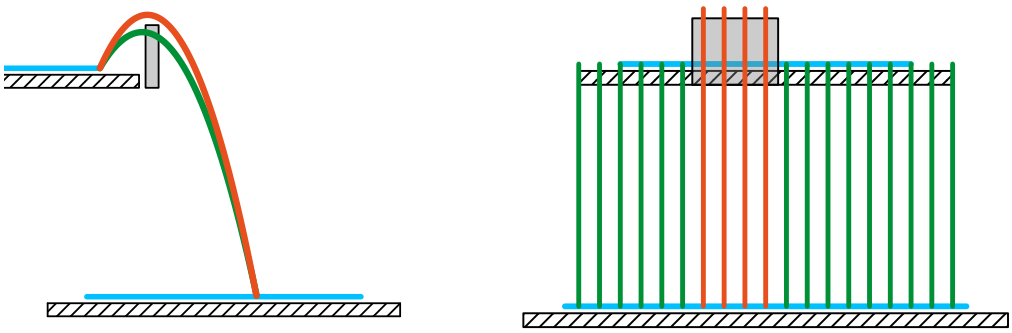
(b) *Center obstruction leading to edge split.*

Figure 41: *Renderings of two different obstruction scenarios.*

two new takeoff edges (one on either side of the cut-out segment) will get their own jump links. The position of the takeoff edges on the outline edge, together with the rest of the jump link information, is stored as described in section 4.7. The original navigation polygon outline edge, on which the takeoff edges lie, is never changed. Basically we do the same no matter at which position of the edge the obstruction lies; we cut off the obstructed area and get several new edge segments from which we construct jump links.

As described above there are two different kinds of obstructions and now we will take a closer look at the second kind. In this situation there exists an obstacle over which we can jump with a higher jump. This is illustrated in figure 42, in which the green coloured slices are based on the minimal jump trajectory and the orange slices on an alternative jump trajectory.

If the slices have significantly different trajectories, the collision free volumes that were found are also significantly different. This leads to the problem that it is not guaranteed that the resulting jump polygon can actually be cross jumped because there are most likely obstructing objects in the dynamically chosen jump path. Accordingly, we want more than one jump link to be created. In order to create a proper jump link for the area with the alternative



(a) *side view* ***(b)*** *front view*

Figure 42: *Two different perspectives of dissimilar trajectories leading to edge split.*

jump trajectories, we have to be sure that there is enough unobstructed space for the jump. As described in section 5.3, the slices are a representation of the jump collision volume which is a union of all single jump collision volumes. Since we want the agent also to be able to jump along the two outer borders of the later created jump link with the alternative jump trajectories, we have to test a jump collision volume for this area. The area has to be extended to make a jump along the outer borders possible and therefore we need to test at least half an agent's width beyond the outer borders of the future jump link. As we do so, we also extend the takeoff and landing segments. The newly created slices from the extension also have to be tested for collision. As a result of this extension by half the agent's width at both sides of the edge, the tested volume will always be as wide as the agent or even wider.

Above we have seen how dissimilar jump trajectories caused by obstructions are handled. But up to now we have not gone into the handling of dissimilar trajectories, which differ in their landing point positions. This will be discussed in the next section.

5.5 Handling Different Landing Points

There are two cases for landing points that need to be handled: On the one hand if no landing point is found and on the other hand the landing points which have a significantly different position than their neighbours. This section will describe how we handle these different landing points. In the *Jump into Polygon Test* this is done after the determination of the landing points and before the slices, which represent the jump collision volume, are tested for collisions.

The case in which no landing point is found means that the minimal jump trajectory does not collide with anything, i.e. there is nothing below this part of the takeoff edge near enough to jump down onto (see figure 43).

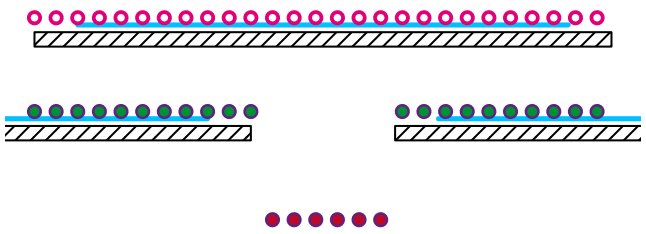


Figure 43: *Sketch of a takeoff edge with takeoff points (pink circles) and their corresponding landing points. The green landing points are marked with "Navigation Mesh" or "World Geometry" and the red landing points are marked with "No Collision".*

In this situation we do not want to have a continuous jump link, so we cut the takeoff points, which have no corresponding landing points, out of the takeoff edge. If a segment is smaller than an agent's width, it will be deleted. With all other segments will be continued separately.

Now we take a closer look at how different positions of landing points are handled, so first of all we have to clarify what different positions of landing points means. Let us take the example of a box standing on the floor under the takeoff edge as illustrated in figure 44. The landing points are created where the minimal jump trajectory hits the first collision. Following this logic, the landing points that are created on the box are significantly higher than the landing points that are created on the floor. This significant height difference is the condition that is meant with different positions of landing points.

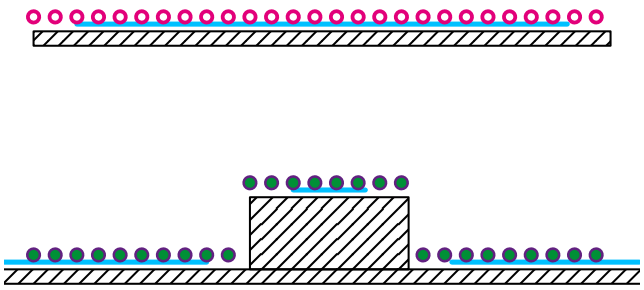


Figure 44: *Significant height differences in the landing points resulting in the construction of three virtual edges.*

Let us take a brief digression about the nature of navigation meshes. A navigation mesh is constructed in a way so that the agent, for whom the mesh is designed, can stay at every point inside of the mesh. In the situation with the box on the floor this means that the navigation mesh has a gap. The size of the gap is based on the width of the agent and as a result it has at least half the agent's width, both at the border of the lower navigation mesh on the ground and at the border of the higher navigation mesh on the box. Together, these two gaps are one gap with the whole width of the agent.

With regard to our topic of the different landing point positions this means that if there is a significant height difference between landing points, there have to be two polygons in the navigation mesh separated by a gap. This gap has to be at least as big as the agent is wide, implying that there have to be some landing points, which were created because the minimal jump trajectory collided with the world geometry.

Because of the gap and the resulting landing points on the world geometry we can split the takeoff edge exactly between those two landing points that differ in their height. We do not have to extend the segments due to the fact that there are landing points outside of the navigation mesh, which cover at least the agent's width. As a result for the jump landing inside the navigation mesh on the outermost point of the mesh towards the gap, it is guaranteed that the jump space has been checked like for all other jumps. With each segment of the takeoff edge that is big enough will be continued separately by creating the slices which connect its takeoff and landing points.

5.6 Jump Link Generation

As we described in the sections above, we have created takeoff points along a segment of the takeoff edge with corresponding landing points connected by similar jump trajectories, which are free of obstruction. Now that we have all of the aforementioned information, we can construct a jump link for this takeoff edge.

The first step in constructing a jump link is to shrink the takeoff edge. As described in section 5.2, we originally extended the takeoff edges, which have not yet been split at that point, in order to determine the landing points. The extension of the takeoff edge was necessary to guarantee that a jump along every takeoff and landing position inside the jump link is possible. Thereafter we created slices along the extended takeoff edge to test them for collisions.

Therefore, we now have to shrink back the segment to ensure that a jump along the outer border of the later generated jump link is unobstructed and thus possible.

In the process of handling landing points with significantly different positions, we split the takeoff edge. As described in section 5.4, there has to exist at least a minimal number of landing points marked with "World Geometry", which add up to a total width of at least the agent's width. For similar reasons as for the previously mentioned extension of the takeoff edge, we want the agent to be capable of jumping along the outer border of the later generated jump link. That also means that the agent has to be able to stand at the landing position and consequently the landing position has to lie inside of the navigation mesh. If there are landing points that are not marked with "Navigation Mesh", the takeoff edge has to be shrunken until it only consists of points whose corresponding landing points are marked with "Navigation Mesh".

Thereafter we can create a virtual landing edge. An edge has to be created because the agent jumps down into a polygon, so there is no edge that the agent could be landing on. But an edge for the landing is necessary to integrate the connectivity logic of the jump link into the navigation mesh data structure. We call this a virtual edge because it is no outline of any navigation mesh polygon. A jump link is the connection between an edge segment of a navigation polygon's outline edge (takeoff edge) and a corresponding virtual landing edge as described in section 4.7. All other information, such as which trajectory connects a takeoff point with a corresponding landing point, the slices and the takeoff and landing sampling points, will not be stored in the jump link. Finally between these two edges a jump link could be constructed, but before this is done, we will present the reverse test to check if an upward jump is performable where the downward jump was found.

5.7 The Upwards Jump Test

After a jump down has been found, a test to find the reverse jump up is applied. The reason for only testing for upward jumps when a downward jump has been found was described in section 4.6. Since we already have the takeoff edge and the virtual landing edge of the jump down including their sampled points, we can reuse these edges. The jump down takeoff edge becomes the landing edge of the jump up and the virtual landing edge of the jump down becomes the takeoff edge of the jump up. Therefore, the takeoff and landing edge for a jump from the inside of a polygon onto the edge of another polygon are given and even the takeoff points along the takeoff edge as well as the landing points along the landing edge have previously been constructed.

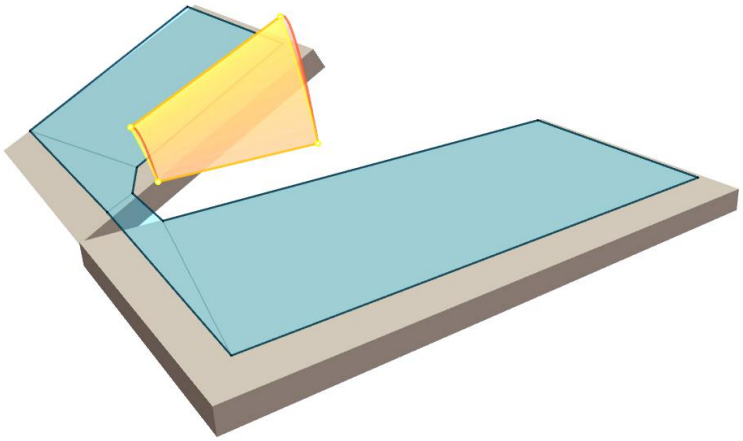


Figure 45: *Jump up and down test resulting in a bidirectional jump link.*

Thereafter, we only have to look up the trajectory connecting one pair

of takeoff and landing points in the jump trajectory lookup table, build the corresponding slice and test it for obstructions. The only difference to the jump down is that the trajectories are fetched from the lookup table and that there is no need to construct a virtual edge. The evaluation of the unobstructed and blocked slices is also similar. If the jump up edges are connected by unobstructed trajectories that are similar to the jump down trajectories, the jump down link can be stored as a bidirectional jump link as shown in figure 45.

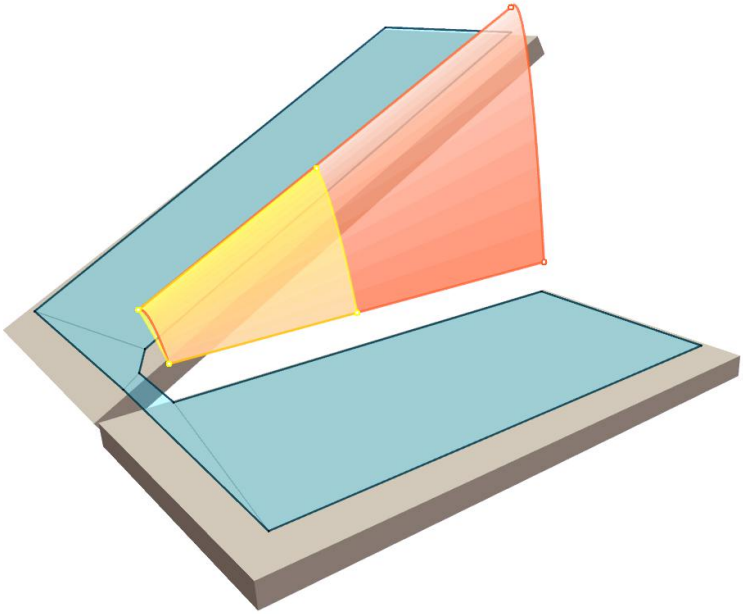


Figure 46: *A jump down link (orange) and a jump up link (yellow). The jump up cannot reach the higher spots of the jump down, so they are both saved as directional links.*

In case some slices of the jump up are obstructed or the trajectories are significantly different from each other, the upward jump's takeoff and landing edge will be split in several parts, each generating a monodirectional

jump link as visualized in figure 46. These jump links can finally be stored as described in section 4.7. Thereby the slices and sampling points along the takeoff edge and the virtual landing edge are discarded and only the information along which segment of the edges jumps can be performed are stored in the jump link.

5.8 Summary

This section gives a summary of the *Jump into Polygon Test* as illustrated in figure 47. A more detailed figure, visualizing all steps of the *Jump into Polygon Test*, can be found in the appendix A. That test generates jumps of the class *jump from edge into polygon* and of the class *jump from polygon onto edge*, which informally would be called jumps down and jumps up.

The *Jump into Polygon Test* has a given takeoff edge to which a jump down link should be generated. First, the corresponding landing points, based on the given takeoff points, are constructed. Afterwards the takeoff edge is split into smaller takeoff edges if takeoff points do not have corresponding landing points or if the landing points have a significantly different position. For each of these takeoff edges, a collision volume, represented by slices, is tested for collision. In the case that the minimal trajectory for a slice is obstructed, alternative trajectories are acquired with the jump trajectory lookup table. Thereafter, obstructions that block all trajectories of a slice or lead to significantly different jump trajectories cause a takeoff edge split. Then for every takeoff edge the corresponding landing edge is created. Now the jump down calculation is completed.

Thereafter, for every jump down that has been found, a jump up is calculated. The downward jump's takeoff edge becomes the landing edge and the virtual landing edge becomes the takeoff edge. Both edges already have

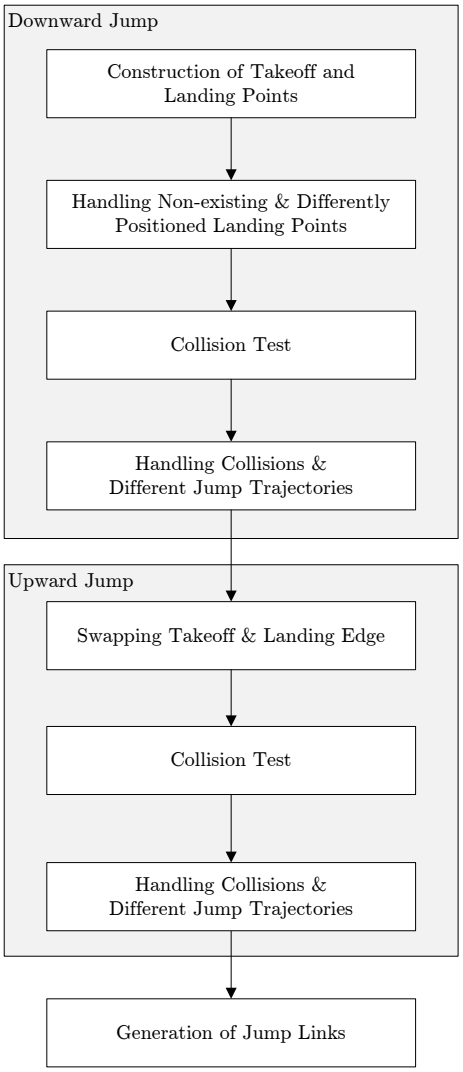


Figure 47: *Overview flowchart of the Jump into Polygon Test.*

takeoff and landing points. The jump down test starts with the slice construction on the basis of dynamically looked up jump trajectories from the jump trajectory lookup table. After that it works the same way as the jump down test.

Finally, jump links can be generated. Based on the results of the jump down, the jump up has been tested and the jump links have been stored in the navigation mesh as monodirectional or bidirectional jump links. Therewith the *Jump into Polygon Test* is concluded.

6 Jump onto Edge

6.1 Introduction

The previous section described the *Jump into Polygon Test*, which finds downward jumps from an edge into a polygon and upward jumps from a polygon onto an edge. The *Jump onto Edge Test*, which we will focus on throughout this section, finds long jumps between two outlines of the navigation mesh. Those jumps are needed for example to jump over a canyon or from one roof of a house to another. Furthermore, we will see how the complete jump trajectory lookup table, which was explained in section 4.4, will be used. In comparison to the *Jump into Polygon Test*, the *Jump onto Edge Test* has a takeoff edge and a landing edge. They are outline edges of the navigation mesh of different polygons. To find most jumps, it is reasonable to test one takeoff edge against all outline edges of other polygons of the navigation mesh which are within range of the maximum jump width.

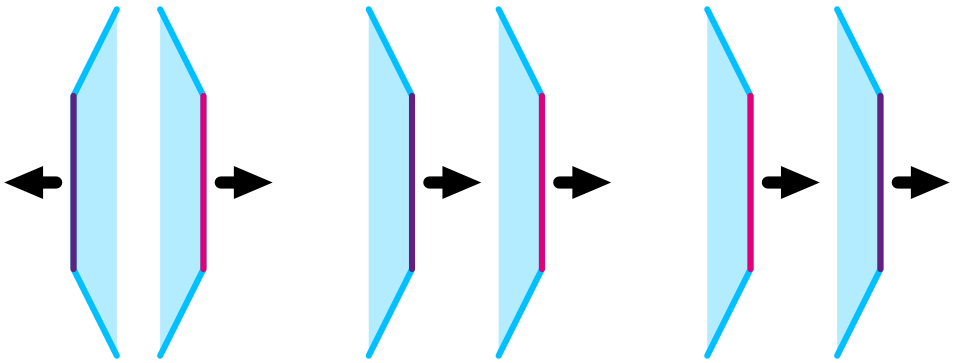
6.2 Preprocessing of the Two Edges

We iterate over all pairs of outline edges of the navigation mesh polygons with the maximum jump range as the first constraint and the ones that need to be tested are passed to the *Jump onto Edge Test* as input parameters. The first step of the *Jump onto Edge Test* is the preprocessing of the edges. The takeoff edge as well as the landing edge are the input for this test, but we cannot expect possible jumps to be there between all pairs of takeoff and landing edges. The preprocessing will quickly reject edges between which no jump can be found or clip the edges as it is useful for the further test.

The *Jump onto Edge Test* finds jumps that connect two different polygons with each other. Of course we are looking for believable jumps, which means

that we want to connect the polygons in such a way that their facing edges are connected by jump links. We will now have a look at how the takeoff edge and the landing edge are positioned to each other and what it means if they are facing each other.

