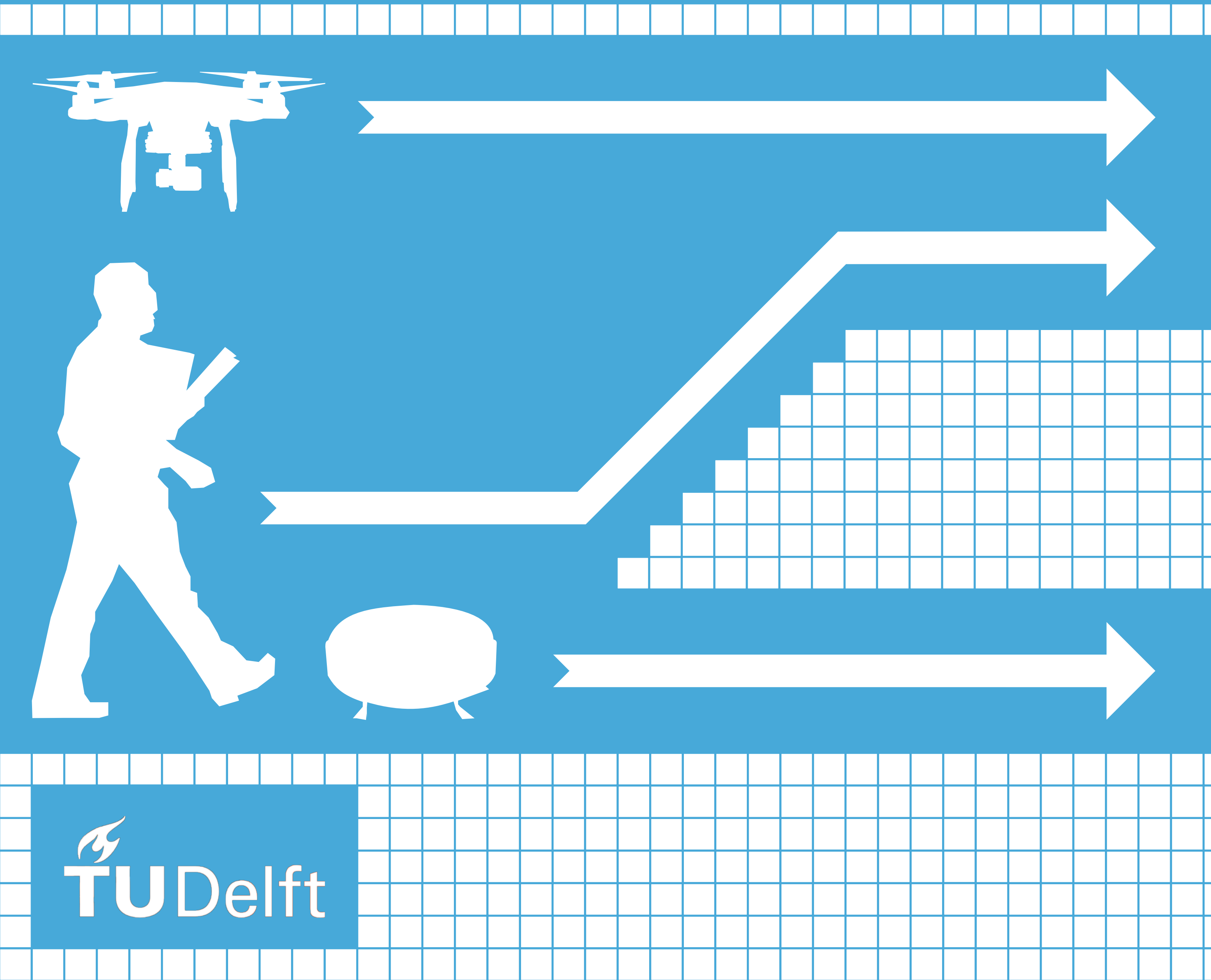
MSc thesis in Geomatics

# 3D Path-finding in a voxelized model of an indoor environment

Martijn Koopman

2016



# 3D PATH-FINDING IN A VOXELIZED MODEL OF AN INDOOR ENVIRONMENT

A thesis submitted to the Delft University of Technology in partial fulfillment of the requirements for the degree of

Master of Science in Geomatics

by

Martijn Koopman

November 2016

Martijn Koopman: *3D Path-finding in a voxelized model of an indoor environment* (2016)

| | |
|---|---|
| Supervisors: | dr. ing. Sisi Zlatanova |
| | dr. ir. Ben Gorte |
| | dipl.-ing. Ulf Hackauf |
| Co-reader: | ir. Edward Verbree |

# ABSTRACT

Indoor environments can be complex and need to be represented by a 3D representation. This is especially important for 3D path-finding. 3D features of the environment can have an effect on the computed path and these features can not be represented correctly in a 2D representation.

A voxelized model is a good 3D representation that supports path-finding. It is a geometrical and topological model and it is easy to incorporate semantics. It is also fairly easy to consider the shape and size of the actor.

This thesis presents a new path-finding method that operates on a voxelized model and supports different kinds of actors. Distinction is made between actors by their size and mode of locomotion. Supported modes of locomotion are walking, driving and flying. A hierarchical data structure is used to reduce the time complexity of the path-finding problem. This hierarchical data structure is a graph which is derived from a cell decomposition of space.

The results indicate that the path-finding method in its current state operates well for walking and driving actors, but further improvements are required for flying actors.

# ACKNOWLEDGEMENTS

I would like to thank everyone directly or indirectly involved in this thesis research. Special thanks goes to my mentors dr. ing. Sisi Zlatanova and dr. ir. Ben Gorte for their feedback and ideas. I would also like to thank my co-reader ir. Edward Verbree for his feedback along with all other participants of the indoor lab. The indoor lab offered a good environment with people working on related topics.

Apart from this research, I also want to thank all students and teachers with whom I worked the last few years. It was fun and I learned a lot.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# ACRONYMS

| | | |
|---|---|---|
| **CDT** | Chamfer distance transformation | 8 |
| **CPG** | cell-and-portal graph | 11 |
| **VTK** | Visualization Toolkit | 29 |

# 1 INTRODUCTION

## 1.1 MOTIVATION

Path-finding is the process of finding a collision free path between two locations. Within the domain of geomatics is path-finding mostly used for navigation, but it can also be used for other applications like emergency evacuation simulation [Meijers et al., 2005] and building performance measurements [Goldstein et al., 2014].

To date, most research on path-finding has been conducted on 2D representations of an environment. These 2D representations are sufficient for most outdoor applications, but indoor environments require a 3D representation in some situations. Indoor environments can be complex and can have 3D features that are of importance for path-finding. For example, an actor that moves from one location to another has a certain height that has to be considered. Next to that, some actors are not bound to a ground surface (e.g. flying) and can move over or underneath obstacles. Also split level floors or hanging platforms are 3D features. These situations require a truly 3D representation of the environment.

There are multiple ways to represent a 3D indoor environment. One of them is the voxelized model. A voxelized model is a three-dimensional uniform rectilinear grid in which each grid element indicates the occupancy of an object within that space. This grid spans the entire model and has a certain granularity (i.e. resolution). One grid element in a voxelized model is called a *voxel* which is short for 'volumetric pixel'.

This type of representation has at least three advantages for 3D path-finding. First of all, it is simple. The model is just a 3D array in which each voxel is indexed by its X, Y and Z coordinates. Secondly, based on Poincaré duality the grid can be thought of as a special graph with rectilinear connections. This inherent topology makes it possible to do path-finding directly on the model without deriving a topological model first. Thirdly, each voxel can be associated with one or more attributes to incorporate semantics.

A voxelized model has one big disadvantage with respect to path-finding. That is the large number of voxels that has to be enumerated for path computation. This number depends on two characteristics of the model: the spatial extent and the resolution. For example, consider a building of 100x50x40 meters with a resolution of 20 centimetres. This means the model contains 25 million voxels. To be able to perform (near) real-time path-finding in such a model it is necessary to consider optimizations like hierarchical path-finding. Near real-time path-finding is often required because an actor will hold in many applications as long as the path is not supplied.

Path-finding can be applicable for human actors like pedestrians or people in wheelchairs, but also for non-human actors like flying drones or domotica robots. Human actors can usually find the way on their own, but non-human actors require a detailed path to navigate. These non-human actors are now emerging and will probably become more and more prominent in the future. What distinguishes these different types of actors are

| Actor | Dimensions (w x h) (cm) | Mode of locomotion | Notion of best path |
|---|---|---|---|
| **Walking adult** | 50 x 190 | Walking | Shortest; Least turns |
| **Person in wheelchair** | 70 x 150 | Driving | Shortest |
| **Rotary wing drone** | 40 x 20 | Flying | Shortest; Fixed elevation |
| **Vacuum cleaner robot** | 35 x 10 | Driving | Hamiltonian |

Table 1.1: Example actors

their size, mode of locomotion and notion of the best path. Several example actors are defined in Table 1.1.

The existence of research literature on 3D indoor path-finding is limited. As stated in [Zlatanova et al., 2014], more research should be conducted on path-finding in 3D regular grids for human applications as well as for indoor robots. Therefore new approaches should be investigated or existing approaches should be combined.

## 1.2 OBJECTIVES

The objective of this thesis research is to develop a path-finding method that is applicable for different kinds of actors. This path-finding method has to operate on a voxelized model which is a geometrical, topological and semantical representation of an indoor environment. Next to that, the path-finding method should be fast enough to perform this path-finding real-time or near real-time.

The following main research question has been formulated based on this research objective:

- Is it possible to develop a single, uniform path-finding method that is applicable for different kind of actors?

To answer this research question, the following sub questions have to be answered:

- What kind of actors exist in an indoor environment?
- What requirements does each kind of actor have on the computed path?
- What parameters can describe the required path of an actor?
- In what data structure should the voxels be stored to facilitate path-finding?
- What is the influence of the model's resolution on the path-finding?
- What implementations could improve the performance of the path-finding method?

## 1.3 SCOPE OF THIS THESIS

This research focuses on 3D indoor path-finding. The path-finding method accepts a voxelized model as is input. This voxelized model must be a

Cartesian grid in which each voxel has the shape of a cube. Other types of 3D representations (e.g., boundary representation) are not considered. The path-finding method derives a model from the input model which is used for path-finding. This model is static (i.e., can not be altered when the indoor environment changes) and has to be generated for each distinct kind of actor.

## 1.4 OUTLINE OF THIS THESIS

This thesis has 6 chapters. The next chapter (Chapter 2) describes the theoretical background and presents related work. Chapter 3 describes the methodology of the newly developed path-finding method. The implementation of this methodology is described in Chapter 4. The path-finding method has been tested on two types of actors: a walking person and a flying drone. The results of these tests are presented in Chapter 5 along with an analysis of the results. The research questions are answered in Chapter 6 followed by the conclusion. A discussion and recommendations for future work are also presented in Chapter 6.

# 2 THEORATICAL BACKGROUND & RELATED WORK

## 2.1 THEORATICAL BACKGROUND

### 2.1.1 Grid or graph

A path-finding algorithm requires a topological model. It defines relations between spaces and determines what steps an actor can make when it moves. In general, two types of topological models are used: a grid or a graph. A voxelized model is a grid and can therefore be considered a geometrical and topological model at the same time. Each voxels has neighbours that determine the connectivity within the model. Other 3D representations like boundary representation (b-rep) and constructive solid geometry (CSG) are merely geometrical models and a graph must be derived from these models to enable path-finding. Deriving a graph comprises two steps: First, a set of nodes must be defined that represent locations in space or cells of space. Secondly, connectivity between these nodes must be determined. The exact methodology for these two steps is dependent on the application and the type of graph that is required.

### 2.1.2 Grid connectivity

A voxel is a single element in a voxelized model. It can have various shapes depending on the voxelized model. In a Cartesian grid – which is a rectilinear grid – has each element the shape of a cube.

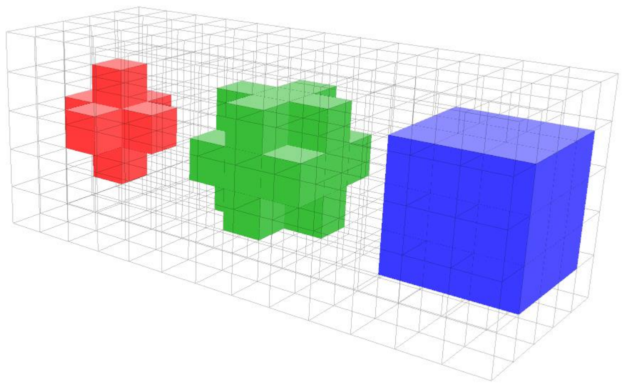


**Figure 2.1:** Voxel connectivity. From left to right: face-connectivity (6); face and edge-connectivity (18); face, edge and vertex-connectivity (26)

The connectivity between a voxel and its neighbouring voxels can be defined by the faces, edges and vertices of the cube. Face-connectivity implies that the voxel is connected to neighbouring voxels with whom it shares a face. Edge-connectivity and vertex-connectivity respectively imply the same for edges and vertices. Usually these types of connectivity are combined as depicted in Figure 2.1. The chosen connectivity determines the steps that an actor can make when it moves through the grid: along the grid, diagonal while staying inside the same horizontal/vertical plane or freely to any of the 26 neighbours.

#### 2.1.3 Static or dynamic model

A model in which a path has to be computed can be static or dynamic. A static model does not change over time whereas a dynamic model can change over time. A dynamic model is required for any path-finding application in a changing environment. One example of such an application is navigation during an emergency evacuation. Due to a disaster, like a fire or flooding, parts of the indoor environment can become inaccessible over time. This requires a dynamic model that can be quickly updated when such an event occurs. The updated model can then be used to quickly re-compute the path of the actor. The actor can then continue its way to its destination.

#### 2.1.4 Single actor or multi actor

Finding a path between two points can be done for a single actor or for multiple actors. Finding a path for a single actor often only requires consideration of the static environment. Finding a path for multiple actors also requires consideration of each actor's volume because actors should not collide with each other. Next to that, the methodology used for finding a path for one actor is not efficient enough to do for hundreds of actors simultaneously. This would require too much computational power and a (near) real-time solution would not be feasible. Finding a path for multiple agents therefore requires a different approach.

## 2.2 RELATED WORK

### 2.2.1 Shortest path algorithms

Shortest path algorithms lie at the basis of path-finding. It refers to finding the shortest path between two endpoints while avoiding obstacles. Over the past decades various search algorithms have been proposed such as Dijkstra's algorithm, breadth first search, depth first search and A* algorithm. Although these algorithms (most notably A*) are quite efficient, more efficient solutions are still required to solve path-finding in complex environments with limited time and resources [Cui and Shi, 2011].

*Bushfire algorithm*

A well-known shortest path algorithm is the bushfire algorithm [Latombe, 1990]. This algorithm works in two stages. In the first stage a navigation front is generated and in the second stage this front is traversed from start to goal. The navigation front is generated by starting an expansion from the starting point. Each voxel in the navigation front has an enumeration level that indicates the number of expansion iterations from the start. Neighbouring voxels of the navigation front are appended to the front and are assigned the current enumeration level + 1. In the resulting navigation front voxels nearby the start have a low enumeration level and voxels further away a high enumeration level. A path is then found by traversing the navigation front from ending point to starting point by following the steepest descent in enumeration levels.

The bushfire algorithm is equivalent to breadth first search and Dijkstra's algorithm [Dijkstra, 1959] which are graph searches. It guarantees to give the shortest path, but it is not very efficient because it enumerates a large number of voxels before reaching the ending point.

#### *Greedy best-first search*

Another shortest path algorithm is the greedy best-first algorithm [Doran and Michie, 1966]. Opposed to the bushfire algorithm this algorithm does not keep track of the distance to the starting point, but to the ending point. It relies on heuristic to expand the navigation front in the direction of the ending point. The heuristic function estimates the cost from the current position to the ending position and may, for example, be the Euclidean distance or Manhattan distance.

This greedy best-first algorithm is faster than the bushfire algorithm because it reduces the number of voxels to enumerate, but it does not guarantee to give the shortest path.

#### *A* algorithm*

The A* algorithm [Hart et al., 1968] combines the principles of the bushfire algorithm and greedy-best first search. It keeps track of the distance to the starting point like the bushfire algorithm and introduces a heuristic like greedy-best first search. It hereby significantly improves the computational efficiency and yet guarantees to give the shortest path. When expanding the navigation front it uses the following function to prioritize the voxels.

$$f(n) = g(n) + h(n)$$

In which $g(n)$ represents the exact cost from starting point to $n$ and $h(n)$ represents the estimated cost from $n$ to the ending point. Voxels with the lowest $f$ value are expanded first.

According to [Hart et al., 1968] A* is guaranteed to find the optimal path if $h(n)$ is less than or equal to the actual shortest path cost, but according to [Rabin, 2000] a slightly overestimated cost usually results in a faster search with a reasonable path.

Although A* is an optimal shortest path algorithm, it still requires optimizations for complex environments. For example, in a voxelized model of 500x500x500 voxels, there is still a search space of 125 million voxels. Optimizations can be accomplished in various ways. First of all, the heuristic function could be altered. If more information about the (dynamic) environment is known, then a better estimation of the cost to the ending point could be made. Secondly, the memory consumption of A* could be lowered. A* requires a certain amount of memory to store the data structure that is involved with the path-finding. In Iterative Deepening A* [Korf, 1985] the whole path is computed in smaller pieces to reduce the memory load. Thirdly, the search space could be reduced by introducing a hierarchical data structure in which path-finding is performed from a higher level to a lower level. See Section 2.2.3 for hierarchical path-finding.

### *Distance transformation*

Another important method for shortest path computation is based on a distance transformation. A distance transformation produces a field in which each point indicates the distance from that point to one or more other points. These other points can be on the boundary of objects, but it can also be the ending point of a path. The produced field is a discretization of space, i.e., a raster. A distance transformation is a typical raster operation while the previously described algorithms involve graph operations.

There are many implementations for a distance transformation. In [Grevera, 2007] and [Jones et al., 2006] a good overview is given. Each implementation has its own characterizations like memory consumption, computation time, dimensionality and accuracy. One commonly applied implementation is the Chamfer distance transformation (CDT) [Borgefors, 1986]. This implementation approximates the Euclidean distance by shifting a mask over the input model similar to a convolution filter in image processing. The distance value is calculated by adding up the value of the previous position with the corresponding value in the mask. If this value is less than the value on the current position, then the current position is replaced by this newly computed value. Only two passes of the filter in opposite direction are required to compute the distance field. The distance values are propagated recursively through the model by the mask. CDT can be applied on an image with an arbitrary number of dimensions. Figure 2.2 shows the algorithm on a one-dimensional image.

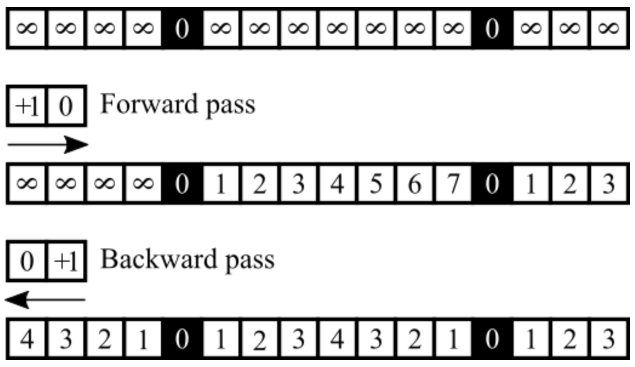


**Figure 2.2:** Chamfer distance transform

As introduced by [Dorst and Verbeek, 1986] CDT can also be used to find the shortest path in an n-D image. A navigation front can be generated by computing the distance field in which each location indicates the distance to the ending point. The computation of such a distance field requires more than two passes. The number of passes depends on the complexity of the indoor environment because the mask should be iterated over the image until no more changes occur.

### *Configuration space*

Configuration space is another important aspect for path-finding. Configuration space finds it origin in the domain of robotics and is also called robot state space. Configuration space comprises all the possible configurations of an actor (position and orientation) in a higher dimensional model. Simply put, each degree of freedom of the actor becomes a dimension in configuration space. The problem of path-finding – including rotational movement of the actor – is thereby reduced to finding a collision-free path in configuration space. This makes it possible to consider the dimensions of the actor and its orientation in the path-finding process.

Finding the shortest path in configuration space requires a path-finding method that is applicable for an arbitrary number of dimensions. In [Verbeek et al., 1986] the CDT is used to compute a collision-free path in configuration space.

The problem of configuration space is the shortest path algorithm's complexity. As stated by [Canny., 1988], the complexity of such a shortest path algorithm is exponential to the number of degrees of freedom of the moving actor. This makes the configuration space approach too slow for (near) real-time path-finding.

### 2.2.2 Graph generation

Path-finding can be performed directly on a voxelized model, but often a graph is derived for this purpose. This choice is based on the requirements of the path-finding method. For example, the fine granularity of the voxelized model may not be desired or ground patches should be considered only instead of the whole volumetric space.

Graph generation is a two-step process. First, regions of space have to be identified that act as nodes in the graph. Secondly, connectivity between these regions has to be found and represented as edges in the graph. In this process the geometrical or semantical properties of the regions can be assigned to the nodes and edges of the graph. This makes it possible to take these properties into account when path-finding is performed.

#### *Pillar based*

There are multiple ways to form regions of space. One of them is by creating pillars of empty space in the horizontal plane as is done by [Xiong et al., 2015] and [Yuan and Schneider, 2010]. These pillars all have the same width and length, but may vary in height. Each pillar is also assigned a type like *floor*, *obstacle* or *stairs*. Regions of space are then formed by grouping pillars together. Pillars can be grouped together if they are adjacent, have the same type and are on the same elevation.

The pillar model is easily derived from a voxelized model, but it is hard to derive this model from a vector model. The methodology of [Xiong et al., 2015] starts with a vector model, but the first step is voxelization to enable this pillar construction.

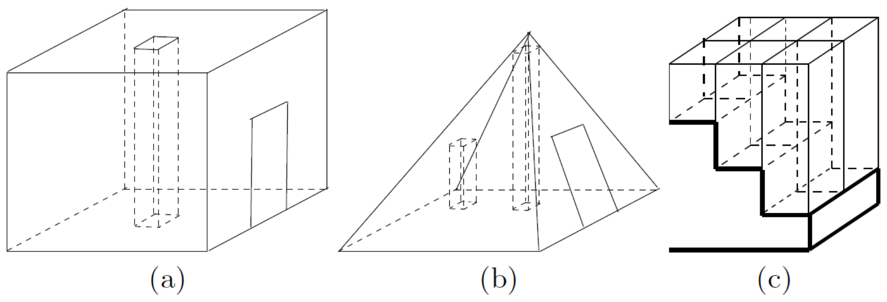


**Figure 2.3:** Subdivision of space in pillars as described in [Yuan and Schneider, 2010]. A pillar in a cell with regular shape (a), pillars in a pyramid shaped cell (b) and pillars representing stairs (c).

In [Xiong et al., 2015] the pillars are grouped together by using a watershed algorithm. In this algorithm regions are created at the centers of empty spaces and then expanded outwards until they reach the borders. A navigation mesh (graph) is then generated by applying a triangulation on the footprints of these regions. This process is depicted in Figure 2.4.

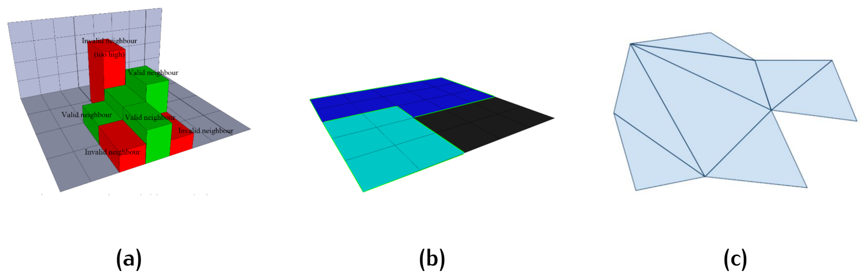


Figure 2.4: Graph generation as descibe in [Xiong et al., 2015]. Space subdivision in pilars (a), contours of region footprints (b) and final triangulation (c).

In [Yuan and Schneider, 2010] the pillars are grouped together by merging them in the horizontal directions as is depicted in Figure 2.5 A. This is an elaborate process and the final outcome heavily depends on the orientation of the model because it can not handle diagonal geometries very well.

The resulting regions of the merging process may be touching or overlapping with each other. That means they are connected and should be denoted by an edge in the graph. The width and height of the shared face between two regions can be assigned as attributes to the corresponding edge. This makes it possible to state the maximum actor size in the graph.

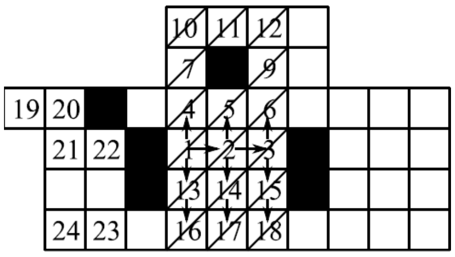


Figure 2.5: Pillar merging as descibed in [Yuan and Schneider, 2010] to generate larger blocks.

## *Distance field based*

Another method for region forming relies on a distance field that can be computed by applying a distance transformation on a voxelized model like CDT. In such a distance field every location marks the distance to the nearest geometry. From this distance field cells of empty space can be derived in multiple ways.

In [Vandapel et al., 2005] spheres are created at local maxima in the distance field. The radii of these spheres are equal to the distance values at the center of these spheres. The resulting spheres are completely empty and form a skeleton in the empty space as depicted in Figure 2.6. Each sphere represents a node in the graph and the intersection of two spheres represents an edge in the graph. The intersection of two spheres is a circle and the radius of this circle is assigned to the edge as an attribute. In this way the size of the actor can be considered, but only one value (the radius) may not be accurate enough to represent an actor's dimensions.