Figure 48 shows some examples of pairs of takeoff and landing edges which are not facing each other. As we can see in figure 48(a) the normals of the edges are pointing away from each other. We do not want to test this constellation for a possible jump, because if there are connections between the polygons, there are better ones covered by the *Jump into Polygon Test* or other edge pairs for the *Jump onto Edge Test*.



(a) *Takeoff and landing edge are facing away from each other.*

(b) *Landing edge behind takeoff edge facing in the same direction.*

(c) *Landing edge in front of the takeoff edge facing in the same direction.*

Figure 48: *Edge pairs of a takeoff edge (pink) and a landing edge (violet) that are not facing each other with their normals shown as arrows.*

Figure 48(b) represents the possible situation of the takeoff polygon lying over the landing polygon, so that the *Jump into Polygon Test* should be applied to find a jump from the takeoff to the landing polygon. The found jump down

would be shorter and thereby more believable than a jump from the takeoff edge to the landing edge. Considering that the landing edge is an outer edge of the navigation mesh, it is most probable that after an edge onto edge jump the agent would even walk back to the left, or in other words he would walk back into the direction where the jump started.

The same placing of the polygons but with switched takeoff and landing edge is shown in 48(c). In this situation we are testing for a jump from the lower polygon with the takeoff edge up to the polygon with the landing edge. A jump from the takeoff edge over the polygon with the takeoff edge to the landing edge is not an upward jump like we would expect it. What we are looking for is the reverse jump of the downwards jump as upwards jump.

We want to find jump links between edges that are facing each other. This means that all landing edges which are lying entirely on the same side of the takeoff edge as the takeoff polygon will be rejected. That is because in order to jump from the takeoff edge onto such a landing edge, the agent would have to jump down through the takeoff polygon, which obviously is impossible, or jump up over the takeoff polygon. Correspondingly, all takeoff edges which are lying entirely on the same side of the landing edge as the landing polygon will also be rejected.

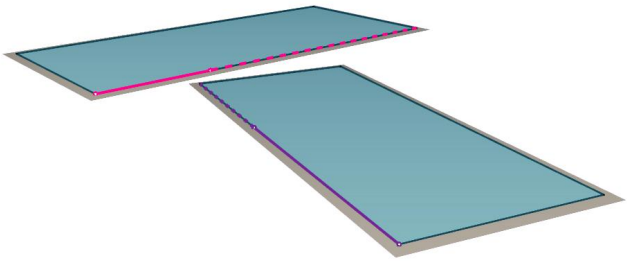
Once we know that the takeoff edge and the landing edge are facing each other and therefore have not been rejected, we can extend the edges. The navigation mesh is constructed in such a way that the agent can stand at every point inside of the mesh. Similar to this fact we want every jump inside of a jump link to be possible, so especially the jumps along the outer borders of the jump link must also be possible. Therefore, we have to test the space beside the actual jump link for collision, which is achieved by

extending both edges at their two terminal points by half an agent's width. After this blind extension of the edges, we clip the edges depending on their positioning to each other.

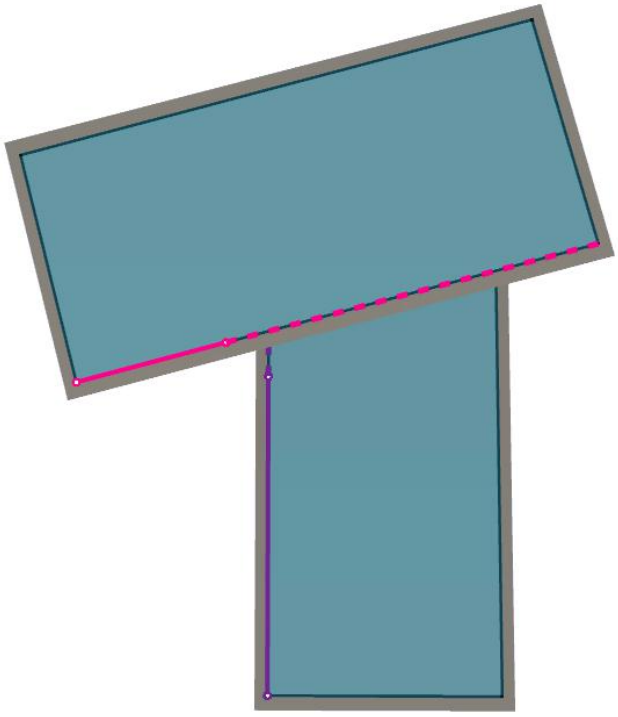
Let us take a closer look at a situation in which the projection of takeoff and landing edge into the horizontal plane, which contains the origin, intersect each other. In figure 49 an example of this situation with a pink takeoff edge and a violet landing edge is visualized. The takeoff edge does not completely lie on the same side of the landing edge as the landing polygon. The same holds for the landing edge, meaning that these edges are partially facing each other. In this example we want to get two different jumps: At the right end of the takeoff edge we expect a jump down into the landing polygon, while on the left side of the takeoff edge we want to find a jump towards that part of the landing edge which is not lying beneath the takeoff polygon.

In this example one can see clearly that the right side of the takeoff edge lies on the same side of the landing edge as the landing polygon, namely over the landing polygon, and will therefore not be connected by a jump found via the *Jump onto Edge Test*. The same applies to the part of the landing edge beneath the takeoff polygon, because it lies on the same side of the takeoff edge as the takeoff polygon. Providing that we do not need these parts (dashed line segments in figure 49) for further computations, we clip them off the takeoff and landing edge.

After these described parts of the preprocessing we know that the takeoff and landing edge are facing each other and that the edges have been clipped to the facing parts. But there is another clipping which is part of the preprocessing that we call clipping by the mapping angle. As the name indicates it has something to do with the mapping, which will be described in section 6.3.



(a) *perspective view*



(b) *top view*

Figure 49: *Two quadrangles partially overlapping each other with the facing parts as solid lines, which are the takeoff edge (pink) and the landing edge (violet). The clipped off parts are dashed.*

Here we just take a look at this sort of clipping and later we will come back to it and describe the reasons for the clipping by the mapping angle.

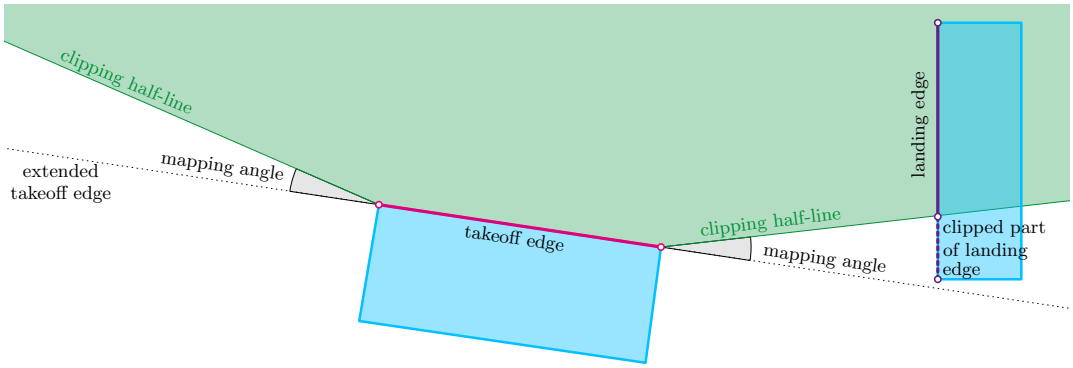


Figure 50: *Sketch of the clipping by the mapping angle.*

In figure 50 we can see a takeoff edge and two clipping half-lines each starting at an end point of this edge. The angle between the extension of the edge and one of the clipping half-lines is the mapping angle. The size of this angle is defined by the user and we usually use a mapping angle of 15°. The landing edge is clipped at its intersections with the two half-lines. As a result, only the part of the landing edge that lies inside the green-shaded area in figure 50 will be used for the further computations. Thus, the bigger we define the mapping angle, the more will get clipped off the other edge. The clipping by the mapping angle is done for both edges as described above for the landing edge as well as for the takeoff edge in the same manner. This is the last step of the preprocessing and the next step in the *Jump onto Edge Test* is the mapping, which we will take a closer look at in the next section.

6.3 Mapping

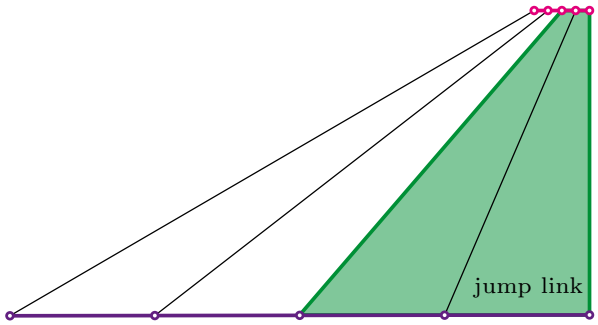
6.3.1 Mapping of One Edge onto the Other

We have seen in the *Jump into Polygon Test* that by determining the landing points, we have not just created landing points, but we have also mapped the takeoff edge onto a virtual landing edge defined by the landing points. In the *Jump onto Edge Test* we have two given edges, so the mapping has to be done in another way. Through this mapping we will also determine the takeoff points and the landing points. The given edges can vary enormously in length and are not parallel in most of the cases, which leads to a totally different mapping situation in comparison to the *Jump into Polygon Test*.

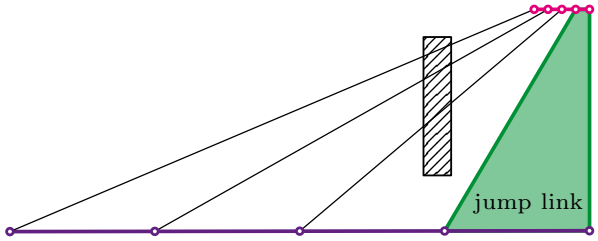
Let us look at the properties of the mapping connections of two different kinds of mapping. These mapping connections, that are line segments which each connect one point of the takeoff edge with one point of the landing edge, will be used to create the collision volume. We want them to be equally distributed in the gap between takeoff and landing edge, so they must not cross each other. Therefore, these connections can be parallel or non-parallel, as long as they do not cross each other in-between the takeoff and landing edge.

As we can see in figure 51, the non-parallel mapping connects the corresponding points of the takeoff and landing edge, meaning that it connects the left end points of both edges, the centre points of both edges and so on. One problem about this kind of mapping is that the mapping connections do not reflect the distance between the edges very well, which is visualized in figure 51(a). Therefore, just a small part of the upper edge would be connected by a jump link, because the rest of the connections are longer than the maximal jump width. An additional drawback lies in the fact that even

the part of the upper edge where the connection distance is small enough could be smaller than the agent is wide, so that no jump link at all would be found for this edge. With this non-parallel mapping it would thus be possible that in a situation like the one in figure 51(a), no jump link would be created although several jumps are possible for the agent. In figure 51(b) we see a situation, in which an obstruction blocks a lot of the connections between the two edges. Thereof only a small part of the upper edge can be freely connected.



(a) *Mapping error caused by mapping connections exceeding the maximum jump length.*



(b) *Mapping error caused by an obstruction.*

Figure 51: *Problems that occur when using non parallel mapping.*

Because of all these drawbacks of the non-parallel mapping we now take a closer look at the parallel mapping. Parallel mapping simply by its definition does not have the problem of finding connections of different length between two parallel edges. Still, there are other drawbacks we have to discuss.

If the mapping connections are parallel, we first think of connections that are perpendicular to one of the two edges. But this obviously bears problems when the two edges have a different length. Then it would only be possible to map the area straight in front of the smaller edge. Furthermore, there would be no mapping connections found at all if the second edge is not positioned straight in front of the edge from which we start the perpendicular mapping. So what we need is a mapping with parallel connections that are oriented towards the edges. The mapping that we use starts with the connection of either the left or the right end points of both edges, as shown in figure 52, and all following connections are parallel to this first constructed connection. We call this type of mapping the end point mapping.

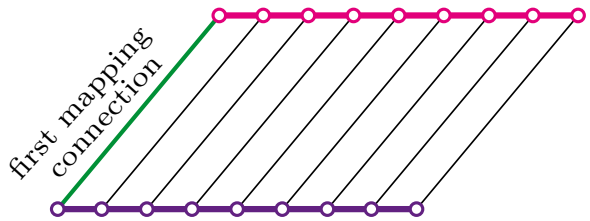


Figure 52: *Parallel mapping starting at the left end points of the edges.*

Until now we have not yet determined which edge shall be mapped onto the other edge. The next section will describe the two cases of the takeoff edge being the domain and the landing edge the codomain for the mapping and the other way round.

6.3.2 Domain and Codomain for the Mapping

As explained before, we want to have a mapping from one edge to the other in order to get parallel slices. First though, we have to decide whether the takeoff edge shall be mapped onto the landing edge or the other way around. In addition this section will discuss what happens if the edges are varying enormously in length.

In figure 53 we can see the result of mapping the larger edge onto the smaller edge. In fact, we would receive several mapping connections that do not hit the smaller edge, which would result in the red marked part of the larger edge being not included in the jump link. On the other hand, we have the advantage that the smaller edge is completely covered with connections.

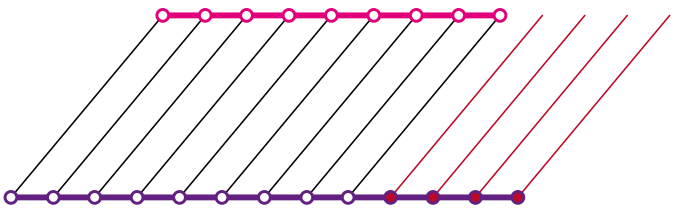
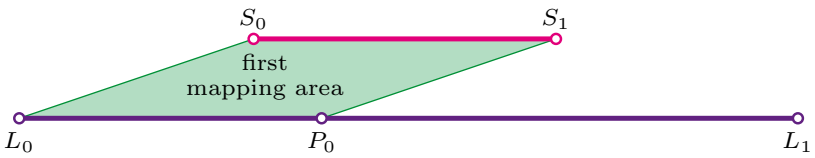
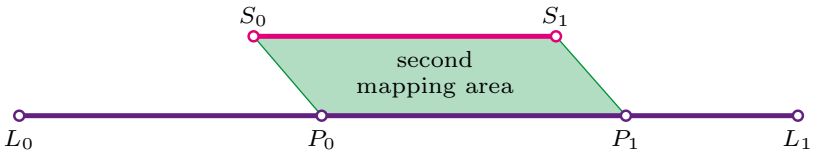


Figure 53: *Parallel mapping of the larger edge (violet) onto the smaller edge (pink).*

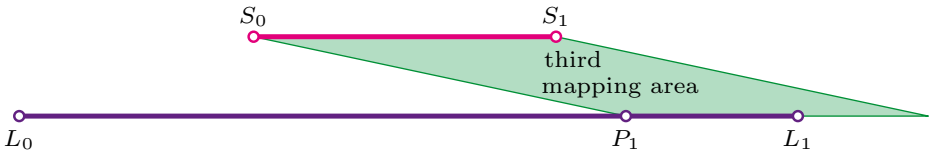
Now we examine how it looks the other way around. If we mapped the smaller edge onto the larger edge, a part of the larger edge will obviously not be covered by connections. To avoid this uncovered part we can simply do several mappings after each other. This multiple mapping of the smaller edge onto the larger edge is illustrated in figure 54. To construct the first connection of the second mapping we use point s_0 of the smaller edge and the point p_0, at which the first mapping ended, on the larger edge. The further mappings are done using the same pattern.



(a) *first mapping*



(b) *second mapping*



(c) *third mapping*

Figure 54: *Mapping of the smaller edge (pink) onto the larger edge (violet).*

With this kind of multiple mapping we still have an area in which the connections do not hit the other edge (in figure 54 the right part of the third mapping). Still, it has a big advantage over the mapping of the larger edge onto the smaller edge: Assuming that the smaller edge is the takeoff edge, the agent can jump onto the left sideas well as the right side of the landing edge. This would not be possible with a single mapping of the larger edge onto the smaller edge.

As we have seen above, the smaller mapping edge is totally covered with connections by the first mapping while the larger mapping edge is only partly covered. This leads us to the criterion for the determination of the smaller mapping edge. Let an end point mapping start on one side of the edge, then this edge is the smaller edge if the line, that is parallel to the first constructed connection and passes through the other end point of this edge, intersects the other or in this case the larger mapping edge. If there is no intersection it means that the edge we started off with is the larger mapping edge.

From this distinction of the edges we derive the following test for differentiation. From now on we denote the larger mapping edge as the edge whose end point (S_1 and L_1) has the longest distance to the line defined by the first mapping connection ($\overline{S_0L_0}$). This edge is called codomain. The other edge, which is the domain, is called the smaller mapping edge. Consequently it is possible that the edge with the smaller length is the larger mapping edge, as we can see in figure 55. The point S_1 from the smaller mapping edge is closer to the first mapping connection than the point L_1 from the larger mapping edge. Therefore, the pink edge is the smaller mapping edge with a length of $4.11cm$, whereas the violet edge is the larger mapping edge with a length of $3.39cm$.

Now that we know how the mapping of the smaller mapping edge onto the other edge works, let us revisit the issue of the clipping by the mapping

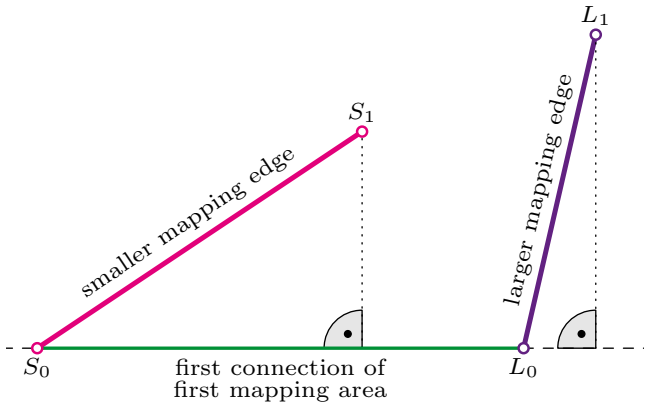
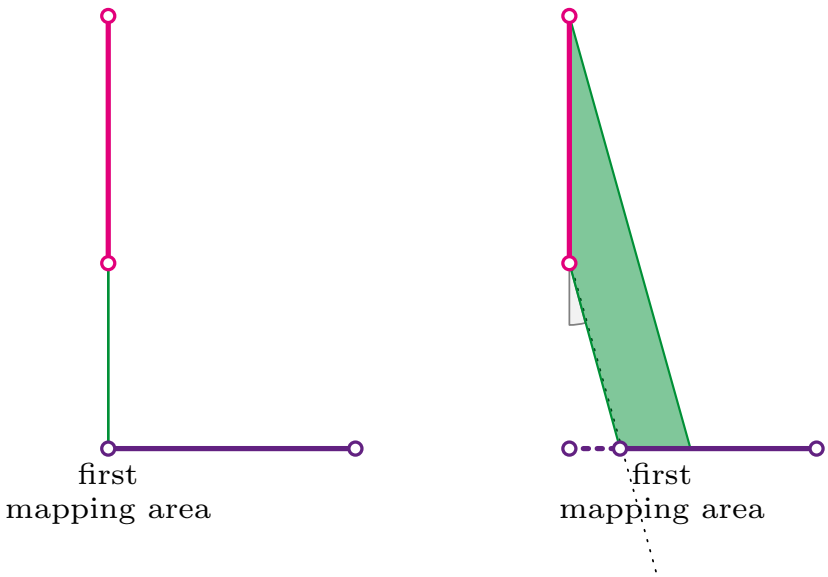


Figure 55: *Determination of the smaller and the larger mapping edge.*

angle, which we have already described in section 6.2. Through the end point mapping we can be sure that the first connections after the first defining connection hit the other edge. But for example if the edges are perpendicular to each other, like in the situation in figure 56(a), the first defining connection would simply be an extension of the vertical edge onto the left endpoint of the horizontal edge. The first mapping would then map the complete upper edge onto this point of the horizontal edge and even the second mapping would map everything onto this one point. To avoid this situation and to avoid mapping one edge onto a very small part of the other edge, we clip the edges by the mapping angle as shown in figure 56(b).