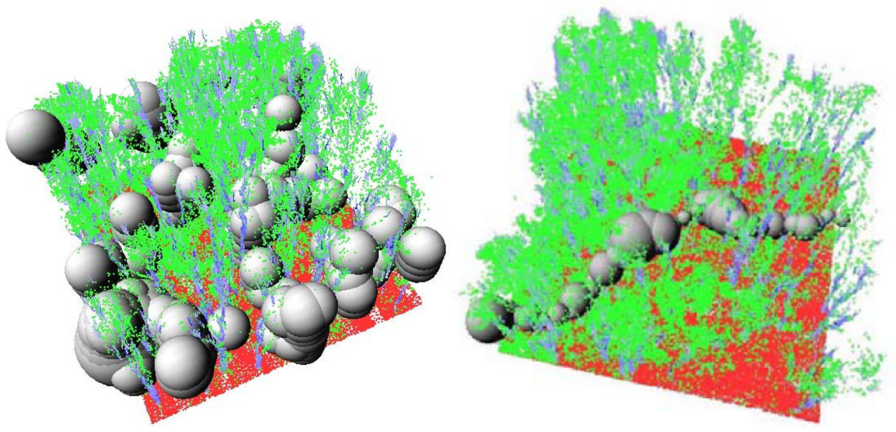


Figure 2.6: Subdivision of space in spheres as described in [Vandapel et al., 2005]. A subset of all spheres (a) and a subset of spheres constituting a path (b).

CELL-AND-PORTAL GRAPH In [Andújar et al., 2004] cells (regions) are created by applying a watershed transformation on the distance field. These cells are then used to construct a cell-and-portal graph (CPG) in which cells correspond to rooms and portals correspond to openings (e.g. doors and windows). An implementation of CPG generation is described in [Haumont et al., 2003]. Generation of a CPG is described here step by step.

1. Distance transformation

CPG generation involves 5 steps. The first step is a distance transformation. In both [Andújar et al., 2004] and [Haumont et al., 2003] CDT is used for this purpose because it is relatively fast and the approximation of the Euclidean distance is accurate enough if a 5x5x5 filter is used.

2. Cell generation

The second step is cell generation. In this step a watershed algorithm is used to segment the distance field into cells as is depicted in Figure 2.7. This algorithm iterates over the distance values in descending order and then performs two steps for each iteration:

1. Existing cells are expanded by appending adjacent voxels with a distance value equal to that of the iteration.
2. New cells are created by grouping all remaining voxels with a distance value equal to that of the iteration.

At the end of the iteration all voxels with a distance value greater than 0 will be assigned to a cell.

One problem with this approach is that furniture and other objects can have a strong influence on the distance field. They can cause many local maxima in the distance field which may lead to over-segmentation of the cell decomposition. The existence of these local maxima also depends on the resolution of the model. A high resolution model is prone to have more

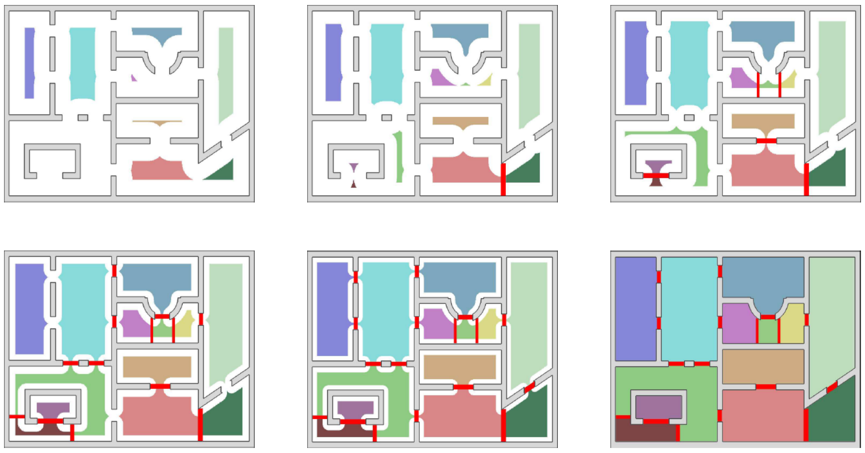


Figure 2.7: Watershed transformation

local maxima in the distance field compared to a lower resolution model. To resolve this problem there are two options. The first option is to simplify the scene by manually removing the objects. This approach provides a good result, but the manual elimination can become a tedious task. The second option is to incorporate a tolerance value in the cell growing process that decreases as the cell grows. As a result small variations in the distance field near the cell origin do not impact the cell generation. However, it is unclear what this tolerance should be exactly and how fast it should decrease over the enumeration.

3. Cell merging

The third step is cell merging. Cell merging improves the cell decomposition by merging uninterested cells. There are a number of rules that can be applied in this merging process. For example, in [Andújar et al., 2004] cells are merged if they do not meet a required minimum size. Next to that, cells are merged if the sum of shared faces with another cell exceeds a given maximum size.

4. Portal detection

The fourth step is portal detection. Portal detection comprises finding shared faces between cells to form portals. Shared faces are found by iterating over all cell voxels and comparing the cell ID of the current voxel with neighboring voxels. Once all shared faces have been found they can be assigned to a portal. If a pin point position of the portal is needed, then there are two possible methods for determining this position. The first method is to take the average XYZ-positon of the shared faces and snap this position to the nearest face of the portal. The second method is to take the XYZ-position in the portal with the maximum distance value.

5. Graph generation

The fifth step is graph generation. In this step a graph node is defined for each cell. These nodes are connected to each other if the corresponding cells are connected by a portal.

### 2.2.3 Hierarchical path-finding

The computational effort of path-finding increases with the search space. This makes path-finding in (near) real-time almost impossible for large models. Consider a model of 500 x 250 x 200 voxels. A shortest path algorithm such as A* could give us the shortest path, but it would require a fair amount of time because CPU resources and memory are limited. This problem can be solved by introducing abstraction. The complexity of the path-finding problem is then reduced by using a hierarchical approach.

Consider that a shortest path has to be found from one room to another room in a large building. Given a space decomposition like a grid, an A* implementation could give the optimal route. This may be an expensive computation given the sheer number of voxels in the model. The complexity of this computation can be reduced by solving the problem in three steps. First, finding the route from the starting point to the door that connects to the hallway. Second, finding the route through the hallway to the door of the other room. Third, finding the route from the door to the ending point. The first and third step require a fine grid while the second step only requires an abstract map. This hierarchical search is not guaranteed to find the shortest path, but it is much faster.

In [Botea et al., 2004] a method for hierarchical path-finding on grid-based maps is presented. This method is called Hierarchical Path-finding A* (HPA*) and creates an abstraction by subdividing a map into linked local clusters. It uses a domain independent way for abstraction and therefore supports multiple abstraction levels. However, this method only works for two-dimensional grids and the methodology for subdividing can not be extended to 3D.

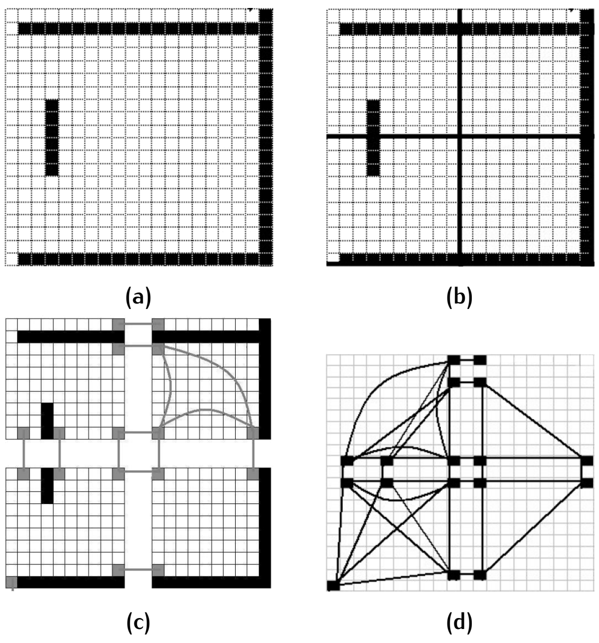


Figure 2.8: Topological abstraction of grid. Grid (a), clusters (b), transitions between clusters (c) and abstracted graph (d).

*Pre-processing a grid*

The first step of HPA* is to define a topological abstraction of the grid. This grid abstraction is then used to build an abstracted graph. The topological abstraction is created by forming clusters. Figure 2.8a and Figure 2.8b show a 20 x 20 grid that is grouped into 4 clusters of size 10 x 10. For each border line between two adjacent clusters, a set of entrances is identified that connects them. An entrance is a line of empty grid tiles along the common border. In Figure 2.8c the two clusters on the left side are connected by two entrances with width 3 and 6. For each entrance, one or two transitions are defined, depending on the width. If the width is less than a predefined constant (6 in this example), then one transition is defined at the center of the entrance. Otherwise two transitions are defined at the borders of the entrance. A transition is defined as two points that both reside in a different cluster while they remain adjacent to each other.

The transitions are used to build an abstract graph as depicted in Figure 2.8d. For each transition two nodes are defined and connected by an edge. This edge (inter-edge) has a length of 1. Within one cluster all transition nodes are connected to each other by edges (intra-edges). The lengths of these edges are computed by finding the shortest paths between the nodes in that cluster.

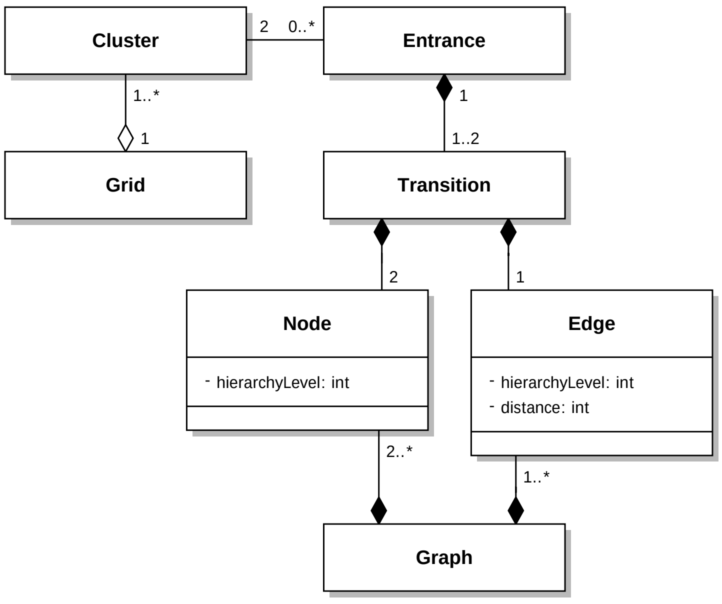


**Figure 2.9:** Classes and relations as defined by [Botea et al., 2004].

*On-line search*

Searching the optimal path in the abstract graph involves 3 steps. First, a new node is created at the starting position and then connected to the graph. Second, a new node is created at the goal position and then connected to the graph. Third, the shortest path is found in the graph using A*.

This search provides an abstract path in which the actor moves from the starting position to the border of that cluster, through the graph to the cluster of the goal position and from the border of that cluster to the goal position.

The following two steps are optional. First, the path can be refined by finding the shortest path within each cluster that is crossed by the abstract path. Secondly, the quality of the path-refinement can be improved by path-smoothening.

### 2.2.4 Space subdivision for indoor navigation

In [Zlatanova et al., 2014] a conceptual framework is presented with formalisation of indoor spaces. It also tries to develop guidelines for automatic partitioning of indoor spaces for various indoor navigation applications. This conceptual framework describes indoor spaces, agents that operate in these spaces and activities that take place in these spaces. This framework presents the following concepts:

INDOOR SPACE Indoor space is an enclosed volume bordered by physical elements. Space is continuous and can have a semantic meaning.

SUB-SPACES Space can be partitioned into sub-spaces. These sub-spaces are non-overlapping, may or may not have semantic meaning and can be navigable or inert (non-navigable).

AGENTS Agents are clients that engage in activities. They are able to move between sub-spaces. Agent are typically human, but can also be robots. The navigable space of an agent depends on its mobility, shape and size. Agents are characterised by their dimension (2D, 3D), individual characteristics (size, age, gender, locomotion mode) and preferences (e.g. shortest path).

# 3 METHODOLOGY

This chapter describes the methodology of the newly developed path-finding method. This methodology combines techniques from three papers. First of all, the same concepts are used as defined in [Zlatanova et al., 2014]. Secondly, the methodology of [Haumont et al., 2003] is used for cell decomposition. Thirdly, the methodology of [Botea et al., 2004] is used for graph generation. Next to that, an algorithm is developed that semi-automatically assigns semantic labels to the empty space.