6.3.3 Two-Sided Mapping

In the previous section we have seen that the end point mapping of the smaller mapping edge onto the larger mapping edge does not completely cover the larger mapping edge. As we have mentioned before, to completely cover the larger mapping edge we need to do multiple mappings. In this section we will discuss how the multiple mappings can be positioned so that



(a) *unclipped results in false mapping area*

(b) *clipped results in valid mapping area*

Figure 56: *Sketch illustrating the necessity of the clipping by the mapping angle.*

the larger mapping edge is completely covered by connections.

Figure 57 presents a mapping which has started at one side and is continuously extended along the larger mapping edge. In this example the first mapping completely hits the larger mapping edge, but of the second mapping just the green connections hit the larger mapping edge while the red connections do not. Obviously there are some connections that we do not have to compute. Let us have a look at what this kind of one-sided mapping would mean for the jump links.

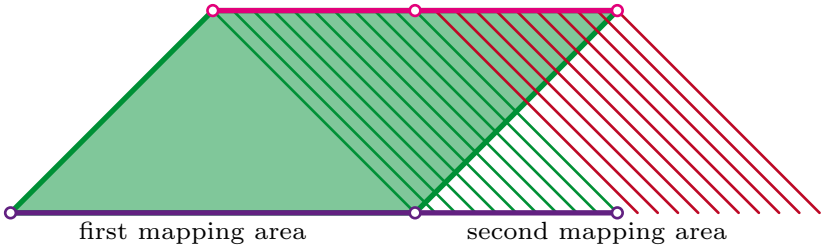


Figure 57: *Mapping from the smaller edge (pink) onto the larger edge (violet) with an overhanging second mapping area.*

For this example we want one jump link to be generated that covers the complete upper and lower edge and that works both ways, meaning that it does not matter which edge is the takeoff edge and which one is the landing edge. The section 6.4 will describe exactly how we test the jump volume between two edges for collisions, but it will be briefly described here for a better understanding: We will construct slices, similar to the *Jump into Polygon Test*, along the connections of the mapping and test them for collision. As a last step we will merge jump collision volumes which correspond to the areas covered by the first, second, third etc. mapping, which will be described in more detail in section 6.5.

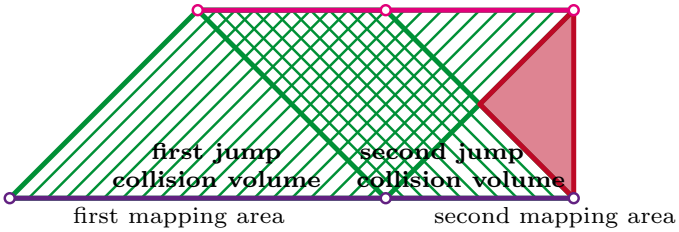
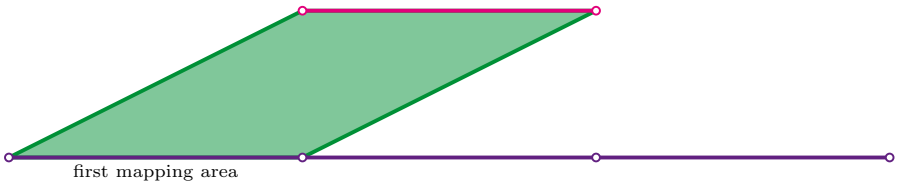


Figure 58: *Issue with merging jump collision volumes of one-sided mapping.*

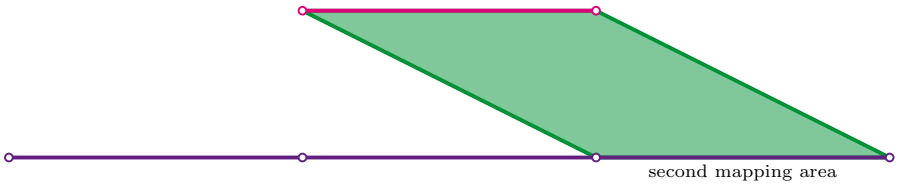
For our current example this means that the first mapping will be completely converted to a jump collision volume whereas for the second mapping, that only partly hits the larger mapping edge, we will only convert the area with the connections that do hit the larger mapping edge into a jump collision volume. If we simply united these two jump collision volumes to get one jump link, we would include the area that is marked with the red triangle in figure 58. The problem about the area is that it has not been tested for collision and we clearly do not want a jump link with an untested area.

A straightforward solution to this problem is the two-sided mapping. In comparison to the one-sided mapping, that starts with the end points on one side of the edges and then continuously maps from this direction, the two-sided mapping starts at both sides of the edges. This is visualized in figure 59. The two-sided mapping basically consists of two one-sided mappings, which start at the opposite end points of the edges. The two one-sided mappings will be calculated alternating, so they both progress towards the middle of the larger mapping edge until they meet or overlap. This way we do not have connections that do not hit the larger mapping edge, but we can have overlapping mappings in the middle of the larger mapping edge.

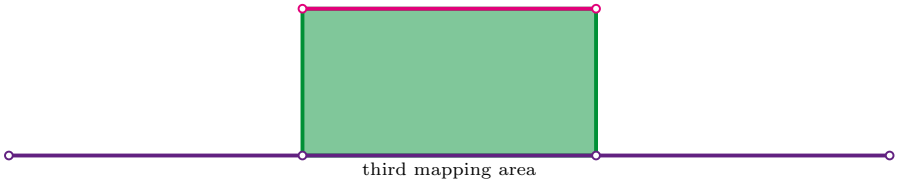
The mapping areas of the two-sided mapping represent the jump collision volumes, as shown in figure 60, which connect the complete smaller mapping



(a) *first mapping area*



(b) *second mapping area*



(c) *third mapping area*

Figure 59: *Two-sided mapping and its resulting mapping areas.*

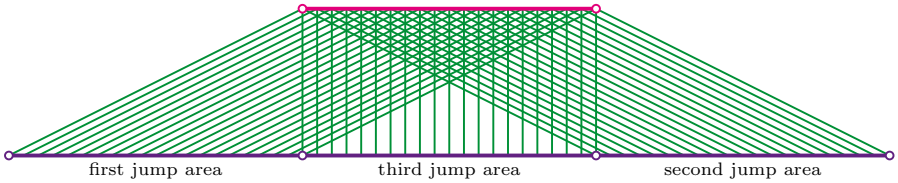


Figure 60: *Combined resulting jump collision volumes correlating to figure 59.*

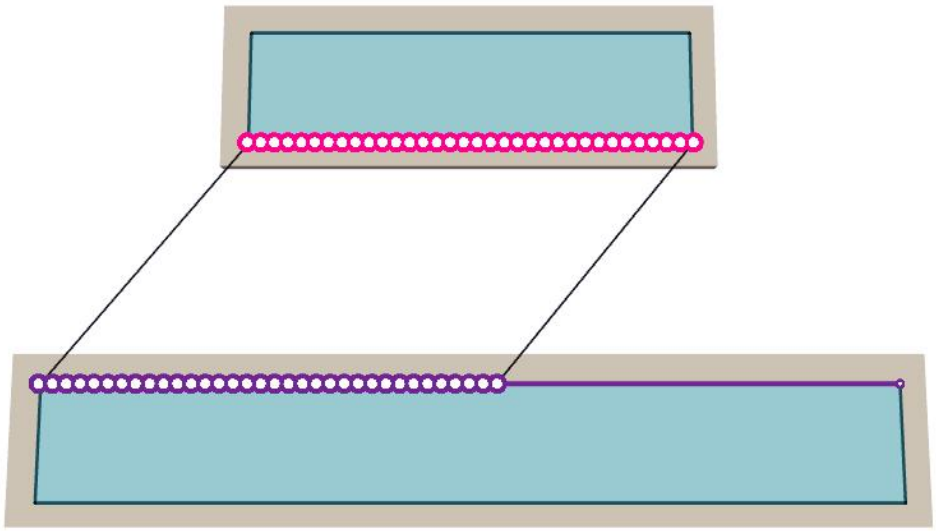
edge to parts of the larger mapping edge. In the next section we will see how these jump collision volumes are tested for collisions.

6.4 Jump Collision Volume Test

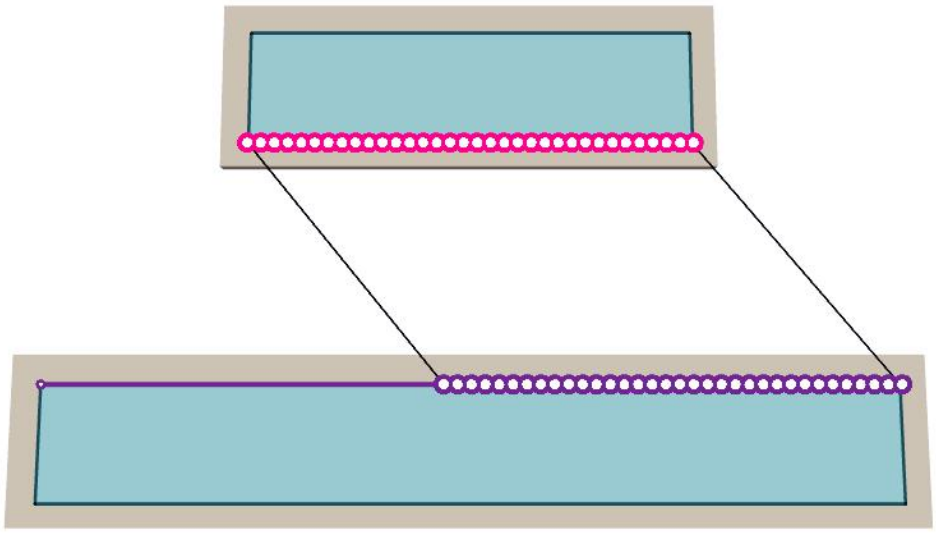
In this section we will see a lot of similarities to the *Jump into Polygon Test*, such as the representation of the jump collision volume by slices and the construction of the slices with the jump trajectory lookup table. These similarities occur because just as in the *Jump into Polygon Test*, we want to find possible jumps and test their volume for collisions and these are tools we use to accomplish this. The following steps are done for each mapping area individually.

The first step towards the slice representation of the jump collision volume is the sampling of the mapping area, which is visualized in figure 61. This means that we sample the smaller mapping edge and the part of the larger mapping edge that belongs to the current mapping area. The amount of sample points is determined by the width of the mapping area. Then the takeoff sample points as well as the landing sample points are equally distributed along the edges of this mapping area. A takeoff point and the landing point vis-a-vis to this takeoff point form a pair which will be used to construct a slice.

To construct a slice, we need not only the pair of sample points but also a jump trajectory connecting these points. As in the *Jump into Polygon Test* we use the jump trajectory lookup table to find a trajectory according to jump width and jump height defined by this pair of sample points. In the *Jump into Polygon Test* all sample points were sampled along the jump down trajectory, so the jump trajectory table was only used to find alternative trajectories. Now the lookup table is actually used to its full extent by providing connecting trajectories for all possible sets of takeoff and landing



(a) *first mapping area*



(b) *second mapping area*

Figure 61: *Top view renderings of two mapping areas and the sample points of the corresponding jump collision volumes.*

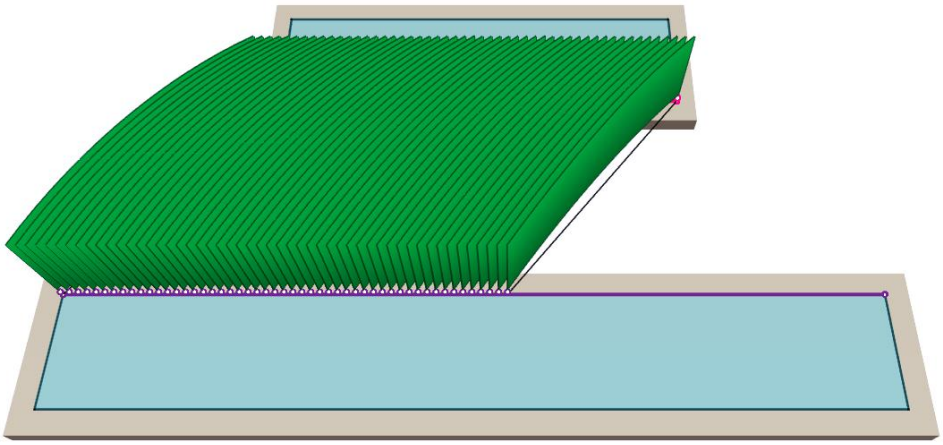
points as well as alternative trajectories. The slice is constructed exactly the same way as in the *Jump into Polygon Test*. The surface of a slice is the area between the jump trajectory and an upwards-shifted copy of the jump trajectory. The shifted trajectory is translated by the height of the agent. The next step is to test this slice for collision, and if the slice is not free of obstruction, another slice with an alternative trajectory from the jump trajectory lookup table is created and tested. A slice is considered as blocked if there does not exist a jump trajectory in the lookup table that leads to a collision-free slice. In figure 62 a jump collision volume with its slices is visualized.

After all slices of one jump collision volume are tested for collision, the next jump collision volume is tested and so forth, until all jump collision volumes between the takeoff edge and the landing edge are tested for collision. The next step in the *Jump onto Edge Test* is the postprocessing, which will be described in the following section.

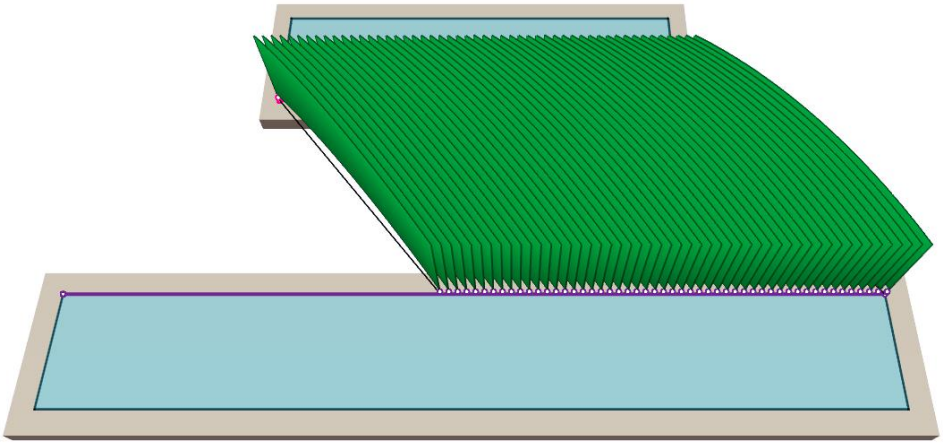
6.5 Postprocessing of the Jump Collision Volumes

6.5.1 Merging of Jump Collision Volumes

In the previous sections we have seen the transition from a takeoff edge and a landing edge over the mapping areas to the jump collision volumes, which are represented by tested slices. This section will finally describe the generation of the jump links based on the jump collision volumes. A straightforward way of generating jump links is to convert every jump collision volume into one jump link. The jump collision volumes are based on the mapping areas which are constructed during the mapping process. Since we chose multiple mappings for this process, the resulting number of mapping areas leads to that same number of jump collision volumes. The



(a) *first jump collision volume*



(b) *second jump collision volume*

Figure 62: *Perspective view of two jump collision volumes with their slices.*

multiple mapping areas between a takeoff and a landing edge are visualized in figure 59, and figure 62 shows all jump collision volumes represented by their slices between that takeoff and landing edge. By this straightforward generation of jump links we would get a lot of jump links between one pair of takeoff and landing edge. Since we obviously want as few unnecessary jump links as possible between two edges, it would be optimal to get only one jump link which completely covers both edges, provided that there are no obstructions. Let us look more closely at the merging of jump collision volumes into a new fused jump collision volume.

In this section we only consider edges with jump collision volumes that have no blocked slices and no slices that are significantly different from each other. In figure 62 we can see two adjacent jump collision volumes. These two volumes are designed to represent the volume in which all jumps between their parts of the takeoff edge and landing edge are performed by the agent. As we can see in figure 63, the two jump collision volumes do not just overlap, but rather the union of them describes a continuous volume. This merged jump collision volume is an adequate representation for all the jumps between united parts of the takeoff edge and the landing edge.

The merged jump collision volume consists of the union of the source jump volumes as well as of the union of the source edge parts. Because the merged volume has already been tested for collision, there is no need to change the representation of the volume from a set of sets of parallel slices to something else. The conditions for the merging are that the jump collision volumes are free of obstruction and that they are adjacent so that both the resulting merged volume and the united part of the larger edge are continuous. If these conditions are fulfilled, two or more jump collision volumes can be merged to a new jump collision volume.