The conceptual framework of this research is analogue to the conceptual framework of [Zlatanova et al., 2014]. Space is partitioned into semantically labelled sub-spaces (cells) and agents (actors) move through these sub-spaces. As stated by [Zlatanova et al., 2014], the navigable space of an actor depends on its mobility, shape and size. The shape of the actor is represented in this research by a cylinder. The diameter and height of this cylinder define the size of the actor. The mobility of the actor is defined by its mode of locomotion. Distinction is made between three modes of locomotion: driving, walking and flying.

The navigable space of an actor is decomposed into cells (sub-spaces) by using an adapted version of the cell decomposition algorithm of [Haumont et al., 2003]. These cells make it possible to construct a graph later on and enable hierarchical path-finding. The cells are assigned to one of the semantic classes: *floor*, *stairs* or *obstacle*. These classes indicate whether or not the cell is above a floor, stairs or obstacle.

The cell decomposition is used to build a graph by applying the methodology presented in [Botea et al., 2004]. This graph and the cell decomposition are then used to perform path-finding on two levels. First, an initial path is found in the graph. Secondly, the path is refined by finding the shortest path in each individual cell that is visited by the initial path. [Botea et al., 2004] states that finding a path with this methodology is 10 times faster compared to highly-optimized A* and the resulting path is within 1% of the optimal path. However, their methodology operates on a 2D grid and the algorithm that decomposes the grid into clusters is not extendable to 3D. Therefore, it was chosen to use the methodology of [Haumont et al., 2003] to decompose the space into cells. This methodology can fully automatically decompose the space into cells based on the input geometry. An alternative approach for space decomposition is the pillar based approach of [Yuan and Schneider, 2010], but this approach can not handle diagonal geometries very well because of its orthogonal merging algorithm.

Figure 3.1 shows the data flow of the methodology. The operations that are applied on the data (gray blocks) are build up of smaller individual steps. These steps are explained in later sections of this chapter.

The input model in which path-finding has to be performed is a three-dimensional grid. Each voxel in this model has a value. Empty space has value 0 and non-empty space has value 1.

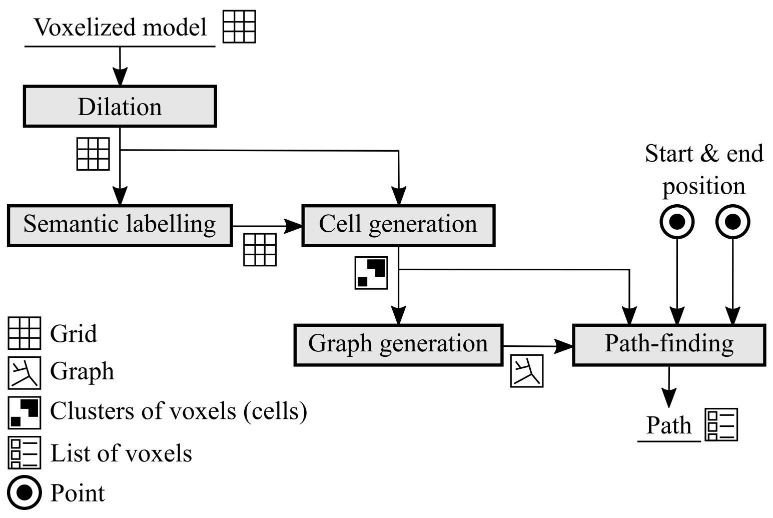


**Figure 3.1:** Data flow

DILATION This input model is dilated to incorporate the size of the actor. That means a horizontal buffer added around the geometry and the geometry is extruded downwards. The navigable space of an actor is hereby formed based on its size. A more detailed explanation of the dilation is given in Section 3.2.

SEMANTIC LABELLING Next step of the methodology is to assign semantic labels to the empty space (voxels with value 0). These semantic labels indicate whether or not the space is above a floor, stairs or obstacle and they are later used to determine the navigable space of an actor. The navigable space of an actor depends on its mode of locomotion because the mode of locomotion determines what spaces are accessible. Table 3.1 shows what semantically labelled spaces are accessible for each mode of locomotion. A more detailed explanation of the semantic labelling is given in Section 3.3.

| | **Semantic class** | | |
|---|---|---|---|
| **Mode of locomotion** | Floor | Stairs | Obstacle |
| Drive | + | - | - |
| Walk | + | + | - |
| Fly | + | + | + |

**Table 3.1:** Accessibility of semantically labelled space based on mode of locomotion.

CELL GENERATION The third step of the methodology is to generate cells from the dilated geometry and semantically labelled space. Each generated cell is a cluster of voxels and each cell belongs to one semantic class. See Section 3.4 for a detailed explanation of the cell generation.

GRAPH GENERATION The fourth step of the methodology is to build a graph based on the cells. This graph is not analogue to the CPG of [Haumont et al., 2003] in which cells are represented by nodes and portals by edges. Instead, it is analogue to the abstract graph of [Botea et al., 2004]. Portals are identical to *transitions* and are represented by two nodes connected by an edge. Either of these nodes resides in a different cell and connectivity with other portals is determined by finding a path within the respective cells. A more detailed explanation of the graph generation is given in Section 3.5.

PATH-FINDING The final step of the methodology is finding the actual path. This step is commonly executed multiple times with different start and end positions defined. The computation time of this step is therefore crucial. The previous steps have to be performed only once for each distinct actor and are therefore less time crucial. See Section 3.6 for a detailed explanation of the path-finding.

The path-finding method requires certain parameters to operate. These parameters describe the characteristics of the actor and are used throughout all steps. These parameters have to be defined before any of the steps can be performed. The input parameter are described in Section 3.1.

The path-finding methodology has been tested on a single model. The applied methodology for this testing is described in Section 3.7.

## 3.1 INPUT PARAMETERS

The following parameters serve as input for the path-finding method. These parameters describe the actor and the environment in which the actor navigates.

- Actor diameter (width & length)
- Actor height
- Mode of locomotion
- Vertical footspan

ACTOR DIAMETER & HEIGHT The actor is a three-dimensional object that can move through space. Its shape is represented by a vertical cylinder and the size of the actor is defined by the cylinder's diameter and height. The diameter of the cylinder corresponds to the width and the length of the actor. The width and length are therefore always equal. The diameter and height of the actor are two parameters for the path-finding methodology.

ACTOR MODE OF LOCOMOTION A distinction is made between three different modes of locomotion: *driving*, *walking* and *flying*. The navigable space of an actor is limited by its mode of locomotion. Driving actors can only navigate over relatively flat ground surfaces, walking actors can also navigate over stairs and flying actors can also navigate over obstacles. The empty space in the model is semantically labelled with those three classes (floor, stairs, obstacle). It is therefore possible to represent the mode of locomotion by three Boolean parameters. Each of these parameters indicate whether or not a semantically labelled space is accessible. Table 3.1 shows what these parameter values should be for each mode of locomotion.

VERTICAL FOOTSPAN Walking actors have legs and are therefore capable of stepping up and down. This enables them to navigate over surfaces with small height differences like a staircase. The vertical footspan is the maximum displacement of an actor in the vertical direction for one step. It is expressed as a number of voxels and should be at least the height of one stair in a staircase.

## 3.2 DILATION

The size of an actor limits its navigable space. Actors can not enter passages which are too narrow or too low. Dilation of the geometry is therefore used to ensure that the actor only accesses spaces that are large enough to hold the actor's size. The dilation adds a horizontal buffer around the geometry and extrudes the geometry downwards. The size of the horizontal buffer and the downward extrusion depend on the size of the actor. The radius of the horizontal buffer should be equal to the radius of the actor and the downward extrusion should be the height of the actor minus 1.

An example can be seen in Figure 3.2. In this example an actor with height 4 and width 3 (orange lines) is placed under a single geometry voxel (gray). A horizontal buffer with radius 1 is added to the geometry and the geometry is extruded downwards by 3 voxels. The remaining empty space is navigable for the actor with respect to its size. The actor can now be considered as if it occupies only a single voxel.

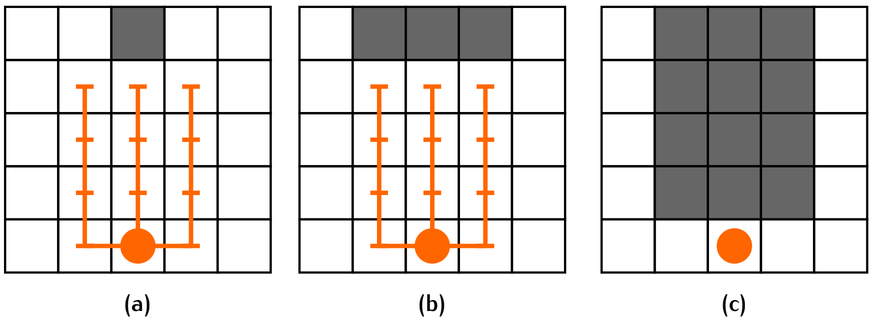


**Figure 3.2:** Dilation. A single non-empty space voxel and an actor (a), horizontal buffer added to voxel (b) and downward extrusion of voxels (c).

This methodology for considering the actor's size has some limitations. First of all, the shape of the actor is always represented as a cylindrical shape and the diameter of this cylinder is always an odd number. Secondly, the entire model has to be updated. Thirdly, the size of the actor can not change during path-finding. On the other hand, this methodology is relatively simple and the size of the actor does not have to be considered during the actual path-finding.

## 3.3 SEMANTIC LABELLING

Semantically enriching the geometrical model with the three classes *floor*, *stairs* and *obstacle* can be done automatically in most cases. The geometrical features of the model are used to make a distinction between floor, stairs and obstacles. The semantic labelling process is depicted in Figure 3.3.

1. EXTRACTION OF HORIZONTAL SURFACES Horizontal surfaces are extracted from the model by selecting all non-empty space voxels that have an empty space voxel above it. Only these voxels are required for the semantic labelling process. See Figure 3.3a.

2. SEGMENTATION OF HORIZONTAL SURFACES The horizontal surfaces are segmented using a flood-fill algorithm. This flood-fill algorithm starts at the

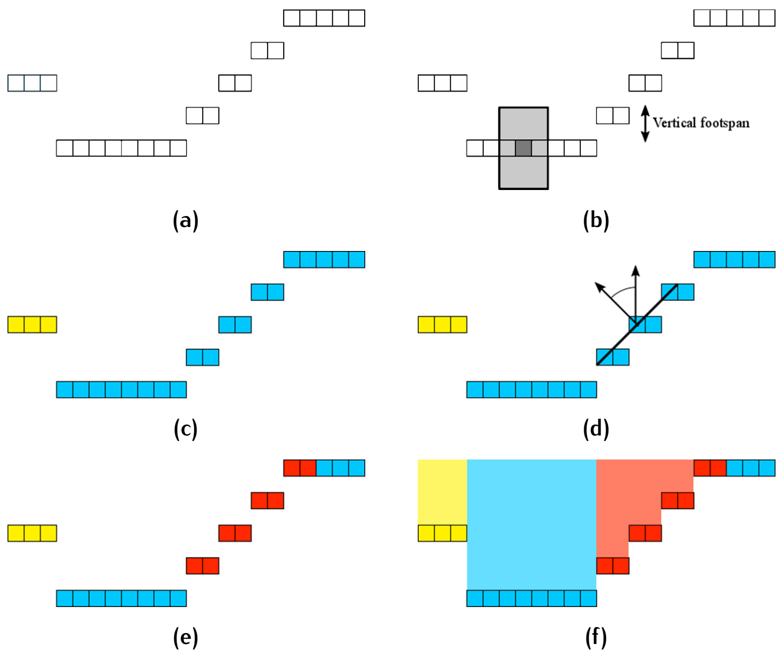


**Figure 3.3:** Semantic labelling. Horizontal surfaces (a), neighbourhood used in segmentation of horizontal surfaces (b), selection of floor segment  obstacle segments (c), slope estimation of floor (d), stairs labelling based on slope (e) and upwards propagation of labels (f).

voxel with the lowest elevation and then expands in all directions. Adjacent voxels are assigned to the same segment whereas disconnected voxels are assigned to new segments. The flood-fill algorithm is capable of expanding upwards and downwards by a given predefined number (vertical footspan). This ensures that the floors and stairs are assigned to the same segment. See Figure 3.3b.

In most situations embodies the first (and largest) segment the floors and stairs. This first segment is labelled *floor* and all other segments are labelled *obstacle*. See Figure 3.3c. The floor segment is colored blue and the obstacle segment(s) yellow.

3. SLOPE ESTIMATION Distinguishing between floor and stairs is done by computing the slope of the surface voxels. The slope is estimated by fitting a plane through the neighbourhood of a voxel. The angle between the normal vector of this plane and a vertical up vector gives the slope. The radius of the neighbourhood is a predefined number and depends on the resolution of the model. See Figure 3.3d.

All voxels with a slope above a given threshold are labelled *stairs*. The other voxels remain labelled as *floor*. See Figure 3.3e. Stairs voxels are colored red.

4. UPWARDS PROPAGATION OF CLASSES All horizontal surfaces are now labelled as one of the three classes: *floor*, *stairs* or *obstacle*. These labels are propagated upwards to also label the empty space above the horizontal surfaces. Propagation of the labels is accomplished by iterating through the voxelized model from bottom to top and assigning the last encountered label to the empty space. See Figure 3.3d.

## 3.4 GELL GENERATION

Figure 3.4 shows the steps involved in the cell generation. Cells are generated by applying a watershed transformation on a distance field of the dilated model. These cells are afterwards merged if they do not meet certain criteria. This methodology originates from [Haumont et al., 2003], but an addition is made to incorporate the semantic classes. This addition ensures that all voxels added to a certain cell belong to the same semantic class.

The resulting cells need to represent parts of the navigable space. Therefore, they need to be compressed for walking and driving actors. The compressed cells have a thickness of one voxel and are located just above horizontal surfaces. This is the space where walking and driving actors can navigate through. The compression is skipped for flying actors.

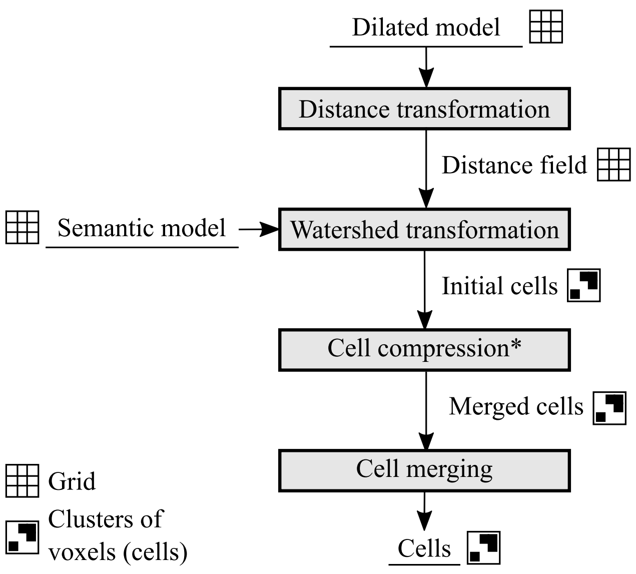


**Figure 3.4:** Cell generation flow

Each step of the cell generation is described here in more detail.

1. DISTANCE TRANSFORMATION The distance field is computed using CDT described in Section 2.2.1. Each location in the distance field indicates the distance to the nearest geometry. The maximum distance values are most often near the centers of rooms, but this can be heavily affected by furniture. The low distance values are close to the floor, ceiling, walls and obstacles.

2. WATERSHED TRANSFORMATION The distance field is segmented into cells using the watershed transformation described in Section 2.2.2. The distance values are iterated in descending order and for each iteration two steps are performed. First, existing cells are expanded by appending adjacent voxels with a distance value equal to that of the iteration. Secondly, new cells are created using a flood-fill algorithm on all remaining voxels with a distance value equal to that of the iteration.

The semantic model is also used in the watershed transformation. Newly created cells receive the same semantic label as the very first voxel of the cell and adjacent voxels may only be appended to the cell if they have the same semantic label.

The entire empty space is decomposed into cells after the watershed transformation and each of these cells has a unique identifier.

3. CELL COMPRESSION All cells together represent the entire empty space. This is the navigable space for a flying actor, but not for walking and driving actors. Walking and driving actors are only capable of navigating through the space just above a ground surface. Therefore they require an additional step to limit the navigable space. This additional step compresses the cells downwards to a thickness of 1 voxel so that only the voxels above a surface remain. The cells are compressed by iterating through the cells from top to bottom and deleting any cell voxels that have another cell voxel underneath.

4. CELL MERGING Cells are required to have a minimum size for the following steps. Portal detection, which is part of the graph generation, requires cells with a minimal size. Cells are therefore merged with adjacent cells if they are too small. The cells are always merged with adjacent cells that have the same type (*floor*, *stairs* or *obstacle*) or another type that is also valid for the actor's mode of locomotion. For example, cells of type *floor* and *stairs* may be merged for a walking actor because they are both accessible for that type of actor. If there are no adjacent cells to merge with, then the cell is deleted.

## 3.5 GRAPH GENERATION

The graph is derived from the generated cells. This involves two steps. First, portals are defined on the boundaries between two cells to imply connectivity. Secondly, graph nodes and edges are defined based on these portals and the connectivity between them.

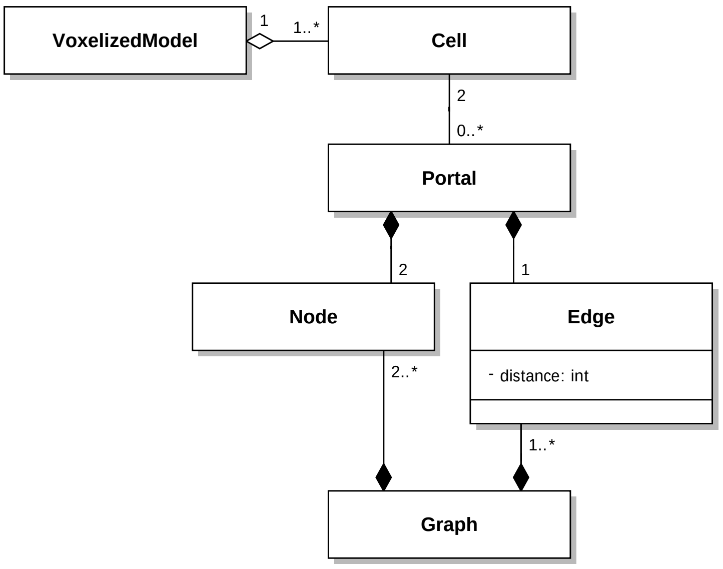


**Figure 3.5:** Defined classes and relations

Figure 3.5 shows the conceptual classes that define the objects in this methodology. A portal is a boundary between two cells. It is the set of voxels that share a face with the voxels of one other cell. A portal has a center point that is used to define the positions of the nodes in a graph. A portal is represented in the graph by two nodes connected by an edge. Both nodes reside in a different cell on either side of the boundary. The edge that connects them has length 1. A portal is identical to a *transition* defined in [Botea et al., 2004] and a cell can be seen as a *cluster*.

The two steps of graph generation are described here in more detail.

1. PORTAL GENERATION Cells are simply clusters of voxels. They do not imply any adjacency or connectivity between each other. Portals are generated on the boundaries between cells to imply this adjacency. A portal is always between two cells and does not necessarily have to be convex.

Portals are detected by finding adjacent voxels with different cell IDs. This combination of two different cell IDs forms a tuple and is unique for every portal. Portals are formed by grouping voxels together with the same tuple of cell IDs.

The centroid has to be computed for each portal. This centroid is computed by taking the mean of all voxels in the portal. The resulting centroid may lie outside the portal if the portal is non-convex. Therefore the centroid is snapped towards the closest portal voxel.

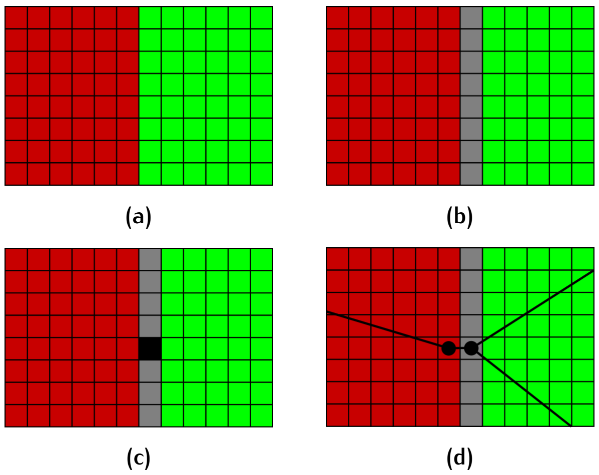


**Figure 3.6:** Portal detection. Two cells (a), voxels of right cell that are part of boundary with left cell (b), mean of bounary voxels (c) and graph (d).

2. GRAPH GENERATION A graph is derived from the cells and portals. In this graph is each portal is represented by two connected vertices. Each of these vertices resides in a different cell and they are connected by an edge of length 1. The connectivity between two portals of the same cell is represented by forming an edge between the corresponding vertices. This edge is assigned a weight that corresponds to the length of the path between the two vertices. This path is computed using A* on the cell voxels.

Each vertex and edge is also assigned the cell ID it resides in. This cell ID is used in the path-finding process.

## 3.6 PATH-FINDING

Path-finding is performed on two levels in consecutive order. First on the graph (level 1) and then on the voxels that constitute a cell (level 2). Path-

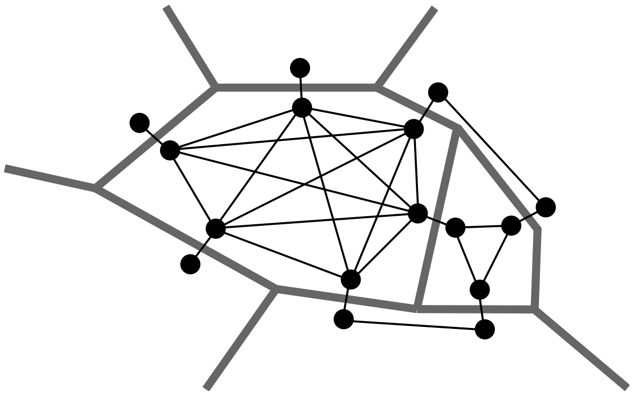


Figure 3.7: Example graph

finding on level 2 is optional and is only required if a fine path is requested. This path-finding process is depicted in Figure 3.8.

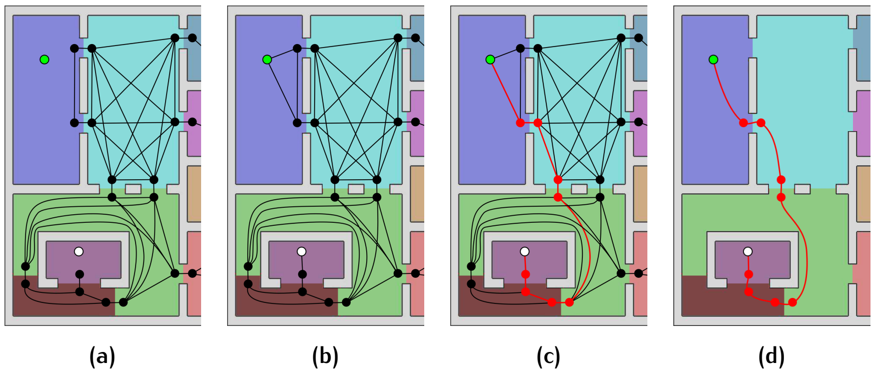


Figure 3.8: Path-finding in graph. Two nodes created at start and end position (a), nodes connected to the graph (b), shortest path in graph (c) and refined path in cells (d).

### 3.6.1 Path-finding in graph

Path-finding in the graph involves 2 steps. First, the start and end point are inserted into the graph as nodes. Second, a shortest path is found in the graph between the two newly inserted nodes.

#### *Insert nodes in graph*

The start and end point are inserted into the graph by creating two new nodes (Figure 3.8a). Each of these nodes resides in a certain cell of the cell decomposition. Connectivity with already existing nodes is determined by finding the shortest path between the newly inserted nodes and already existing nodes in the same cell. If such a path exists, then an edge is created and the length of the path is assigned to this edge (Figure 3.8b).

#### *Shortest path in graph*

The shortest path between the two newly inserted nodes is found by running the A* algorithm (Figure 3.8c). The resulting path is a sequence of nodes that is visited when the path is traversed. Each of these nodes resides in a certain cell and therefore the path can also be seen as a sequence of visited cells.

### 3.6.2 Path-finding in cell

The found path can be refined by finding the exact path between two nodes in the graph (Figure 3.8d). This requires the path-finding method to operate on the voxels that constitute a cell. Which cell this would be can be deducted from the sequence of visited cells determined in the previous step. It is possible to refine the path for all cells at once, but it is also possible to refine the path cell by cell whenever a finer path is requested for that particular cell. In this implementation the former was chosen.

## 3.7 TESTING

In this section the methodology is described for testing the newly develop path-finding method. The results of these tests can be found in Chapter 5

### 3.7.1 Input model

The input model for the test is a 2-storey building with furniture inside. A floorplan of this model can be seen in Figure 3.9. Doors and windows are omitted from the model and are therefore represented by openings or closed surfaces. The ground floor consists of 5 rooms: a hallway, living room, dining room, kitchen and lounge room. The second floor consists of 6 rooms: a hallway, 2 bathrooms and 3 bedrooms. The two floors are connected through a staircase. The height of a step in the staircase is 17 cm. The lounge room on the ground floor is connected to the kitchen through 2 steps with the same height.

Three different versions of this model exist with each a different resolution. These resolutions are 10, 20 and 40 cm.