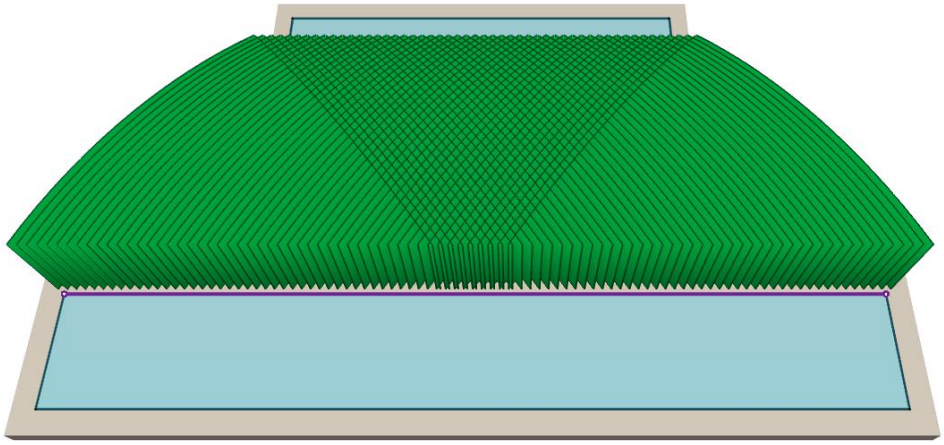


Figure 63: *Perspective rendering of a merged jump collision volume which is based on two unobstructed jump collision volumes.*

After the merging of the jump collision volumes there is one last step before the jump links are generated. We have to shrink both edge segments of the jump collision volume at both end points by half an agent's width to guarantee that all jumps inside this shrunken volume are possible. The shrinking and the generation of the jump links is done both for merged and non-merged jump collision volumes.

6.5.2 Handling Obstructed Slices

In the last section we have seen the merging of jump collision volumes that are free of obstruction. In this section we will explain why jump collision volumes which contain obstructed slices are not merged and how they are processed. First we will discuss why obstructed jump collision volumes can not be merged with unobstructed jump collision volumes.

Figure 64 shows an example of two adjacent jump collision volumes. The

left volume is unobstructed whereas the right volume contains blocked slices. In this example we consider only jump collision volumes from one side of the mapping. The second jump collision volume with the blocked slices could be split into two segments: One containing all the unobstructed slices to the left and the other containing all obstructed slices and the unobstructed slices to the right. The straightforward idea is to merge the left part of the second jump collision volume with the first jump collision volume, which is totally unobstructed. If we merge these volumes, we would get a new volume that includes the complete smaller mapping edge, which is visualized as the upper edge (pink) in figure 64. The orange marked segment of the larger edge is merged to become the second edge of the new volume. The generated jump link would contain the area marked by the yellow triangle in figure 64. This is the area the obstruction is located in. The slices of the merged jump collision volume do not cover the red area, so this area is not part of the two volumes which have been merged. Thus, the merged volume is not a union of the two free jump collision volumes. This example shows that a general union of unobstructed jump collision volumes with the unobstructed parts of a jump collision volume that contains blocked slices can lead to partially blocked jump links. Therefore, we convert obstructed jump collision volumes separately to jump links and we do not merge them.

The handling of jump collision volumes with blocked slices is based on the same steps as the handling of obstructions in the *Jump into Polygon Test*. The first step is the splitting of the jump collision volume so that all obstructed slices are cut out. All resulting new jump collision volumes are shrunken by half an agent's width at each end point. This is done in order to guarantee that all jumps inside the jump link and especially the jumps along the outer borders of the jump link are possible.

These jump collision volumes are further tested for significantly different

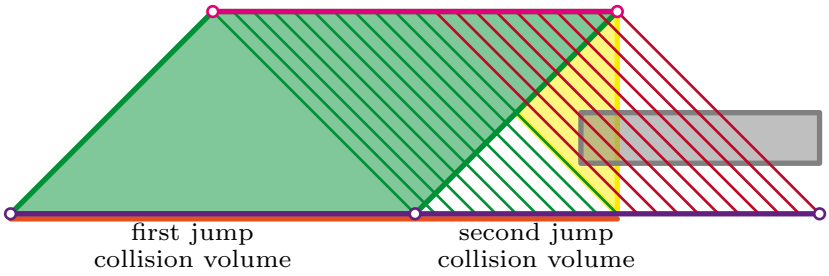


Figure 64: *Example why a partially obstructed jump collision volume can not be merged with other volumes.*

jump trajectories and if necessary processed as it will be described in the next section. Afterwards, these jump collision volumes will generate jump links but they will never be merged with any other jump collision volumes.

6.5.3 Handling Significantly Different Jump Trajectories

Obstructions in the area of a jump collision volume can not only result in blocked slices but also in the construction of a number of slices with alternative jump trajectories. These slices can have a significantly different jump trajectory in comparison to their neighbours next to the obstruction. In that case the jump collision volume is not steady. As outlined in the *Jump into Polygon Test* these slices with the significantly different jump trajectories have to be separated from the rest of the steady jump collision volume.

Accordingly, we split the jump collision volume so that the resulting parts all represent a steady volume. For these parts the merging problem of the blocked slices takes effect, thus prohibiting us to merge the resulting jump collision volumes. Therefore, we process all new jump collision volumes of the splitting separately. This processing is done the same way as in the *Jump*

into Polygon Test and is concluded by creating a finalized jump link for each jump collision volume.

6.6 Summary

The *Jump onto Edge Test* is one of the two tests to generate jump links. It does so between pairs of edges, and the created jump links visually remind of long jumps. The steps of the test, all described throughout this chapter, can be found in a detailed diagram in the appendix A. This section and the figure 65 give a brief summary of the *Jump onto Edge Test*.

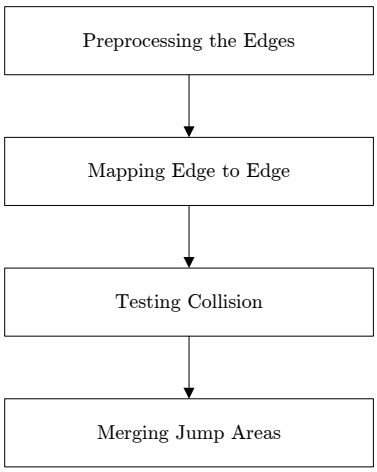


Figure 65: *Overview of the Jump onto Edge Test.*

The first step is the preprocessing of the given takeoff edge and the given landing edge. These edges are rejected if they are not facing each other. The edges are mapped onto each other starting from both sides of the edges towards the centre of the larger mapping edge. With the sample points from the mapping areas slices are constructed based on trajectories from the jump

trajectory lookup table. These slices are tested for collision and thus they represent the jump collision volume. The jump collision volumes are used to create jump links. Depending on the volume's obstruction, the jump collision volumes are handled separately or merged with adjacent volumes. Each jump collision volume will generate a jump link that stores the information from which part of the takeoff edge to which part of the landing edge jumps are possible.

7 Results

7.1 A Robust Solution for an Optimized Search Space

The *Jump into Polygon Test* and *Jump onto Edge Test* form a solution that works robustly and well within the optimized search space which was defined in section 4. Before we discuss the impact of the jump links on the navigation mesh, let us have a look at the different test environments that have been used to evaluate the implemented prototype. We generated jump annotations for seven different maps of the video game *Counter-Strike: Source* [Val13]. In the appendix B on page 140, textured images of these tested maps can be found. The maps "cs_desperados", "cs_east_borough", "cs_parkhouse" and "de_alexandra2" are mostly outdoor maps, whereas "cs_abbey", "cs_ office_-unlimited" and "de_corse" mainly feature narrow urban surroundings.

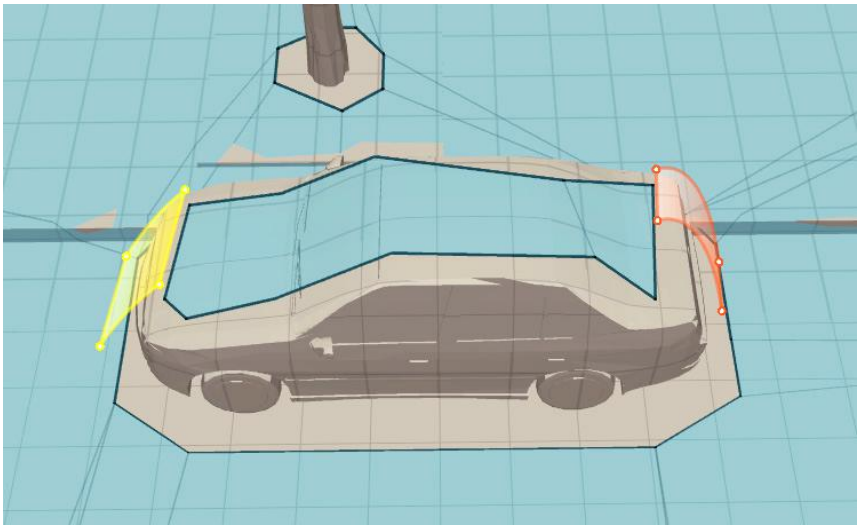


Figure 66: *A car as an example of two unconnected islands.*

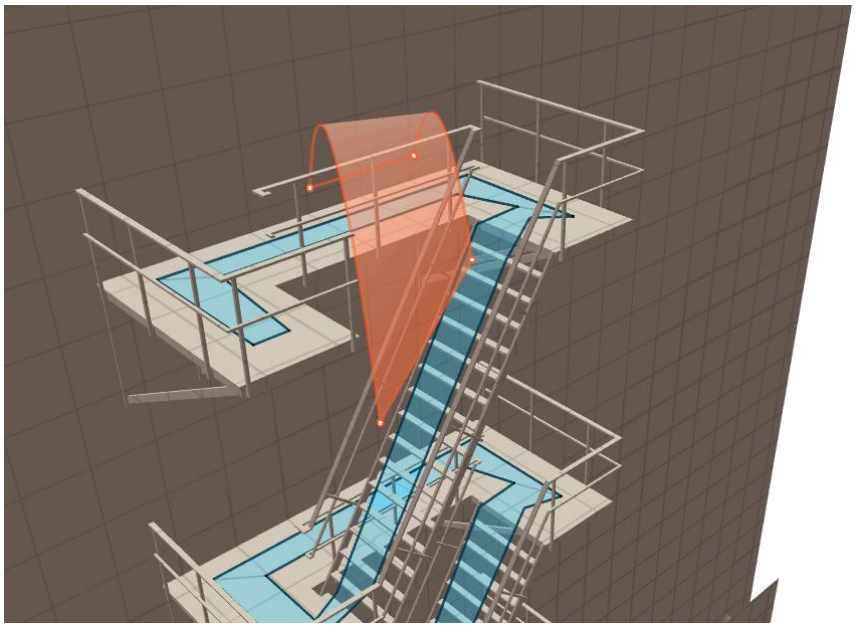


Figure 67: *A jump over the railing of a fire escape as a shortcut example.*

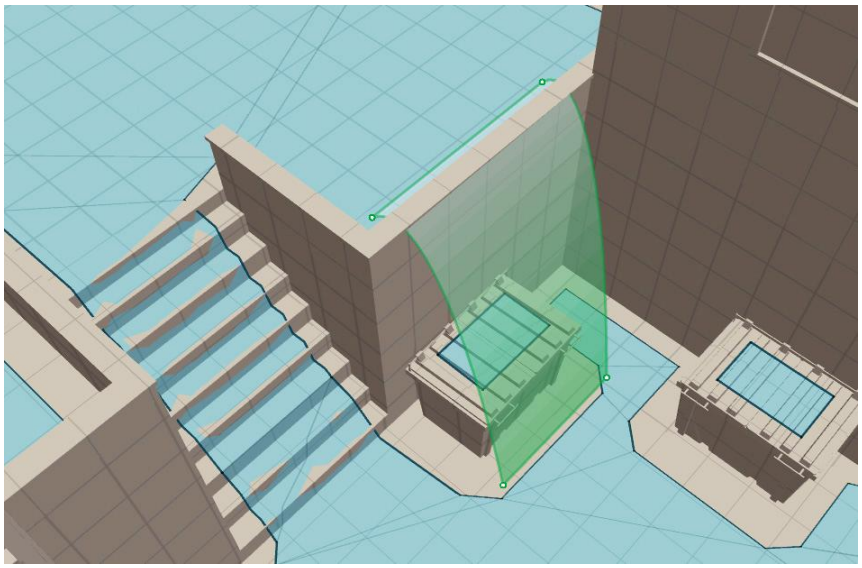


Figure 68: *A jump as an alternative path to the stairs.*

In section 3.1 we distinguished three different kinds of jumps we wanted to find. All of these kinds of jumps are found by the solution presented in this thesis. Figure 66, 67 and 68 show examples of the three kinds of wanted jumps from the test environment "cs_east_borough", which will be presented more detailed in the next section. There are actually a lot more jump links possible in these examples but we deliberately only show the once we need to make the examples clear.

Figure 66 illustrates a jump between two unconnected islands, where one is the ground and the other one is the top of a car. The jump up onto the engine bonnet and the jump down from the car boot connect the navigation mesh of the ground with the part of the navigation mesh on the top of the car. Figure 67 visualizes a shortcut jump along a fire escape. A jump down over the railing onto the ladder bypasses the longer way around the corner to the landing position on the ladder. Finally, figure 68 shows a jump over a low wall and a dumpster as an alternative path to the staircase.

The solution even discovers jumps in very complex situations, like the one shown in figure 69. Finding and constructing jump links in such a complex geometry shows that the solution presented in this thesis is highly robust. In order for the tests to be able to handle the multitude of different situations in arbitrary environments, a robust solution is obviously needed, but only by the means of the trajectory lookup table and intelligent handling of edge mapping and splitting, it is possible to consistently find large numbers of different jump links. The figures 70, 71, 72 and 73 show the quality and additional movement capabilities that the generated jump links provide in different complex scenarios. Specifically the most valuable and beneficial jump links have been rendered for clarity.

These screenshots show again that we build the jump links as surfaces, which is good for path smoothing and dynamic usage of the data, whereas

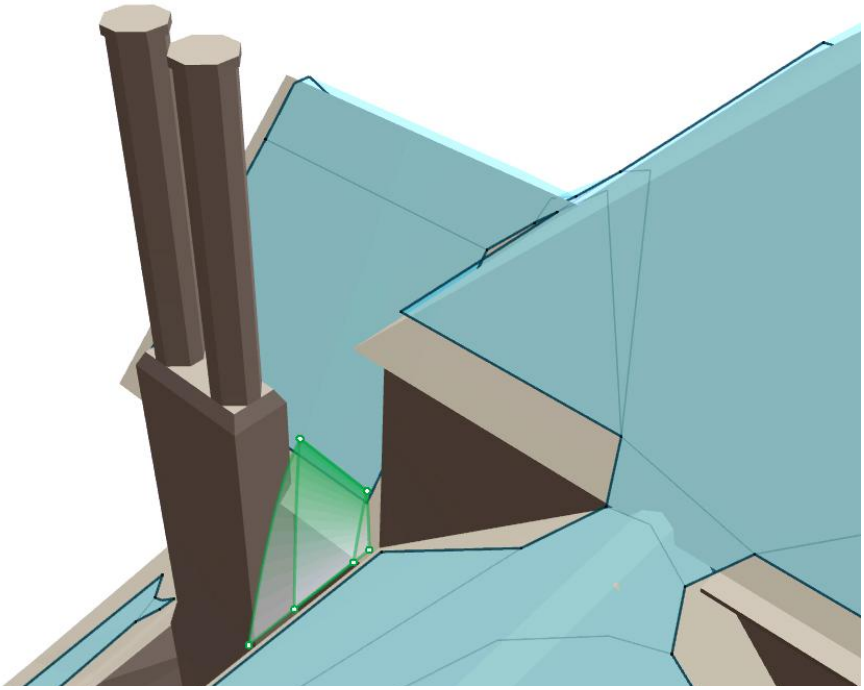


Figure 69: *A jump through a narrow gap between the roof and the chimney.*

Figure 70: *Several jump connections from one roof to another building's roof, granting the agent access to an otherwise unreachable area.*

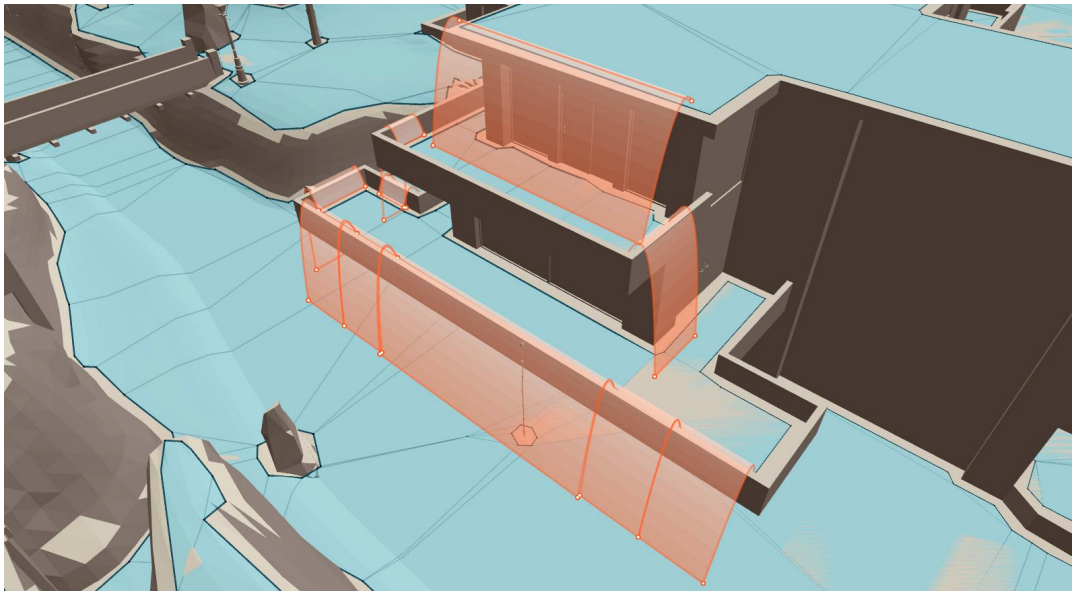


Figure 71: *Downward jumps allowing the agent to jump down from the roof over the balconies onto the ground.*

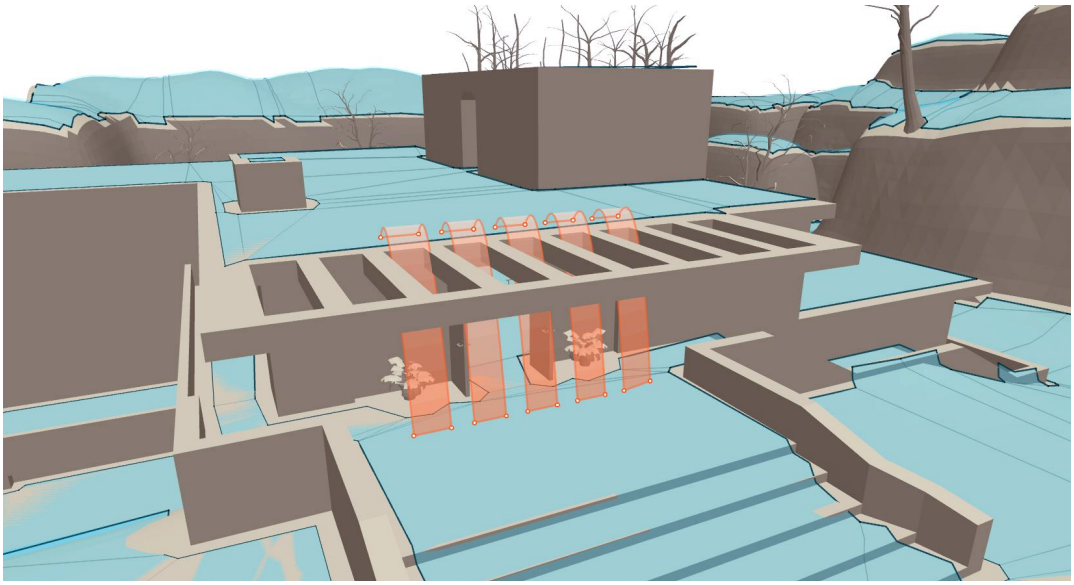


Figure 72: *Jumps down through the awning showing the detail in which jump connections are found.*