### 3.7.2 Input actor

Two different actors are defined for testing. These actors are listed in Table 3.2. Each actor has a different mode of locomotion to ensure that the path-finding method is applicable for each of these modes.

| **Actor** | **Dimensions (w x h) (cm)** | **Mode of locomotion** |
|---|---|---|
| **Adult pedestrian** | 50 x 190 | Walking |
| **Rotary wing drone** | 40 x 20 | Flying |

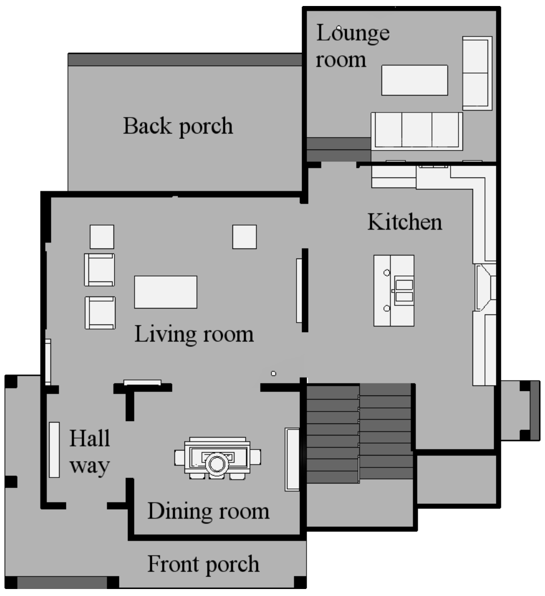
Table 3.2: Input actors

### 3.7.3 Paths

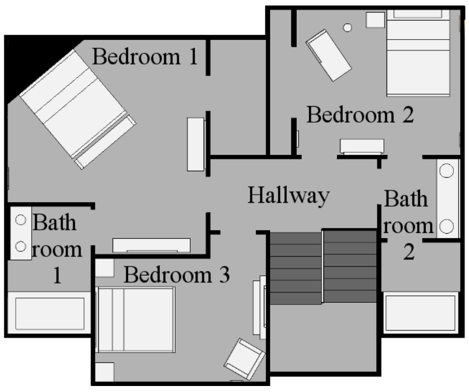
Three paths are computed for each actor. The start and end points of these paths are defined in Table 3.3. These points are located in the centers of the rooms just above the ground surface.

| Path | Start position | End position |
|---|---|---|
| 1 | Lounge room | Bedroom 1 |
| 2 | Bedroom 2 | Living room |
| 3 | Bathroom 1 | Bathroom 2 |

**Table 3.3:** Paths



(a)



(b)

**Figure 3.9:** Input model floorplan. Ground floor (a) and second floor (b).

### 3.7.4 Procedure

All steps described in this chapter are performed to test the developed path-finding method.

1. The input parameters are defined based on the actor and model.
2. The input model is dilated based on the size of the actor.
3. The semantic model is generated from the dilated model.
4. The cell decomposition is computed.

5. The graph is derived from the cell decomposition.

6. The path is found in the graph and then refined in the cells.

To test the effect of the hierarchical approach, it is compared to a non-hierarchical approach. In this non-hierarchical approach the same path is computed using the A* algorithm directly on the voxelized model. These two paths are then compared by their length and computation time.

# 4 IMPLEMENTATION

## 4.1 DEVELOPMENT ENVIRONMENT

The path-finding method is implemented in Python and the application ParaView has been used as development environment. ParaView is an open-source, multi-platform data analysis and visualization application. This application follows a pipeline principle in which data is manipulated by applying one or more filters in consecutive order. The application itself is highly extensible by the use of scripting language Python. New filters can be written in Python and the scene can be manipulated directly through a Python shell.

### 4.1.1 ParaView data types

ParaView is based on the Visualization Toolkit (VTK). VTK is an open-source software system for 3D computer graphics, image processing and visualization. It provides several data formats for 3D data including:

- vtkImageData
- vtkPolyData
- vtkMutableUndirectedGraph

VTKIMAGEDATA vtkImageData is an image data type. It can store data as raster (voxels). Internally the raster data is stored as a one-dimensional array. The elements of this array can be of any type defined by the user like *char*, *integer* or *float*. ParaView is capable of mapping this array as a scalar. That means the rendered transparency and color of a voxel is determined by its value.

VTKPOLYDATA vtkPolyData is a polygonal data type. It can store vector data as vertices, edges and faces. This type of data is used to output the constructed graph.

VTKMUTABLEUNDIRECTEDGRAPH vtkMutableUndirectedGraph is a graph data type. This type of graph is mutable and undirected. That means the graph can be edited programmatically and edges can be traversed in both directions. Internally the graph is stored as three arrays: one array with the vertex indices, one array with the vertex coordinates and one array with the edges. Each vertex index is a positive integer value and each edge is a tuple of two vertex indices. Attributes can be added to the vertices and edges by constructing additional arrays and passing these to the vtkMutableUndirectedGraph object.

## 4.2 DATA MODEL

Figure 3.1 in Chapter 3 shows the data flow in the methodology. There are 4 different types of data being used:

- Grid
- Cell
- Graph
- List

GRID A grid stores all voxels of the model. Its extent has the shape of a cuboid and covers the whole model. Both empty space voxels as well as non-empty space voxels are stored, i.e., it is not a sparse data structure. The grid data type is used for 3 different datasets in the methodology:

| Model | Voxel values |
|---|---|
| Input model<br>Dilated model | 0 (empty space), 1 (non-empty space) |
| Semantic model | Semantic label (See Table 4.3) |
| Distance field | Distance value |

Table 4.1: Grid models

A grid is implemented by the data type vtkImageData described in the previous section.

CELL A cell stores a selection of all voxels. This makes it a sparse data structure and the shape of it can be arbitrary. It is defined by an object definition (class) and has three fields as shown in Figure 4.1. The first field is the unique identifier of the cell, the second field is the type of the cell (semantic class) and the third field is a Python dictionary which contains the voxels that constitute the cell.

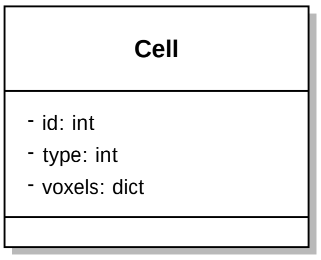


Figure 4.1: Cell object definition

A Python dictionary is an implementation of a hash table. A hash table is a data structure that allows storing values by custom indices (i.e. keys). In such a data structure the tuple (x, y, z) can be used as a key to access a voxel. An advantage of a hash table is that it is able to store only a subset of the total number of voxels. For example, all non-empty space voxels may be stored in a hash table while the empty space voxels are omitted. This can reduce the memory consumption substantially.

Cells are constructed by the cell generation step and then used for the graph generation and path-finding step. All cells together are stored in a hash table and the unique identifier of the cell is used as key. This makes it possible to quickly retrieve a cell by its ID. This is a necessity for the path-refinement step because this step has to find the shortest path through a single cell.

GRAPH A graph is a topological model consisting of nodes and edges. A graph is implemented by the data type vtkMutableUndirectedGraph described in the previous section. A graph is created by the graph generation step.

LIST A list is a sequence of voxels. Each element in the list is a tuple of three indices. One index for each dimension (X, Y, Z).

A portal has no data type definition. It is only a conceptual data type. It is represented in the implementation by two nodes connected by an edge.

## 4.3 PYTHON SCRIPTS

The path-finding method is implemented in ParaView by 14 programmable filters. These filters have to be executed in consequtive order as depicted in Figure 4.2. Each of these filters is described here briefly. More in-depth information about the underlying algorithms can be found in Section 4.4.

The source code has been released under an open license and can be found at `https://github.com/martijnkoopman/thesis`.

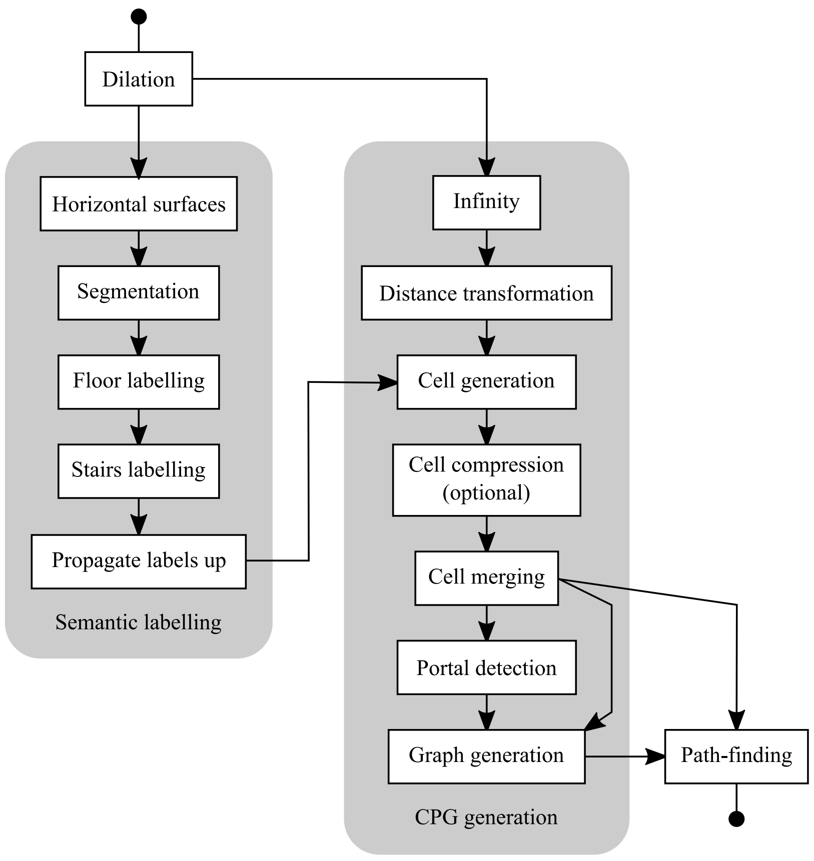


Figure 4.2: Python script execution order

**DILATION** Adds horizontal buffer to the geometry and extrudes the geometry downwards. This script requires two user-defined input parameters: the radius of the horizontal buffer & the amount of downward extrusion.

**HORIZONTAL SURFACES** Extracts all voxels that are part of a horizontal surface.

**SEGMENTATION** Segments all horizontal surface voxels using a flood-fill algorithm. This script requires one user-defined input parameter: the vertical footspan. See Section 4.4.1 for more information on the implementation.

**FLOOR LABELLING** Labels one segment as *floor* and others as *obstacle*. This script requires one user-defined input parameter: the segment number to label as *floor*.

**STAIRS LABELLING** Labels floor voxels as *stairs* based on their slope. The slope is estimated using plane fitting. This script requires two user-defined input parameters: the radius of the neighborhood to fit a plane through & a threshold for the maximum angle between a vertical up vector and the normal vector of the plane.

**PROPAGATE LABELS UP** Labels the empty space above floor, stairs and obstacles by propagating the labels upwards.

**INFINITY** Assigns infinity (a very large number) to empty space voxels and 0 to non-empty space voxels. This configuration is required for the following distance transformation.

**DISTANCE TRANSFORMATION** Computes distance field using CDT. See Section 4.4.2 for more information on the implementation.

**CELL GENERATION** Generates cells from the distance field and semantic labels. See Section 4.4.3 for more information on the implementation.

**CELL COMPRESSION** Compresses cells downwards to a thickness of 1 voxel. This filter is only required for ground surface actors (i.e., walking and driving actors).

**CELL MERGING** Merges or deletes cells if they do not meet certain criteria. This script requires two user-defined input parameters: the allowed semantical classes for cells & the vertical footspan. The allowed semantical classes depend on the actor's mode of locomotion and can be found in Table 4.4. See Section 4.4.4 for more information on the implementation.

**PORTAL DETECTION** Finds boundaries between cells (portals) and assigns an ID to each boundary. This script requires one user-defined input parameter: the allowed semantical classes.

**GRAPH GENERATION** Generates a graph from the cells and portals.

**PATH-FINDING** Finds the shortest path between two points by utilizing the cells and the graph. See Section 4.4.5 for more information on the implementation.

## 4.4 IMPLEMENTED ALGORITHMS

### 4.4.1 Flood-fill algorithm

The horizontal surfaces are segmented using a flood-fill algorithm. This flood-fill algorithm searches for the voxel with the lowest elevation and then adds it to a queue. For each voxel that is removed from the queue the following actions are undertaken:

1. The voxel is assigned the current cell ID.
2. The neighbours of the voxel are added to the queue.

When the queue is empty this means there are no more voxels adjacent to the cell. When this happens the next voxel with the lowest elevation is searched and added to the queue. The current cell ID is then also incremented.

The used connectivity for finding neighbours is *vertical footspan + 4-connectivity*. This means 4-connectivity is used on multiple elevations relative to the current position. The vertical footspan defines this range of elevations. For example, a vertical footspan of 2 means that the neighbours of the 2 voxels above and below the current position have to be checked. This ensures that segmentation can go up the stairs. The same type of connectivity is also used for path-finding in a cell.

### 4.4.2 Distance transformation

The distance transformation implements CDT described in Section 2.2.1. This algorithm shifts a mask over the model and replaces the values of infinity by the calculated distance values. Already calculated distance values can be overridden in the second pass if the outcome of that calculation is less than the distance value calculated in the first pass. Figure 4.3 shows the masks that are used in the implementation.