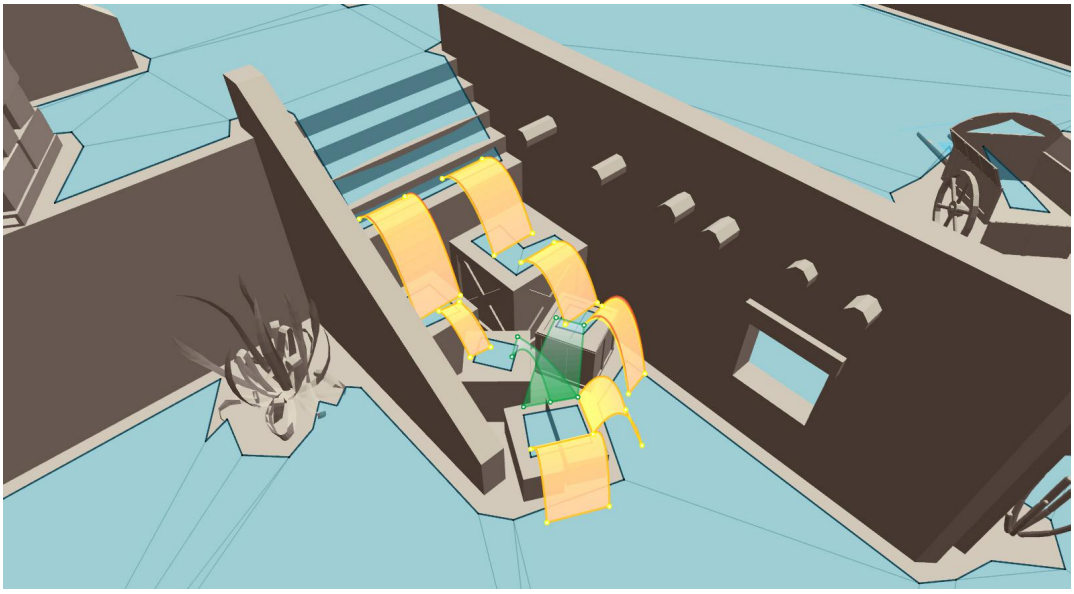
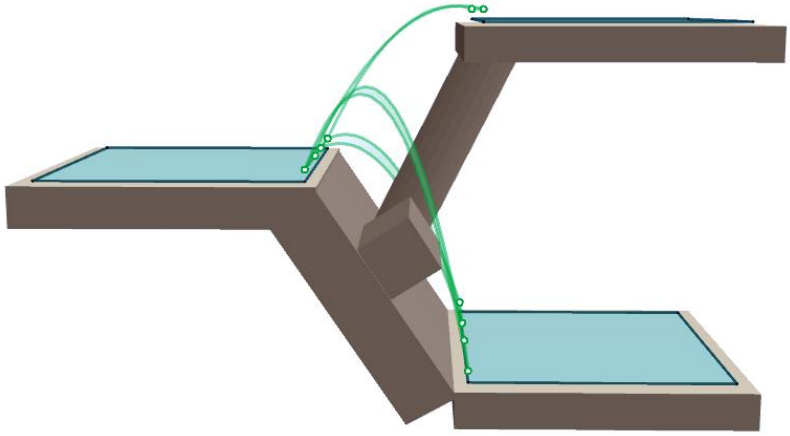
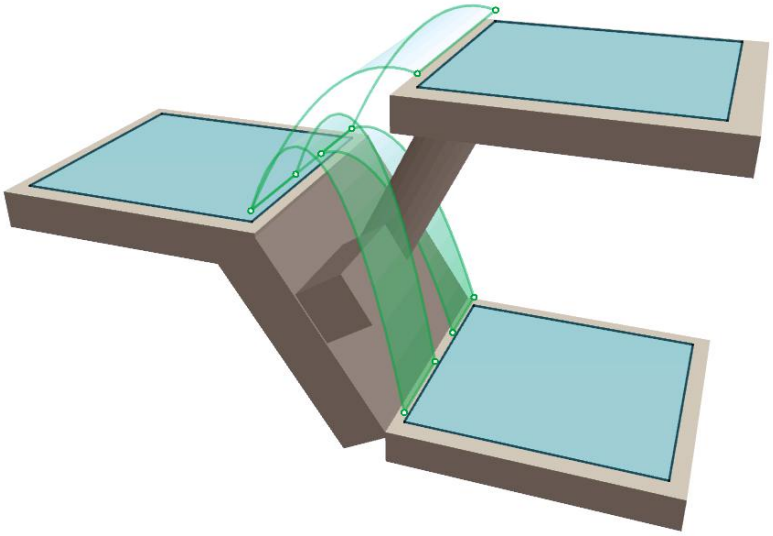


Figure 73: *A normally not traversable broken staircase can now be passed by jumping up and down the crates.*



(a) *side view*



(b) *perspective view*

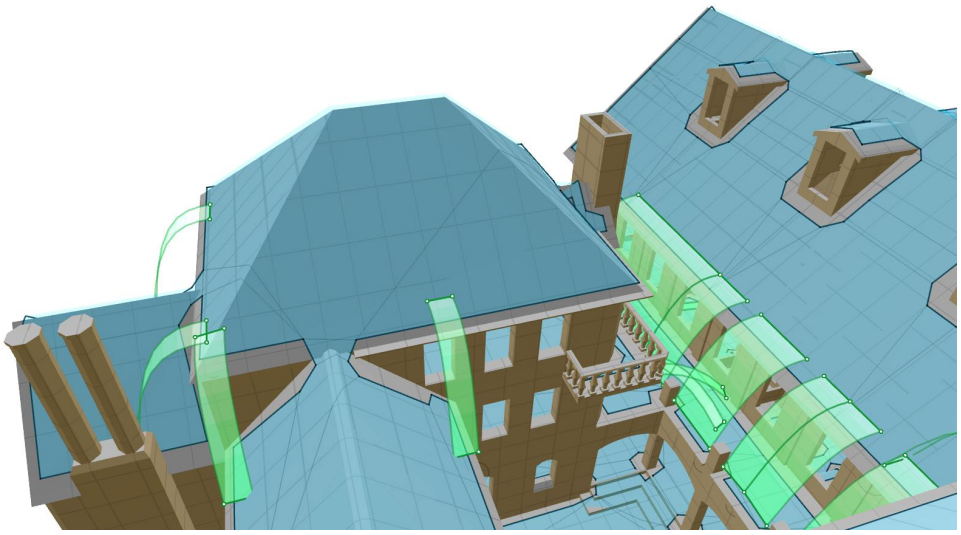
Figure 74: *A complex test scene, that shows specific strength of our solution by finding multiple obstacle avoiding jump links.*

Quake III Arena and *Brink* only find point to point connections that can only be used one way and often store redundant information. Also *Quake III Arena* and *Brink* only find jump down connections where the edges perfectly vertically overlap, whereas we find downward jumps also over gaps and slopes as shown in figure 74. The two jump down connections evade the pillar in the middle, jump over a slope and the left one jumps higher to get over the obstructing box. None of the state of the art solutions is able to evade the box by using alternative trajectories, which is clearly beneficial because that is why our solution finds the left jump down. The number of possible jump trajectories of the state of the art solutions is so small that they almost never find multiple connections from one edge. Because our solution has quick access to many alternative trajectories through the trajectory lookup table, we find multiple jump links for one edge when possible. This results in well connected edges like again shown in figure 74 where the edge from which the jump down links start also jump up onto the higher platform.

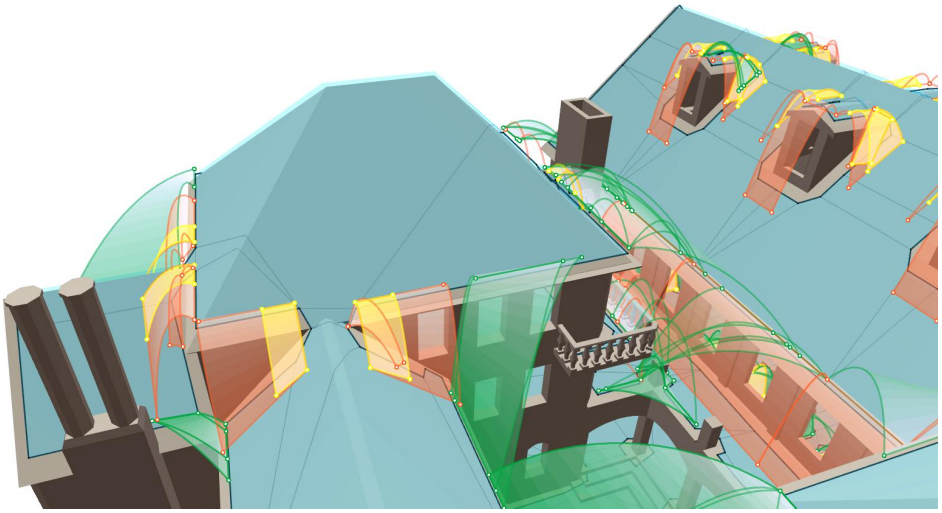
When comparing the screenshots of figure 75 it is visible that the *Killzone 3* implementation finds clean jumps but finds a lot less jumps then the implementation of this thesis. Most obvious are the upward jump links (yellow) and the width of downward jump links (orange) on the left side among the roofs and the additional diagonal jumps (green) near the chimney at the left and the ones that connect the center roof with the neighbouring roofs.

7.2 Study of Jump Link Generation Configurations

All computations were done with the same hardware and the same software settings. The processor used for the calculations was an Intel Core i5-3470 CPU @ 3.20GHz. The software settings include values to define the jump properties, so that jumps can be adjusted to human-like jumps or for example to fit the abilities of a supernatural agent. This way one can get optimal results



(a) *Killzone 3 [Mon12].*



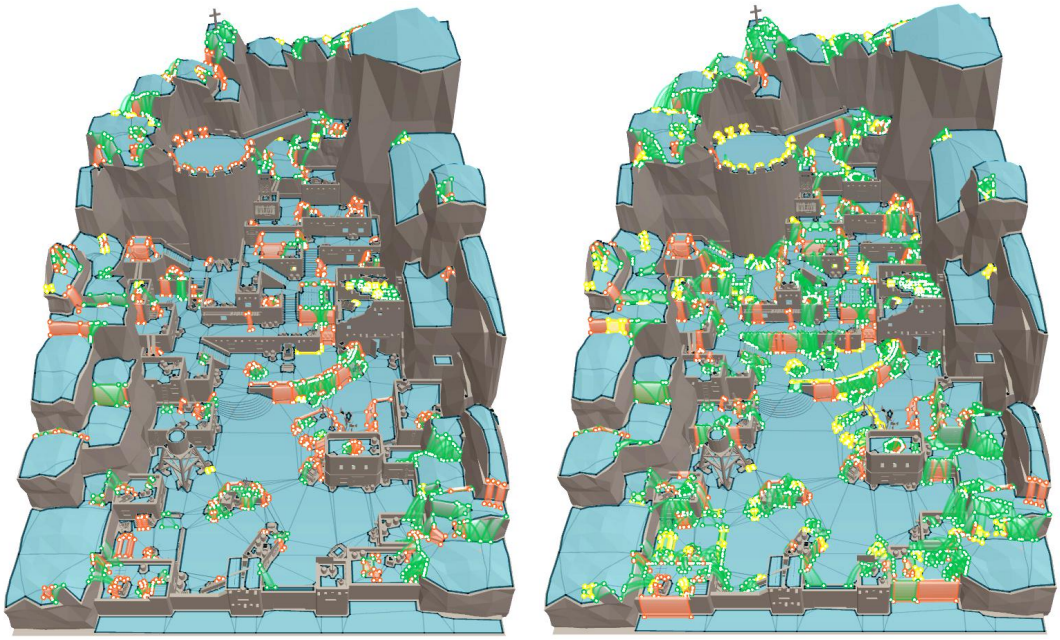
(b) *Implementation of this thesis.*

Figure 75: *A comparison between the jumps found with the Killzone 3 implementation (a) and the implementation presented in this thesis (b).*

for a specific scenario as described in section 3.1. The jumps an agent should be able to perform are stored in the jump trajectory lookup table as described in section 4.4. The lookup table itself as well as the variables for the filling of the table are all tunable by the user. The jump-defining values are the takeoff angle interval, which measures from 1° to 89° with a step size of 0.1°, and the values for the takeoff velocity interval measuring from $0.5\frac{m}{s}$ to $6.0\frac{m}{s}$ with a step size of $0.1\frac{m}{s}$. Another tunable value is the maximal fall depth of an agent, which will be four meters for the following calculations. Additionally, we have values that define the width and height interval mapped to one cell of the jump trajectory lookup table. For the following calculations, both the width and the height interval are 0.1 meters. Finally, there is a value for the mapping angle, which has been described in section 6.2. All of these variables allow us to configure the jumps and the jump trajectory lookup table for specific situations in order to obtain the desired results.

With the above mentioned variable settings for the jump trajectory lookup table, nearly 49 000 different jump trajectories are stored in the table. But the number of different jump trajectories alone does not necessarily have an impact on the number of found jumps. For example, an angle and velocity step size of 0.2° respectively $0.2\frac{m}{s}$ result in approximately 12 000 different jump trajectories, which is a quarter of the previously used trajectories. However, for the number of generated jump links on the map "cs_desperados" this results in only 4.6% less jump links.

By contrast, if the maximal velocity is reduced to $4.0\frac{m}{s}$, the number of different jump trajectories only shrinks to around 32 000 but this results in 37.2% fewer generated jump links on the test environment "cs_desperados". Obviously there exists a coherence between the velocity and the jump width, concluding that a reduced maximal velocity results in a smaller maximal jump width. So there are not just lower numbers of trajectories in the jump



(a) *jump velocity* $4.0\frac{m}{s}$. ***(b)*** *jump velocity* $6.0\frac{m}{s}$.

Figure 76: *The "cs_desperados" test environment with all found jumps, each with a different maximal jump velocity.*

trajectory lookup table, but there are more empty cells in the area of the lookup table that features trajectories with long jump widths, which corresponds to the smaller maximal jump width. Overall this adds up to the conclusion that not the number of different jump trajectories is most meaningful for the generated jump link count, but rather the amount of filled cells of the jump trajectory lookup table has an immense impact on the amount of found jump links. In numbers, a jump trajectory lookup table with a maximal velocity of $4.0\frac{m}{s}$ has 865 filled cells while a table with $6.0\frac{m}{s}$ maximal velocity has 2048 filled cells, meaning that the first table has around 41.5% fewer cells filled. Figure 76 shows these circumstances with respect to the map "cs_desperados".

(a) *An correct trajectory resulting from a Lookup table width and height interval of* $0.1m$.

(b) *An incorrect trajectory resulting from a lookup table width and height interval of* $10.0m$.

Figure 77: *Two jump links generated with different width and height intervals of the jump trajectory lookup table, illustrating the error (b) that can occur if the cell interval is too large.*

From now on the jump trajectory lookup table is always created with $6.0\frac{m}{s}$ as maximal velocity. In figure 77 two jump links are visualized, these jump links are generated with different width and height interval sizes of the

jump trajectory lookup table. As illustrated in figure 77(b) a large interval size causes a jump trajectory that would normally overcome just a small horizontal distance is falsely stretched to overcome a big gap. This trajectory does not look natural nor believable, so it is undesired. On the other hand a very small width and height interval of the jump trajectory table, especially with greater step sizes for the angle and velocity, could create table cells which are empty between filled cells. This could produce issues if for one pair of takeoff point and landing point no connecting trajectory is found. This means when the cell size is reduced, the step sizes of the angle and velocity have to be decreased, so when the table is filled it is guaranteed that every cell has entries. Concluding, the width and height interval have to be adapted to all other user defined variables. As default value for the jump trajectory lookup table width and height step size we choose $0.1m$ and everything else remains as described above.

7.3 Quantitative Evaluation

With the previously described values we generated jump links for the test environments which are summarized in table 1. For instance the map "cs_-abbey" has a navigation mesh with 1207 nodes and 1196 edges. For this map 769 jump links were generated. Downward jumps often generate multiple navigation graph edges because they can land in multiple navigation mesh polygons. The 769 jump links result in 1312 edges that have to be added to the navigation mesh in order to integrate the jump links into it. In this case the number of edges in the navigation mesh was approximately doubled, resulting in 110% more edges in the navigation mesh after the jump link integration. On average, the edge count in the navigation mesh was doubled for all maps.

Storing the jump links as navigation mesh nodes is useless, because then

map name	cs_abbey	cs_desperados	cs_east_borough	cs_napoli	cs_office	cs_parkhouse	cs_alexandra2	cs_corse
navigation graph node count	1207	1526	1353	1536	1282	1521	1018	1615
navigation graph edge count	1196	1241	1438	1486	1196	1480	875	1525
jump link count	769	1755	830	967	660	1280	741	1340
additional graph edges	1312	3135	1374	1642	1166	2331	1294	2244
graph edge count increase	110%	254%	96%	110%	97%	158%	148%	147%

Table 1: *Quantitative analysis of the test environments.*

jumps of the class *jump from edge onto edge* would always need exactly two edges to be connected to the two navigation mesh polygons the jump connects. This is just redundant data, one edge is always more efficient and does not miss any data. Storing jumps generated by the *Jump into Polygon Test* as nodes would always lead to one navigation mesh edge more than storing the jump link as navigation mesh edges. Because its one edge connecting to the jump's takeoff polygon and the other edges connecting the target polygons, otherwise for each navigation mesh polygon the virtual landing edge intersects one navigation mesh edge is saved. The conclusion is clear that jump links have to be navigation mesh edges, that perfectly represents the semantic meaning of the jump link within the pathfinding logic as well as is the most efficient data model to hold the jump information.

The doubled number of edges in the navigation mesh will obviously result in an increased computation time of a pathfinding query. In our test environments some areas with a high density of jump links could be identified. Figure 78 shows such an area from the test environment "cs_desperados". In the middle are some clean jumps down (orange) that were found between

the obstructing timber. On the other hand there is a rather large number of jump links of the class *jump from edge onto edge* (green) on the right and left side of the screenshot. One would intuitively suggest that fewer jumps should be enough to adequately represent jumping capabilities in these areas. The reason for this high density of jumps around the barrels is that for the navigation mesh to correctly envelop the barrels, a lot of small outline edges are necessary. There is also a high number of jumps along the stairs because the railing, the wall and the timber inside of this wall result in complex obstructions which lead to a high number of thin but optimal jump links.

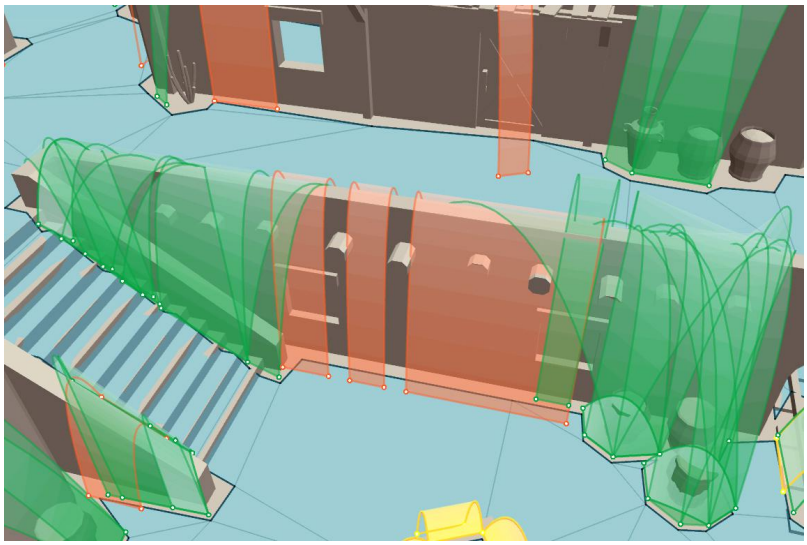


Figure 78: *High environment detail leading to high jump link density.*

The main problem that exists is that for a practical application, the number of jumps should probably be reduced because this high density of jumps most likely does not yield a large benefit over an intelligently reduced number of jumps. An extremely helpful future study would be the development of a method to conclusively calculate a usefulness value for every jump link,

enabling an easy and continuous adjustment of the detail level of the jump link network.

7.4 Time Evaluation and Real Time Capabilities

In the previous section we saw that the generated jump links double the number of edges of the navigation mesh. This section will take a look at how long the generation of jump links takes and whether it is possible to update the jump link network in real time.

map name	cs_abbey	cs_desperados	cs_east_borough	cs_napoli	cs_office_unlimited	cs_parkhouse	de_alexandra2	de_corse
size in meter	80×75	75×110	175×175	125×130	110×105	145×160	95×80	130×130
outer edge count	2544	3609	2737	3193	2696	3271	2355	3426
jumps tested	248 259	268 984	70 252	158 683	159 079	206 066	150 566	205 043
generated jump links	769	1755	830	967	660	1280	741	1340
run time	17.3*s*	26.3*s*	11.7*s*	26.2*s*	14.8*s*	21.0*s*	12.9*s*	18.5*s*

Table 2: *Run time for all test environments.*

In table 2, the run times for all test environments are summarized. The environment "cs_abbey" has a size of $80m \times 75m$ with a navigation mesh including 2544 outer edges. The *Jump onto Edge Test* is executed once for every outer edge and consequently has a linear run time. The *Jump into Polygon Test* has a quadratic run time because it checks for jump links between edge pairs that are within the maximal jump range of each other. All of these 248 259 tests combined generate 769 jump links in 17.3*s* for "cs_abbey".

The way the tests are designed to use an outer edge or a pair of outer edges implies that local changes of the navigation mesh only lead to changes of jump links that are directly connected to this part of the navigation mesh. Because of that consistency we can examine whether an update of a part of the jump link network is possible in real time.

trajectories per cell limit	*total entries*	*jump link count*	*average time*
unlimited	591 213	1 755	26.3*s*
100	189 731	1 724	10.2*s*
9	18 572	1 719	6.7*s*
6	12 428	1 717	6.6*s*
3	6 239	1 716	6.5*s*

Table 3: *Different maximum lookup table cell entries and their effects on the* 268 984 *executed tests for the example of the test environment "cs_desperados".*

In table 3, different restrictions on the maximum trajectory count per jump trajectory lookup table cell are examined. As can be seen in the first line, the lookup table has nearly 600 000 entries in all cells if the entries per cell are not restricted. This leads to a very long run time, because even if a jump is totally obstructed, all trajectory entries of the corresponding cell are still tested. When reducing the trajectory entries per cell limit to 9, 6 or 3 and checking the results in table 3, it becomes clear that this has a minimal impact on the number of found jumps but drastically improves performance. The reduction is done by evenly adding trajectories from the sorted list, so that

there are trajectories from every section of trajectory length in the reduced set. This way we get an even combination of high, short and intermediate jump trajectories in the lookup table.

As presented in table 3, the number of found jumps differs only slightly between 3, 6 and 9 entries per jump trajectory lookup table cell. Since we want to assess the real time capabilities, we value the run time benefit over the small chance to miss a jump link and therefore choose three entries per lookup table cell for our real time test.

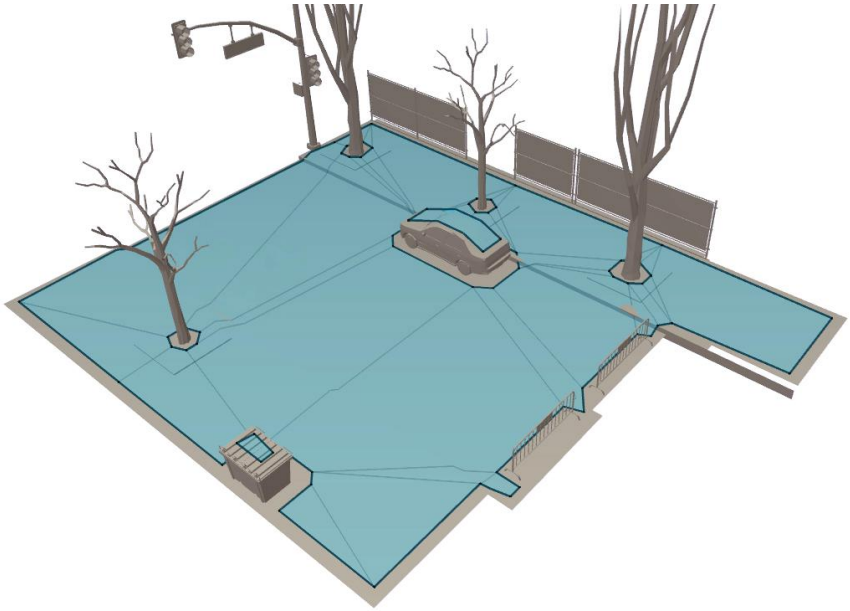


Figure 79: *The real time test environment before the jump link generation.*

We take a car as the dynamically moving object that enters a static state and thus changes the surrounding navigation mesh as a test scenario. This scenario has the size which needs to be computable in real time for our solution to be real time capable. The test scenario is an extracted part from

the test environment “cs_east_borough”, which is rendered without jumps in figure 79. To completely generate all jump links for this test scenario 884 test are executed, it takes 18.7 milliseconds at a 10% CPU load. This time is short enough for humans to perceive the generation as real time and the CPU load leaves a lot of room for parallel processes. The test scenario with the rendered jump links is shown in figure 80. With this evaluation we conclude that the solution satisfies the problem definition in section 3.1 by performing the jump generation while maintaining good performance.

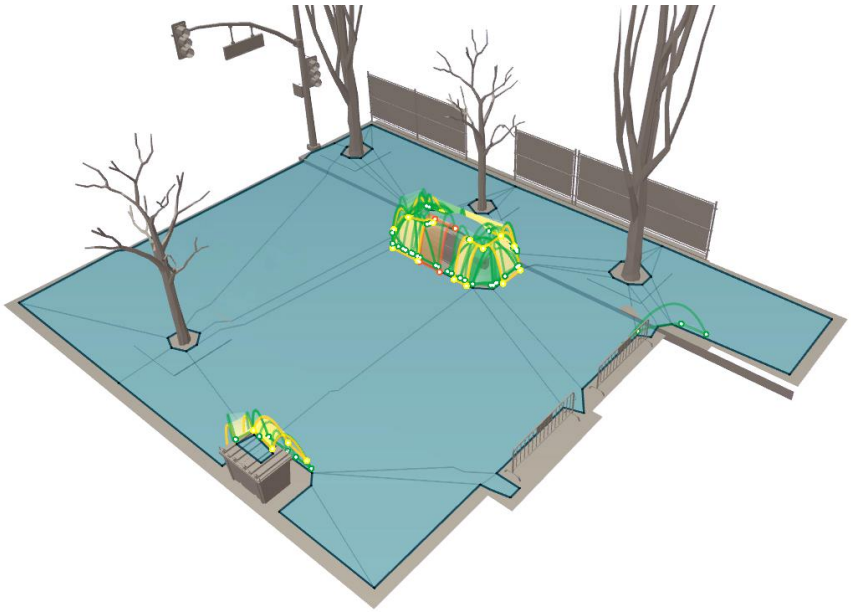


Figure 80: *The real time test environment with the generated jump links.*

8 Conclusion and Future Work

We have shown that the jump generation techniques that were developed over the course of this diploma thesis consistently find high numbers of jumps of good quality even in very complex environments. Furthermore the evaluation of the experiments shows that the algorithm is fast and can compute local changes in real time. The presented data model allows for dynamic jump movements and integrates well with the navigation mesh data structure. The jump trajectory lookup table as well as the jumping capabilities of the agent can be modified to fit different fitness degrees and behavioral scenarios.

The number of found jumps resulted on average in a doubling of the navigation graph links. The experiments show that a significant amount of these jumps represent redundant or comparably suboptimal connections. To improve usage run time for a practical application in the future, it would be extremely valuable to develop criterion of the usefulness of a jump link. For example, a valuing function that allows the jumps to be put into a linear order of usefulness. This would allow a seamless level of detail of the number of actually used jump links. Considering the fact that there are many more movement capabilities to be generated, especially in the field of climbing and closer interaction with obstructions, a controllable reduction to the most valuable jump data is the first step to make room for the next complexity of movement data.

A future implementation of path smoothing for jump links in conjunction with a practical application of character animation to visually utilize the newly available jump data would be extremely interesting, because it most probably would reveal more specific needs and possibilities of complex character movement in the perspective of a complete solution and not just the

data generation side.

This thesis has provided virtual agents with extensive dynamic jumping capabilities enabling them to realistically traverse complex and heavily obstructed terrain and to reach areas that would otherwise not be accessible. The developed methods are an important step towards fully realistic virtual agent movement and will hopefully provide a foundation for further studies.

References

[Axe08] Ramon Axelrod. "Navigation Graph Generation in Highly Dynamic Worlds". In: *AI Game Programming Wisdom 4*. Ed. by Steve Rabin. Charles River Media, 2008, pp. 125–141.

[Bri12] Brink Wiki. *SMART*. 2012. URL: http://brink.wikia.com/wiki/SMART.

[BSL04] Paul Brobst, Ramesh Saran, and Michael Lent. "Dynamic Path Planning and Terrain Analysis for Games". In: *Workshops at the Twentieth National Conference on Artificial Intelligence*. 2004.

[Cha11] Alex J. Champandard. *Recast'ing Automatic Annotations for Player Cover in KILLZONE 3*. Apr. 2011. URL: http://aigamedev.com/open/review/player-cover-killzone3/ (visited on 02/13/2013).

[Cry] Crytek UK Ltd. *Navigation Setup*. URL: http://sdk.crydev.net/display/SDKDOC21/Navigation+Setup (visited on 02/13/2013).

[Far06] Fredrik Farnstrom. "Improving on Near-Optimality: More Techniques for Building Navigation Meshes". In: *AI Game Programming Wisdom 3*. Ed. by Steve Rabin. Charles River Media, 2006, pp. 113–128.

[Fen03] Walter Fendt. *Der schiefe Wurf*. Apr. 2003. URL: http://www.walter-fendt.de/phys/mech/wurf.pdf.

[Fun09] John D. Funge. *Artificial Intelligence For Computer Games: An Introduction*. Peters Corp., 2009.

[Hal12] Arne-Olav Hallingstad. "Vault, Slide, Mantle - Building Brink's SMART System". In: GDC 2012, May 2012.

[MS08] Colt McAnlis and James Stewart. "Intrinsic Detail in Navigation Mesh Generation". In: *AI Game Programming Wisdom 4*. Ed. by Steve Rabin. Charles River Media, 2008, pp. 95–112.

[Mon13] Mikko Mononen. Personal interview. Jan. 2013.

[Mon11a] Mikko Mononen. *Paris Game/AI Conference 2011 Slides and Demo*. July 2011. URL: http://digestingduck.blogspot.de/2011/07/paris-gameai-conference-2011-slides-and.html (visited on 02/13/2013).

[Mon11b] Mikkon Mononen. "Automatic annotations in Killzone 3 and Beyond". In: Paris Game/AI Conference 2011, 2011.

[Mon12] Mikkon Mononen. *recastnavigation*. 2012. URL: http://code.google.com/p/recastnavigation/.

[Mül04] Rainer Müller. "Mechanik in Alltagskontexten". In: *Studienmaterial zu interdisziplinären Aspekten der Naturwissenschaftsdidaktiken*. Lit Verlag, 2004.

[Pat12] Amit Patel. *Map representations*. 2012. URL: http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html (visited on 02/13/2013).

[Pea11] Craig Pearson. *Brink review*. 15.2.2013. May 2011. URL: http://www.pcgamer.com/review/brink-review/ (visited on 02/13/2013).

[Pin01] Marco Pinter. *Toward More Realistic Pathfinding*. 2001. URL: http://www.gamasutra.com/view/feature/131505/toward_more_realistic_pathfinding.php?print=1 (visited on 02/13/2013).

[RG04] Christopher Reed and Benjamin Geisler. "Jumping, Climbing, and Tactical Reasoning: How to Get More Out of a Navigation System". In: *AI Game Programming Wisdom 2*. Charles River Media, 2004.

[Smi02] Patrick Smith. *GDC 2002: Polygon Soup for the Programmer's Soul: 3D Pathfinding*. 2002. URL: http://www.gamasutra.com/view/feature/131409/gdc_2002_polygon_soup_for_the_.php?print=1.

[Sno00] Greg Snook. "Simplified 3D Movement and Pathfinding Using Navigation Meshes". In: *Game Programming Gems*. Ed. by Mark DeLoura. Charles River Media, 2000, pp. 288–304.

[Son10] Sony Computer Entertainment Europe. *Killzone 3*. 2010. URL: http://www.killzone.com/kz3/de_DE/thegame-overview/campaign/overview.html (visited on 02/13/2013).

[Spl13] Splash Damage Ltd. *Brink – Splash Damage*. 2013. URL: http://www.splashdamage.com/brink (visited on 02/13/2013).

[Toz02] Paul Tozour. "Building a Near-Optimal Navigation Mesh". In: *AI Game Programming Wisdom*. Ed. by Steve Rabin. Charles River Media, Inc., 2002, pp. 171–185. ISBN: 1-58450-077-8,

[Toz08] Paul Tozour. *Fixing Pathfinding Once and For All*. July 2008. URL: http://www.ai-blog.net/archives/000152.html (visited on 02/13/2013).

[Toz04] Paul Tozour. "Search Space Representations". In: *AI Game Programming Wisdom 2*. Ed. by Steve Rabin. Charles River Media, 2004, pp. 85–102.

[Val13] Valve Corporation. *Counter-Strike: Source*. 2013. URL: http://www.valvesoftware.com/games/css.html (visited on 03/09/2013).

[Wav01] J.M.P. van Waveren. "The Quake III Arena Bot". MA thesis. 2001, 2001.

[WC02] Stephen White and Christopher Christensen. "A Fast Approach to Navigation Meshes". In: *Game Programming Gems 3*. Ed. by Dante Treglia. Charles River Media, 2002, pp. 307–320.

[Wik13] Wikipedia. *Quake III Arena — Wikipedia, The Free Encyclopedia*. 2013. URL: \url{http://en.wikipedia.org/w/index.php?title=Quake_III_Arena&oldid=535324122} (visited on 02/13/2013).

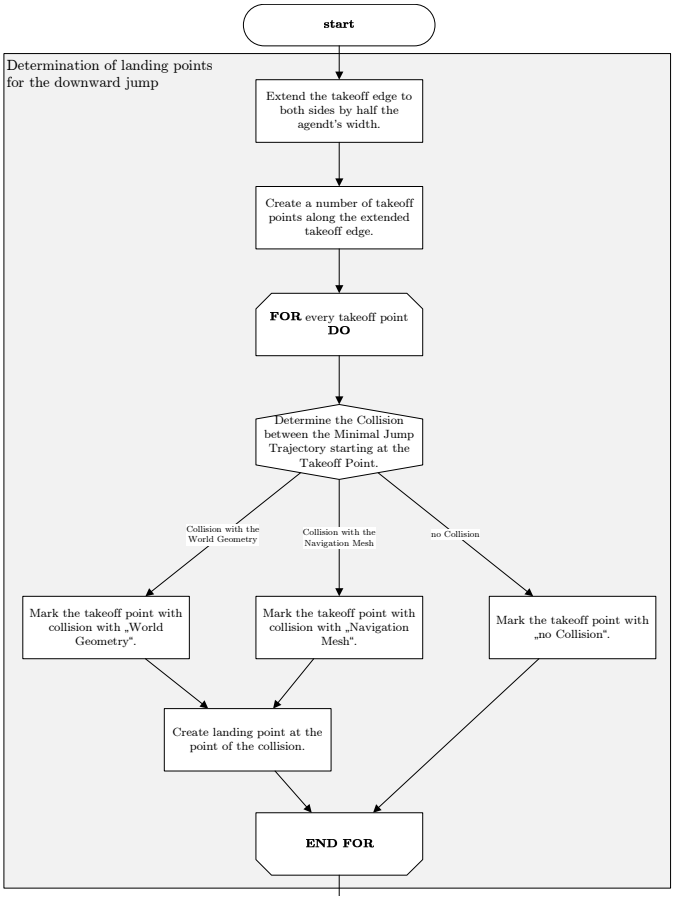
[xai] xaitment GmbH. URL: http://www.xaitment.com/english/products/xaitmap/screenshots.html.