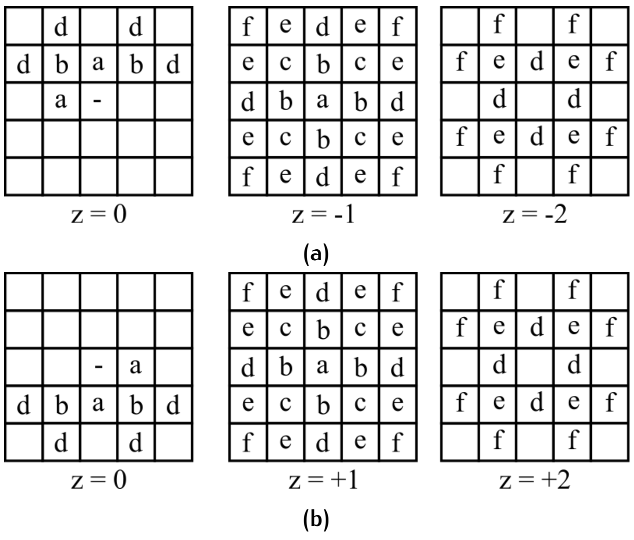


**Figure 4.3:** CDT masks. Forward pass mask (a) and backward pass mask (b).

The numbers *a* to *f* should represent the Euclidean distance from the center of the mask. Table 4.2 shows the calculation of these distances for each mask element.

| Position | Axis 1 | Axis 2 | Axis 3 | Length |
|---|---|---|---|---|
| a | 1 | 0 | 0 | $\sqrt{1^2+0^2+0^2}=\sqrt{1}=1$ |
| b | 1 | 1 | 0 | $\sqrt{1^2+1^2+0^2}=\sqrt{2}=1.41$ |
| c | 1 | 1 | 1 | $\sqrt{1^2+1^2+1^2}=\sqrt{3}=1.73$ |
| d | 2 | 1 | 0 | $\sqrt{2^2+1^2+0^2}=\sqrt{5}=2.24$ |
| e | 2 | 1 | 1 | $\sqrt{2^2+1^2+1^2}=\sqrt{6}=2.45$ |
| f | 2 | 2 | 1 | $\sqrt{2^2+2^2+2^2}=\sqrt{9}=3$ |

**Table 4.2:** Euclidean distances of CDT 5x5x5 mask

The resulting distances are floating point numbers, but integer numbers are preferred because integer arithmetic is computationally less expensive.

Good integer values are: a = 22, b = 31, c = 38, d = 49, e = 54. The average difference between these integer values and the floating point values is only 0.0871 and the standard deviation is 0.0749.

### 4.4.3 Watershed transformation

The cells are generated from the distance field and semantic labels by applying a watershed transformation as described in Section 2.2.2 and Section 3.4.

This algorithm performs a process that is synonym to plunging a perforated surface in an ocean of water. The lower parts of this surface (valleys) flood and start forming basins. These basins grow as the surface sinks deeper and at some points these basins start to merge. To avoid this merging, dams are built. At the end of this process the entire surface is flooded and segmented into basins.

In practice, the elevations in the plunged surface correspond to the inverted distance values in the distance field. The watershed transformation produces a cell decomposition in which each cell is a cluster of empty space. Each cell should have one of the three semantic classes: *floor*, *stairs* or *obstacle*. Therefore, a constraint is added to the watershed transformation. This constraint states that voxels being added to a cell must have the same semantic label as the cell itself. Algorithm 4.1 shows the implementation of the algorithm in pseudocode.

The algorithm iterates all distance values in descending order and then performs two steps:

1. Expand existing cells
2. Create new cells

New cells are created using a similar method as the flood-fill algorithm described in Section 4.4.1. This method utilizes a queue data structure to process the voxels in an ordering that is based on the adjacency.

The semantic labels, used to assign a cell to a semantic class, are just numbers. Table 4.3 shows the semantic labels for each class.

### 4.4.4 Cell merging

Cells are merged if they meet at least one of the following criteria:

1. The cell contains less than 5 voxels.

2. Two of the cell's dimensions (like width and height) are less than 2.

**Algorithm 4.1:** Watershed transformation

```
Data: Distance field $D$ (volume); Semantic labels $S$ (volume)
Result: Cells $C$ (list)
$C = []$
$cellNum = 1$
for $iso$ from Max($D$) to Min($D$) do
    // Expand existing cells
    foreach $cell$ in $C$ do
        foreach $voxel$ in $cell$ do
            foreach $neighbour$ of $voxel$ do
                if $D(neighbour) = iso$ and $S(neighbour) = cell.type$ then
                    $cell.voxels.append(neighbour)$
                    $D(neighbour) = 0$
                    $S(neighbour) = 0$
    // Find new cells
    $startVoxel$ = FindVoxelWithValue($D, iso$)
    while $startVoxel$ do
        $cell$ = Cell($cellNum, startVoxel.type$) ; // Create new cell
        $cellNum$ += 1
        $queue = [startVoxel]$
        while $queue.length > 0$ do
            $voxel = queue.pop()$
            if $D(voxel) = iso$ and $S(voxel) = cell.type$ then
                $cell.voxels.append(voxel)$
                $D(neighbour) = 0$
                $S(neighbour) = 0$
                foreach $neighbour$ of $voxel$ do
                    $queue$ += $neighbour$
        $C$ += $cell$
        $startVoxel$ = FindVoxelWithValue($D, iso$)
```

3. More than 33% of the cell's voxels are part of a boundary with another cell.

A cell is merged with a neighbouring cell that has the same semantic label. If no such neighbour exists, then the cell is merged with a neighbour that has a different semantic label, but this label has to belong to one of the allowed semantic classes. The allowed semantic classes depend on the actors mode of locomotion. Table 4.4 shows the allowed semantic classes for each mode of locomotion. The cell is always merged with the neighbouring cell with whom its shares the most faces.

### 4.4.5 A* path-finding

Path-finding is performed in two steps: First, the shortest path is found in the graph. Secondly, the path is refined. Path-refinement involves finding

| Class | Label |
|---|---|
| Non-navigable space | 0 |
| Floor | 1 |
| Stairs | 2 |
| Obstacle | 3 |

Table 4.3: Labels of semantic classes

| Mode of locomotion | Allowed semantic classes |
|---|---|
| Driving | Floor |
| Walking | Floor & stairs |
| Flying | Floor, stairs & obstacle |

Table 4.4: Allowed semantic classes for each mode of locomotion

the shortest path in one cell between two graph nodes. The shortest path is found in either situation by using the A* algorithm. The implementation of the A* algorithm is an adaption of the source code available at `http://www.redblobgames.com/pathfinding/`.

The implementation uses a *priority queue* to process voxels or graph nodes in a prioritized order. Each element (voxel or node) that is added to the queue has a priority number ($f$). This number is the sum of the traveled distance ($g$) and the estimation of the distance to the end point ($h$). Elements with the lowest priority number are popped from the queue first.

# 5 RESULTS & ANALYSIS

In this chapter the results of the path-finding method are presented. These results have been acquired by applying the methodology described in Chapter 3. This chapter is divided in three sections. The first two sections give the results for two types of actors: a walking person and a flying drone. The third section gives an analysis based on the results for the two actors.

The datasets which were generated by the methodology can be found online at `https://github.com/martijnkoopman/thesis/tree/master/data`

## 5.1 WALKING PERSON

### 5.1.1 Dilation of the model

The walking person has the following dimensions: 50 x 190 cm. Table 5.1 shows the radius of the horizontal buffer and the extent of the downward extrusion for a given resolution.

| Resolution | Horizontal dilation radius (Represented diameter) | Downward extrusion (Represented height) |
|---|---|---|
| 10 | 2 (50 cm) | 18 (190 cm) |
| 20 | 1 (60 cm) | 9 (200 cm) |
| 40 | 0 (40 cm) | 4 (200 cm) |

**Table 5.1:** Dilation parameters for walking person

The higher the resolution, the more accurate the dimensions of the actor can be represented. In the 20 cm and 40 cm resolution model the actor is represented slightly taller than it is supposed to be. Notice that the represented diameter and height are always a multitude of the resolution and the diameter of an actor is always represented by an odd number of voxels.

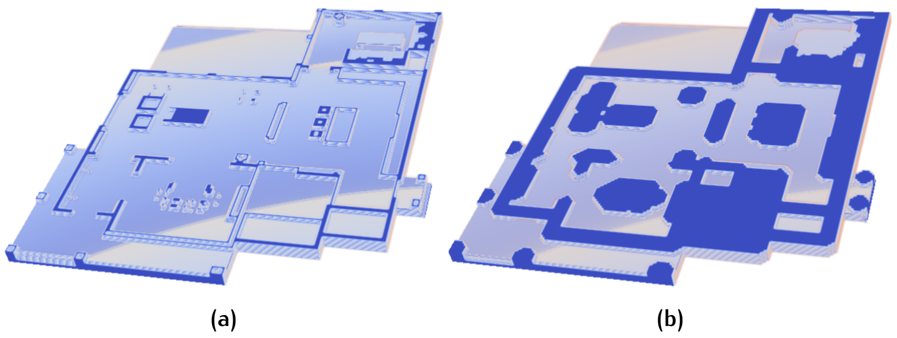

(a) (b)

**Figure 5.1:** Dilation result. Cross section of input geometry (a) and cross section of dilated geometry (b).

Figure 5.1 shows the result of the dilation methodology applied on the 10 cm resolution model.

### 5.1.2 Semantic labelling

The semi-automatic semantic labelling process uses a flood-fill algorithm to select the floor segments and the connecting stairs. This algorithm relies on the vertical footspan which defines the number of voxels that the growing algorithm can go up or down. This ensures the growing algorithm can go up the stairs and assign different floors to the same segment. The vertical footspan is dependent on the resolution of the model and should represent the height of one stair (17 cm). Table 5.2 shows the vertical footspans that have been used during the semantic labelling.

| **Resolution** | **Vertical footspan (Represented height)** |
|---|---|
| 10 | 2 (20 cm) |
| 20 | 1 (20 cm) |
| 40 | 1 (40 cm) |

Table 5.2: Vertical footspans for walking person

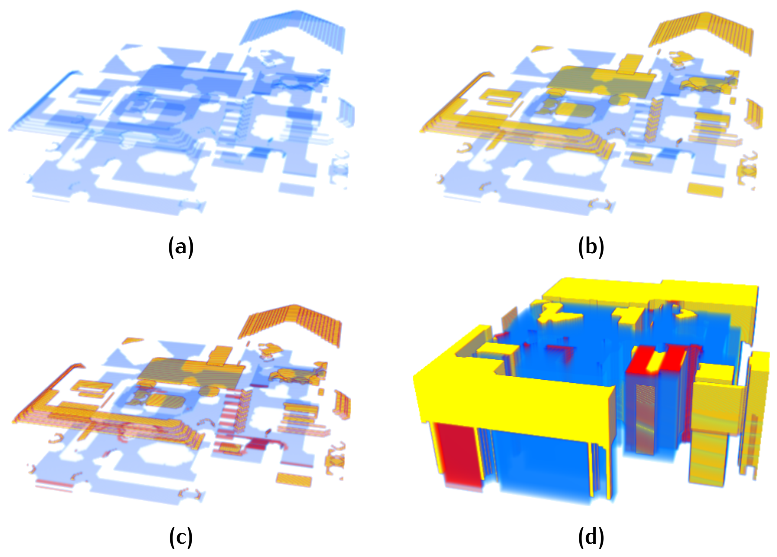


Figure 5.2: Semantic labelling. Horizontal surfaces (a), segmented floor and obstacle surfaces (b), labelled floor, stairs & obstacle surfaces (c) and upwards propagation (d).

Figure 5.2 shows the results of each step on the 10 cm resolution model. All horizontal surfaces present in the dilated model are colored blue in Figure 5.2a. These horizontal surfaces are segmented and one segment is selected as *floor* segment. This is the blue segment in Figure 5.2b. The other segments are labelled as *obstacle* and are colored yellow in the figure. Distinction between floor and stairs is made by estimating the slope of the surfaces. Figure 5.2c shows the voxels labelled as *stairs* in red. The semantic labels are propagated upwards to also label the empty space above the surfaces. Figure 5.2d shows the final semantic model.