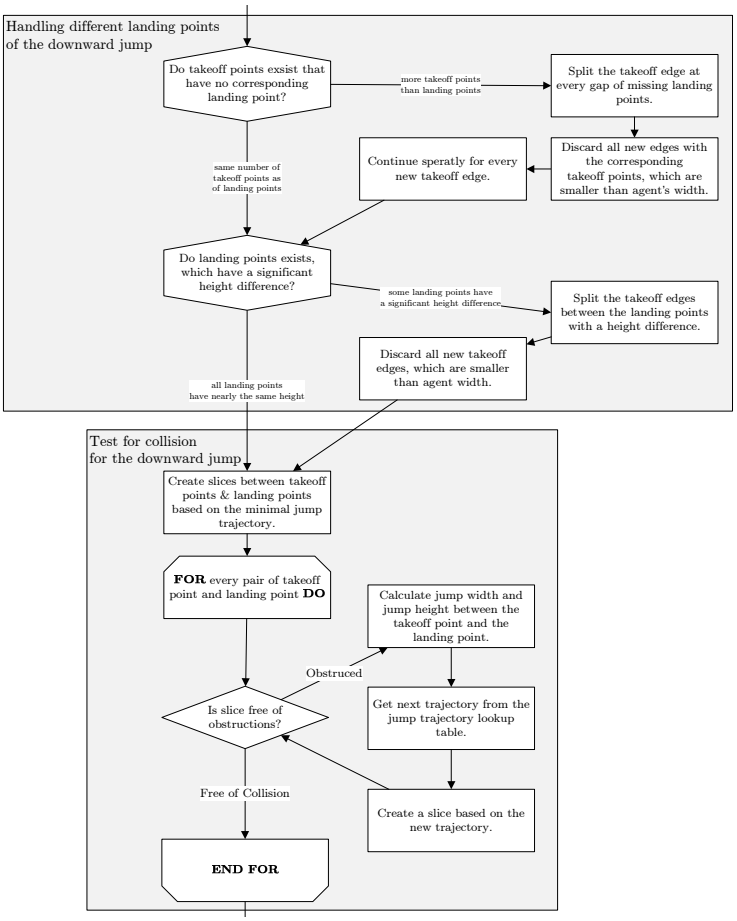
List of Figures

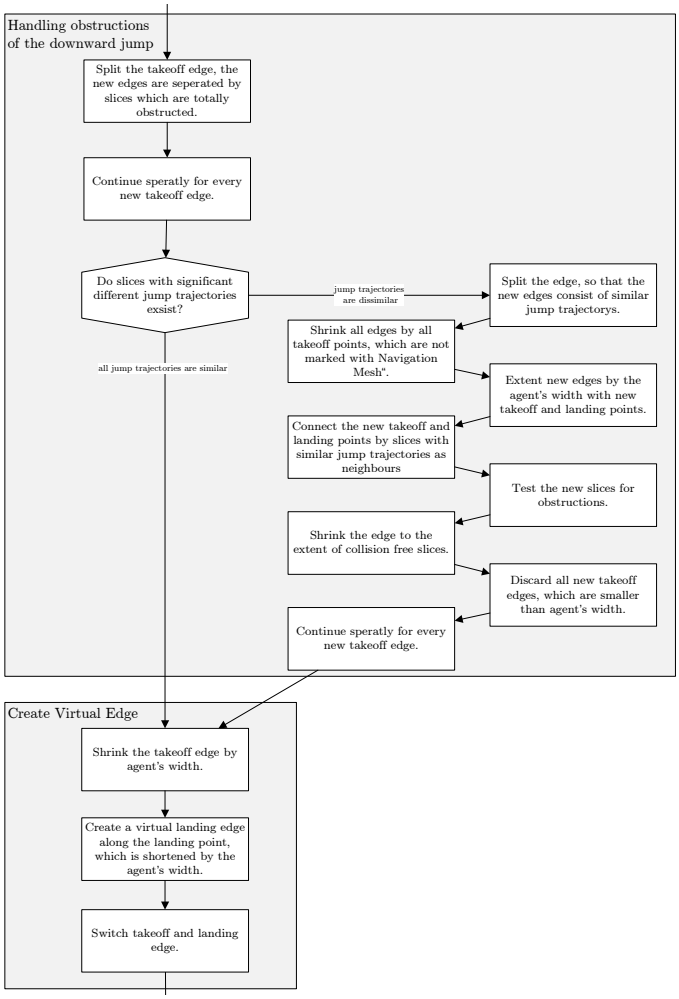
Appendices

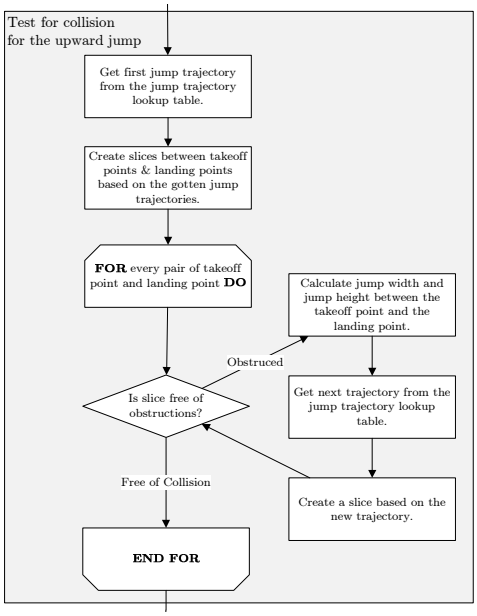
A Detailed Flow Charts

Jump into Polygon Test




Handling different landing points of the downward jump
Do takeoff points exsist that have no corresponding landing point?
more takeoff points than landing points
Split the takeoff edge at every gap of missing landing points.
Discard all new edges with the corresponding takeoff points, which are smaller than agent's width.
Continue speratly for every new takeoff edge.
same number of takeoff points as of landing points
Do landing points exists, which have a significant height difference?
some landing points have a significant height difference
Split the takeoff edges between the landing points with a height difference.
Discard all new takeoff edges, which are smaller than agent width.
all landing points have nearly the same height
Test for collision for the downward jump
Create slices between takeoff points & landing points based on the minimal jump trajectory.
FOR every pair of takeoff point and landing point DO
Calculate jump width and jump height between the takeoff point and the landing point.
Obstruced
Is slice free of obstructions?
Get next trajectory from the jump trajectory lookup table.
Free of Collision
Create a slice based on the new trajectory.
END FOR


Handling obstructions of the downward jump
Split the takeoff edge, the new edges are seperated by slices which are totally obstructed.
Continue speratly for every new takeoff edge.
Do slices with significant different jump trajectories exsist?
jump trajectories are dissimilar
Split the edge, so that the new edges consist of similar jump trajectorys.
Shrink all edges by all takeoff points, which are not marked with Navigation Mesh*.
Extent new edges by the agent's width with new takeoff and landing points.
Connect the new takeoff and landing points by slices with similar jump trajectories as neighbours
Test the new slices for obstructions.
Shrink the edge to the extent of collision free slices.
Discard all new takeoff edges, which are smaller than agent's width.
Continue speratly for every new takeoff edge.
all jump trajectories are similar
Create Virtual Edge
Shrink the takeoff edge by agent's width.
Create a virtual landing edge along the landing point, which is shortened by the agent's width.
Switch takeoff and landing edge.


Test for collision
for the upward jump
Get first jump trajectory from the jump trajectory lookup table.
Create slices between takeoff points & landing points based on the gotten jump trajectories.
FOR every pair of takeoff point and landing point DO
Calculate jump width and jump height between the takeoff point and the landing point.
Obstruced
Is slice free of obstructions?
Get next trajectory from the jump trajectory lookup table.
Free of Collision
Create a slice based on the new trajectory.
END FOR

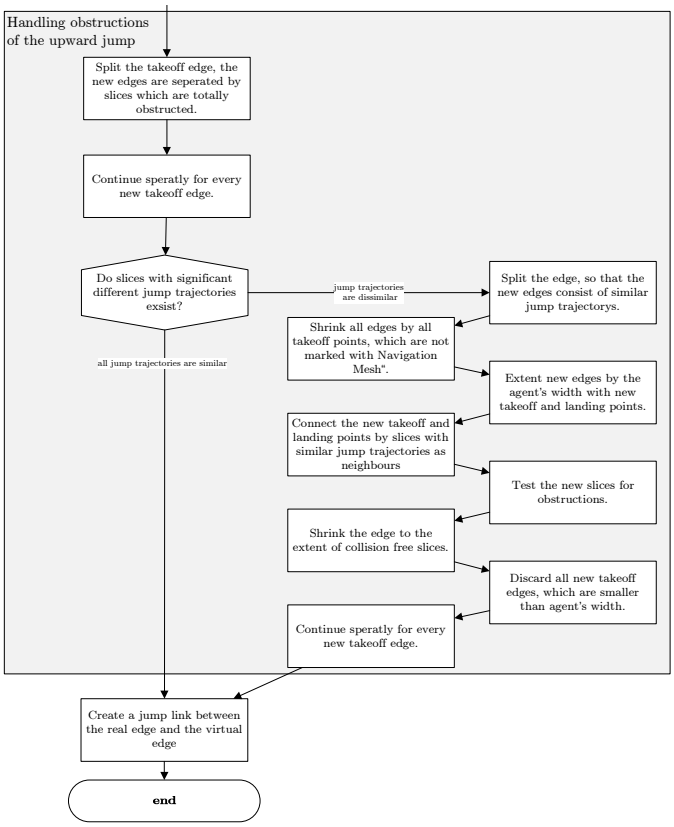
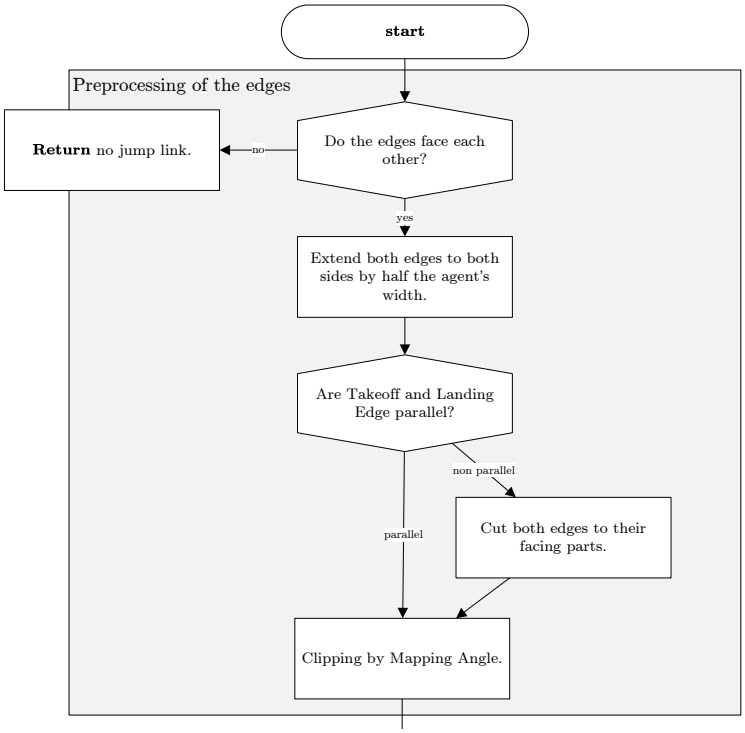
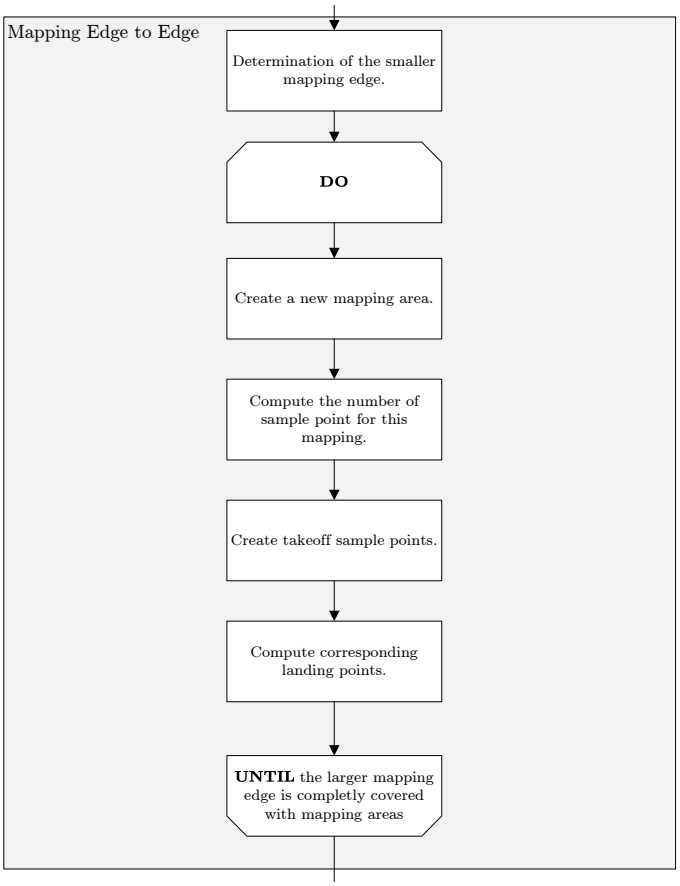
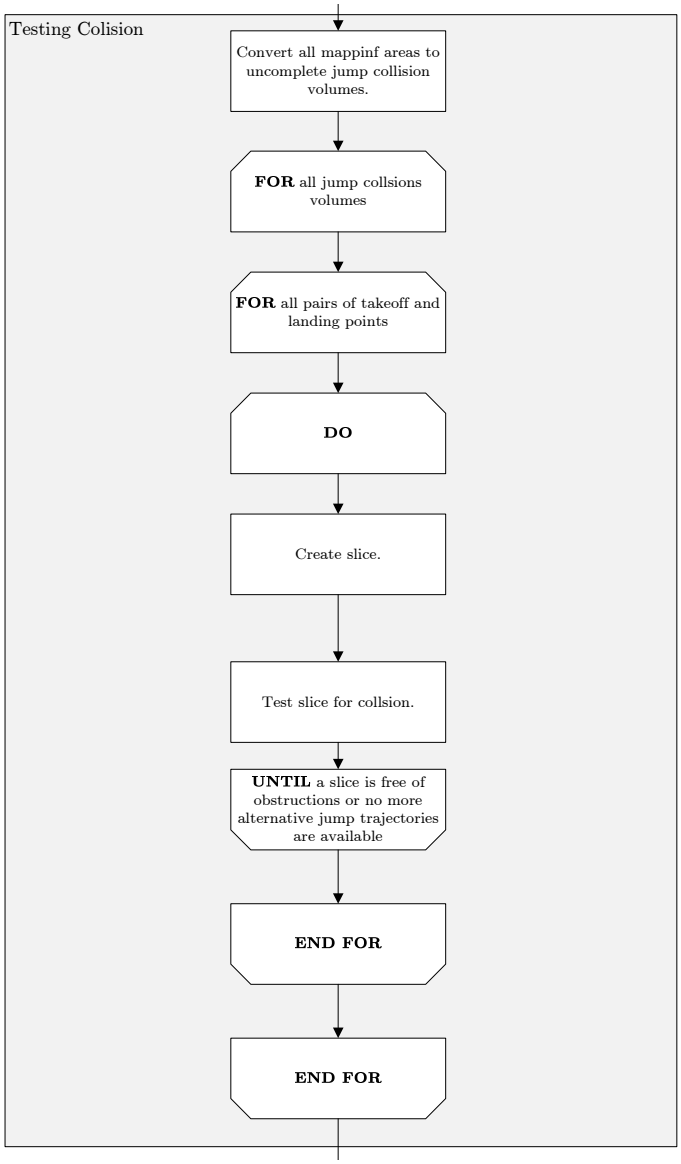


Figure 77: *Detailed Overview of the Jump into Polygon Test*

Jump onto Edge Test




Mapping Edge to Edge
Determination of the smaller mapping edge.
DO
Create a new mapping area.
Compute the number of sample point for this mapping.
Create takeoff sample points.
Compute corresponding landing points.
UNTIL the larger mapping edge is completly covered with mapping areas


Testing Colision
Convert all mappinf areas to uncomplete jump collision volumes.
FOR all jump collsions volumes
FOR all pairs of takeoff and landing points
DO
Create slice.
Test slice for collsion.
UNTIL a slice is free of obstructions or no more alternative jump trajectories are available
END FOR
END FOR

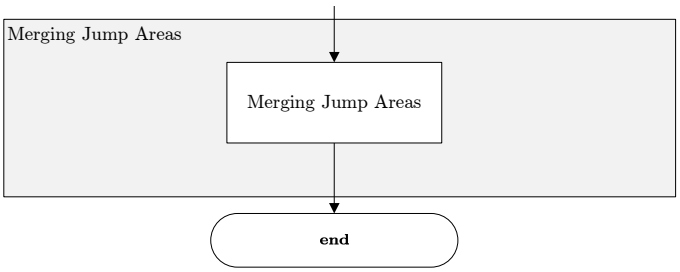
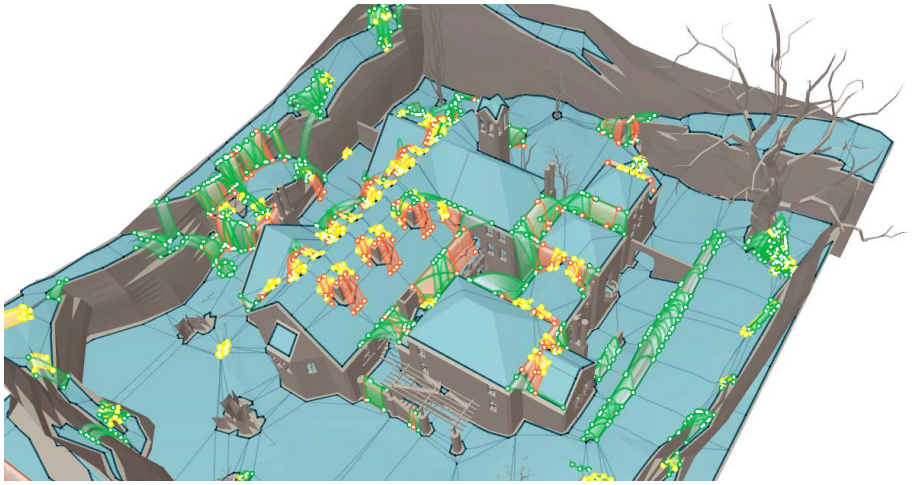


Figure 75: *Detailed Overview of the Jump onto Edge Test*

B Test Environment from *Counter-Strike: Source*

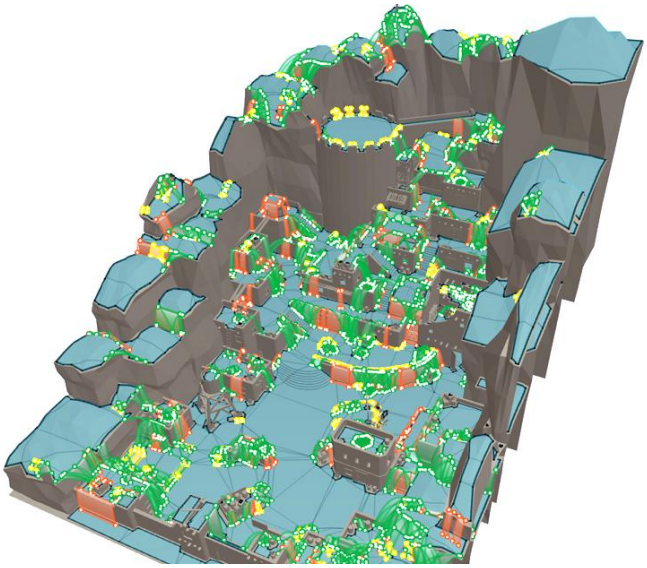
(a) *With Textures*



(b) *With jump links*

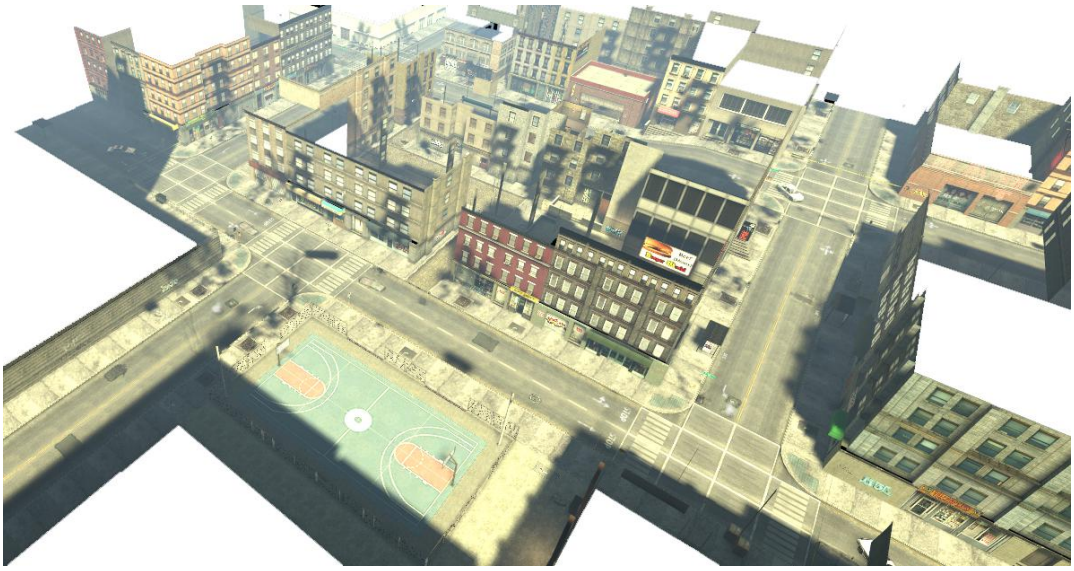
Figure 76: *Test environment "cs_abbey"*

(a) *With Textures*

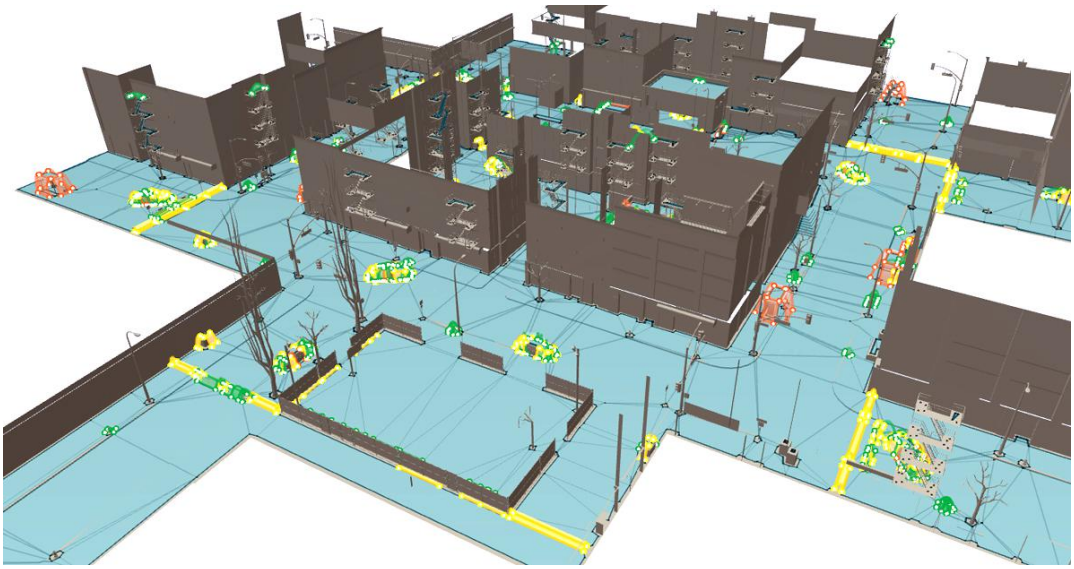


(b) *With jump links*

Figure 77: *Test environment "cs_desperados"*



(a) *With Textures*



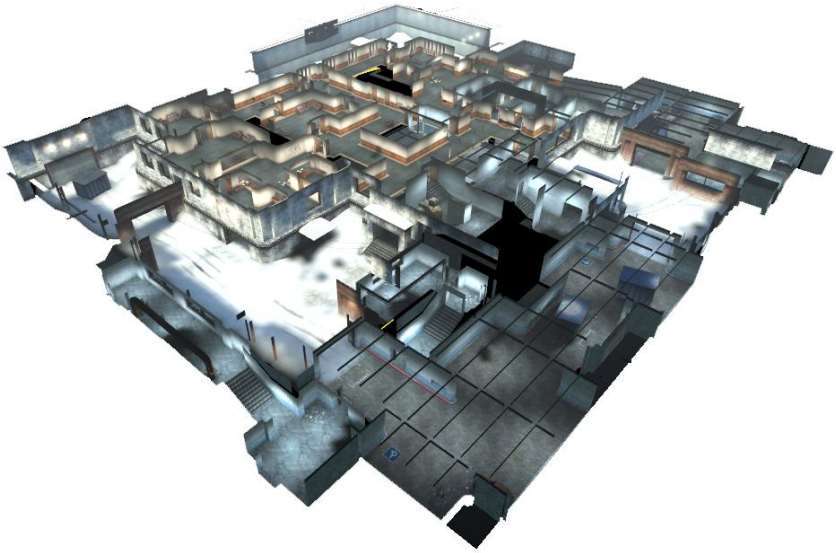
(b) *With jump links*

Figure 78: *Test environment "cs_east_borough"*

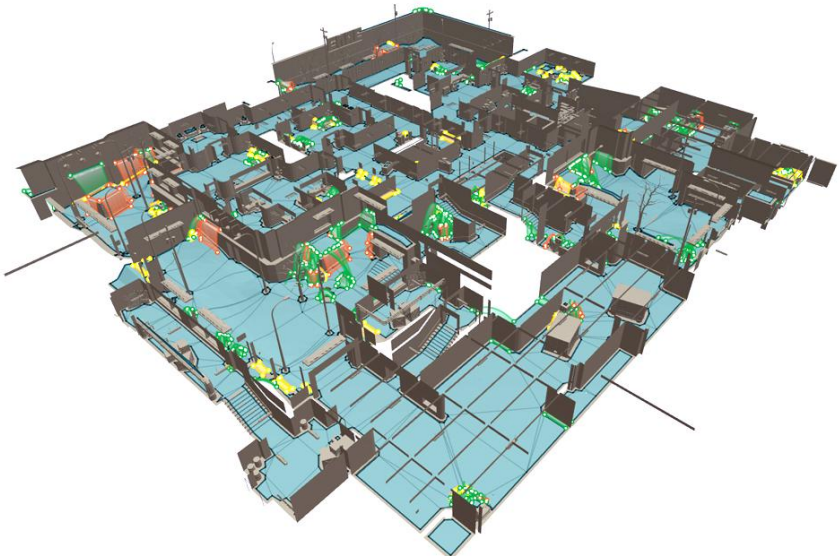
(a) *With Textures*

(b) *With jump links*

Figure 79: *Test environment "cs_napoli"*

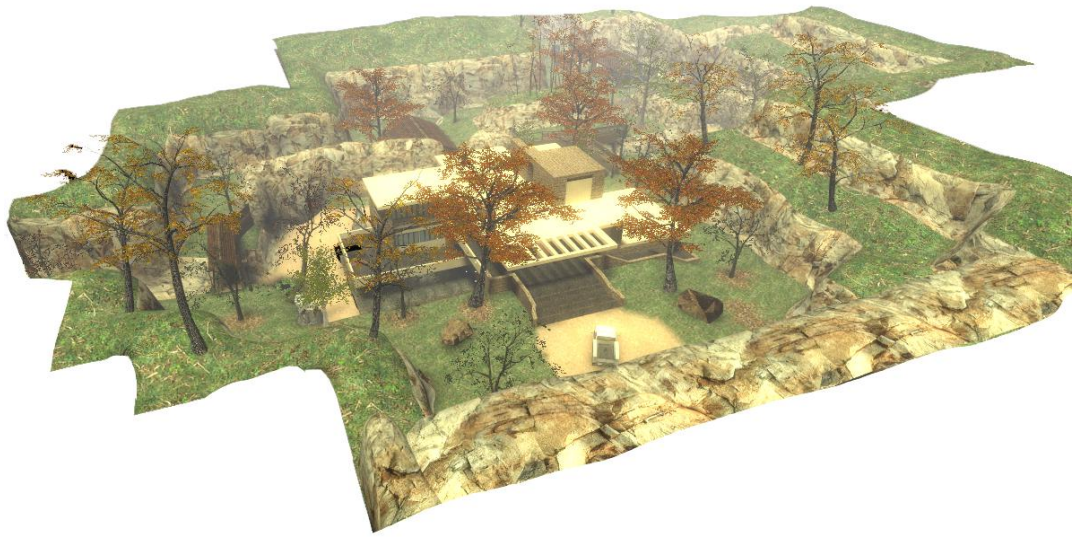


(a) *With Textures*



(b) *With jump links*

Figure 80: *Test environment "cs_ office_unlimited"*

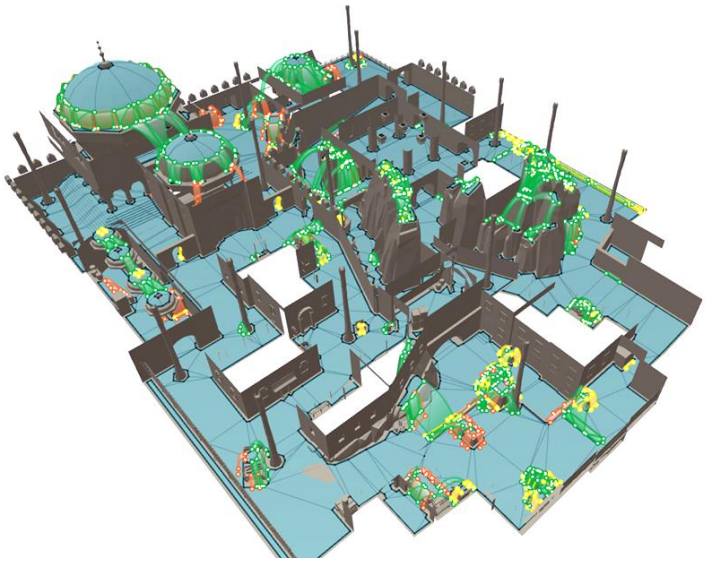


(a) *With Textures*

(b) *With jump links*

Figure 81: *Test environment "cs_parkhouse"*

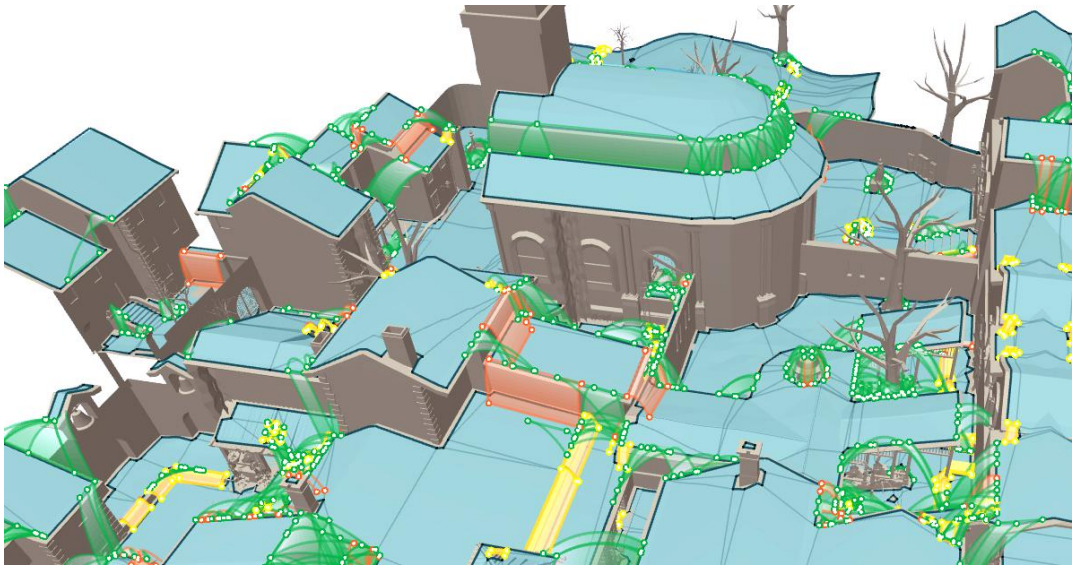
(a) *With Textures*



(b) *With jump links*

Figure 82: *Test environment "de_alexandra2"*

(a) *With Textures*



(b) *With jump links*

Figure 83: *Test environment "de_corse"*

C Interview

Sara Budde: I am Sara Budde and I am currently studying computing science at the Humboldt-Universität zu Berlin, where I am working on my diploma thesis about automatic jump link generation. Mr. Mononen, would you please introduce yourself and briefly state your expertise to explain to the reader why you are qualified for the topic of jump link generation.

Mikko Mononen: My name is Mikko Mononen, I'm currently chief product officer at Tinkercad. I have previously worked on game AI at Crytek GmbH, I've written an open source navigation mesh toolkit Recast & Detour, which is currently being used in many AAA titles at companies such as Guerrilla Games, Insomniac Games, Unity 3D, Epic Games. One part of my research on game navigation included automatic generation of jump link annotation and automatic cover location detection.

Sara Budde: You implemented an automatic jump link annotation at Guerrilla Games for the title *Killzone 3* which you presented at the "Paris Game/AI Conference 2011". Do you know of any other solutions for automatic jump link annotation in 3D environments?

Mikko Mononen: I have not seen any published articles on the subject. I think the most cited and most relevant is maybe the *Quake III* bot thesis from J.M.P. van Waveren [1]. *Brink's* SMART navigation system had similar ideas too [2], as well as *Left4Dead's* zombie climbing [3].

In addition to that, there is extensive research on using existing motion

[1] Quake III thesis: `http://dev.johnstevenson.co.uk/bots/20585341-The-Quake-III-Arena-Bot.pdf`

[2] Vault, Slide, Mantle - Building Brink's SMART System: `http://www.splashdamage.com/publications`

[3] `http://www.valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf`

patterns to find the best possible "move" while following a path. That method is able to find jumps if the example corpus has such behavior. One example of such a system Is James Kuffner's Behavior Planning for Character Animation [4]. The research in this field spans from robotics to animation, but is generally too taxing to real time game use.

As far as I know, there are two commercial game engines which do automatic jump link detection. Unreal Engine has concepts of vaults, and their automatic navigation mesh generation system is able to generate them [5]. Unity3D has automatic jump link detection which borrows ideas from my Paris presentation but actually uses physic engine collision detection tests to sample the validity.

Sara Budde: Your jump test consists of one test for a jump over an obstacle/-gap and one test for a jump down. Each test is based on one fixed jump trajectory. Is there a specific reason for that constraint? Practical reasons like implementation complexity, animation system issues or something completely different?

Mikko Mononen: Your assumption is correct. The AI navigation is generally quite boring to look at, and often one of the main reasons to add jump links is to make the AI behavior more believable and interesting. One way to look at the navigation problem is to divide the AI behavior into actions in a context. I wrote an article about it long time ago: `http://aigamedev.com/open/article/structure-action-game-ai/`

Actions and contexts are the contract between AI coders, technical artists, artist, level designers and animators. One action-context pair could be "vault over 90cm high, and 50cm wide obstacle". That gives instructions to all disciplines how to approach the problem, and in it turn gives the AI programmer the information to implement the trajectory sampling. That

[4] `http://graphics.cs.cmu.edu/projects/behavior_planning/`
[5] `http://udn.epicgames.com/Three/AIAndNavigationHome.html`

action-context pair can also contain tolerances, for example the landing position could potentially have +-20cm deviation and the obstacle width could have -+10cm deviation. There are many ways to parametrize the animation to fit into that space, and it also gives more leeway for the artist and level designers to build levels and also improves the cases where the sampling will work.

Once the production team has a contract like the action-context pair, it is possible to build a rather generic system where AI annotation can be automatically detected from the environment. In case of jumping, it is a matter of finding locations and free trajectories, but for example in case of cover locations, you might want to check for solid walls, too. Usually AI programmers and technical artists work together with game designers to build the building blocks for such tests.

Sara Budde: As I understand, the different professions within a game development context have to agree on everything done with world annotation, so it can be properly integrated into the final game. When the world annotation becomes too complex, this work effort gets too heavy for an individual title to be profitable. This leaves me with the question, why the current AI Engines show no sign of development towards more complex world annotation.

Mikko Mononen: This is a great question! :-) From technical standpoint the problem is that all the technologies need to play together and there are very few common nominators. The common nominator from navigation system is some kind of jump link, a link in the A* graph that has some extra properties.

In order to create good middleware product in that space, you need to combine AI, animation and physics. Havok gets pretty close (`http://www.havok.com/products/ai`). In addition to that you will need to fit that system into a game studio's workflow. The huge problem there is that all

middleware is one generation behind the state of the art in animation (and maybe AI), and many games have very specific needs, in which case you will need to add custom code into the pipeline and that is often hard for middleware providers.

Sara Budde: Thank you very much for this interview Mr. Mononen, and the insights it has provided.

Statement of authorship

I declare that I completed this thesis on my own and that information which has been directly or indirectly taken from other sources has been noted as such. Neither this nor a similar work has been presented to an examination committee.

Berlin, March 24, 2013 .....