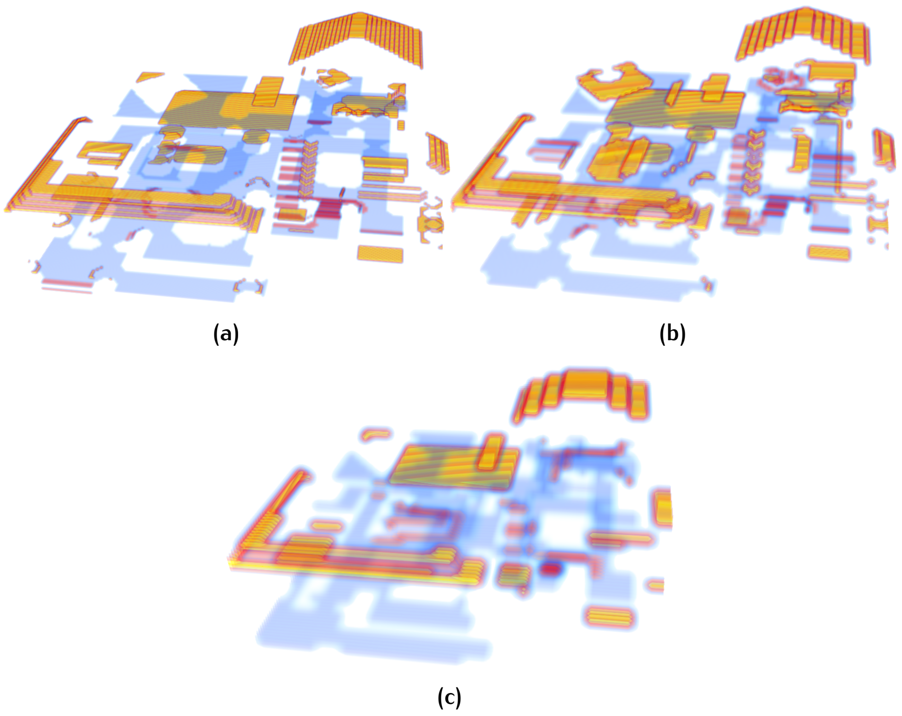


**Figure 5.3:** Semantic labels on various resolutions: 10 cm (a), 20 cm (b) and 40 cm (c). Floor is blue, stairs is red and obstacle is yellow.

Figure 5.3 shows the results of the semantic labelling for all three resolutions (without the upward propagation). It is noticeable that the higher the resolution of the model, the better the vertical footspan can represent the height of a stair and the better the algorithm can distinguish between obstacle and stairs. On a resolution of 20 cm also small obstacles with a maximum height of 30 cm are labelled as stairs like a part of the bathtub. On a resolution of 40 cm also larger obstacles with a maximum height of 60 cm are labelled as stairs like the couches in the lounge room and some furniture in the living room. On a resolution of 10 cm the algorithm can distinguish between obstacle and stairs fairly well. Only some baseboards around the hallway and living room are seen as stairs also.

### 5.1.3 Graph generation

Table 5.3 shows the statistics of the graph generation. It shows that the navigable space constitutes only a small percentage of the entire model. This is because only the voxels just above the floor and stairs are marked as navigable space. Next to that, an unexpected large amount of cells is formed in the 20 cm model. After the merging process there are almost as many cells in the 20 cm model as in the 10 cm model. The only real difference between the two is that the cells in the 10 cm model consist of more voxels.

Figure 5.4 shows the graphs that result from the graph generation. The number of nodes and edges in these graphs are directly related to the number of generated cells and that number is dependent on the resolution of the model.

| Resolution | 10 cm | 20 cm | 40 cm |
|---|---|---|---|
| Voxels in model | 1780200 | 222525 | 27676 |
| Voxels in navigable space | 11092 (0.6%) | 2456 (1.1%) | 734 (2.7%) |
| Initial cells | 125 | 107 | 39 |
| Cells after merging | 91 (73%) | 90 (84%) | 26 (67%) |
| Ratio cells to navigable voxels | 1 : 122 | 1 : 27 | 1 : 28 |
| Portals | 67 | 69 | 9 |
| Graph vertices | 134 | 138 | 18 |
| Graph edges | 354 | 229 | 18 |

**Table 5.3:** Statistics of graph generation for walking person.

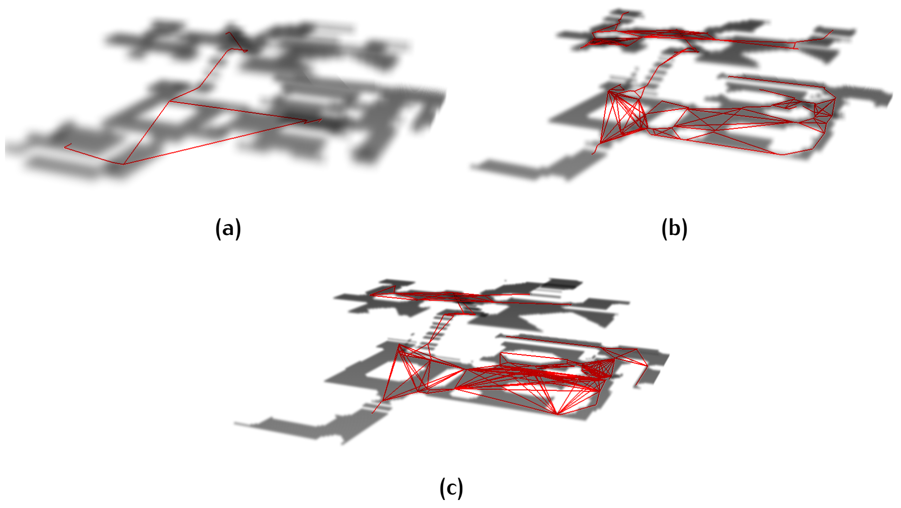
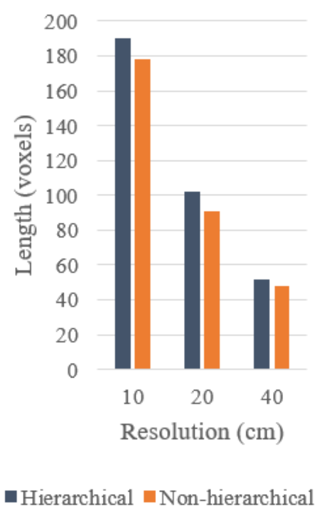
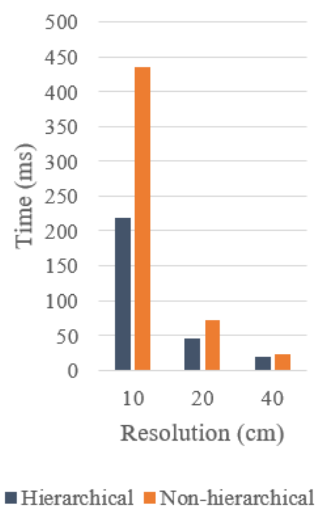


**Figure 5.4:** Generated graphs for walking person on various resolutions: 40 cm (a), 20 cm (b) and 10 cm (c).

### 5.1.4 Path-finding

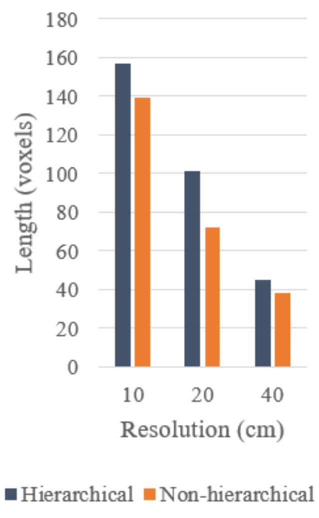
Figure 5.5 shows the quantitative results of the path-finding computations. For each path the length and computation time are plotted in a diagram. The computation time is the average of 10 runs. The exact numbers can be found in the appendices in Table A.1. It was not possible to measure the computation time for the third path with the 40 cm resolution model. The path-finding script ended so quickly that the timer returned 0. This probably has to do with the precision of the timer. The timer was unable to measure the elapsed time under 15 milliseconds.

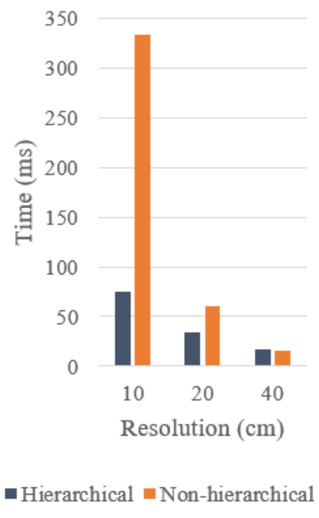




(a)

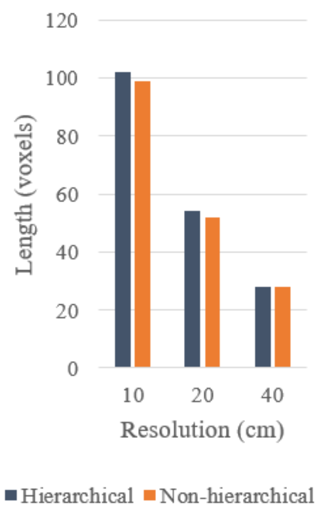
(b)

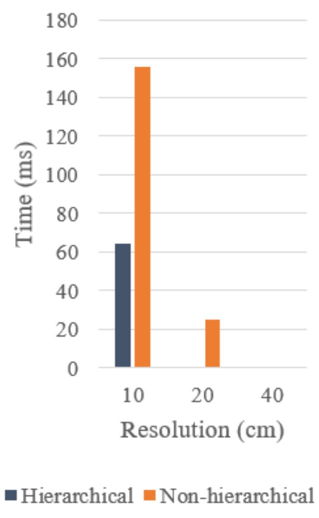




(c)

(d)





(e)

(f)

**Figure 5.5:** Path-finding results (length and computation time) for walking person. Path 1 (a & b), path 2 (b & c) and path 3 (e & f).

## 5.2 FLYING DRONE

### 5.2.1 Dilation of the model

The flying drone has the following dimensions: 40 x 20 cm. Table 5.4 shows the radius of the horizontal buffer and the extent of the downward extrusion for a given resolution.

| Resolution | Horizontal dilation radius (Represented diameter) | Downward extrusion (Represented height) |
|---|---|---|
| 10 | 2 (50 cm) | 1 (20 cm) |
| 20 | 1 (60 cm) | 0 (20 cm) |
| 40 | 0 (40 cm) | 0 (40 cm) |

**Table 5.4:** Dilation parameters for flying drone

The dimensions of the actor can not be represented correctly on any resolution. This is due to the creation of the horizontal buffer. The diameter of the actor is always represented as an odd number of voxels when a buffer is used to incoporate the size of the actor. The dimensions of the actor are represented most accurately on the 10 cm resolution.

### 5.2.2 Semantic labelling

The semantic labelling process is identical to that of the walking person. See Section 5.1.2.

### 5.2.3 Graph generation

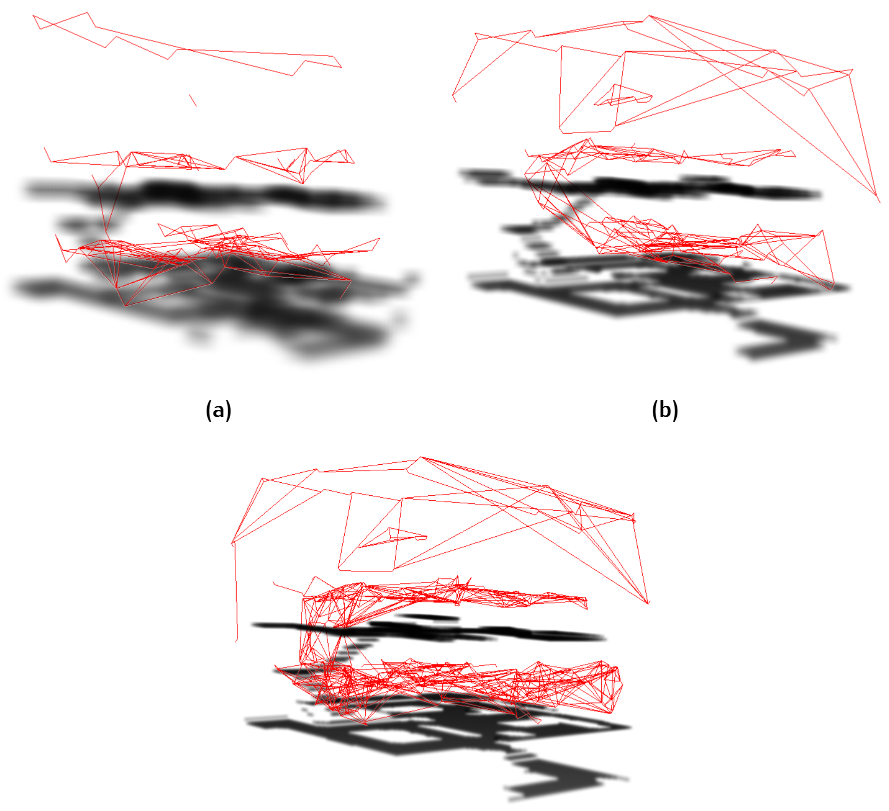
Table 5.5 shows the statistics of the graph generation. It shows that the navigable space is a large percentage of the entire model. This is because all empty space voxels are marked as navigable space. This has consequences for the rest of the graph generation compared to the walking person. First of all, there are a lot more cells generated. Secondly, each cell consists of a lot more voxels. This leads to a very high ratio between the number of cells and the number of navigable space voxels within these cells.

| Resolution | 10 cm | 20 cm | 40 cm |
|---|---|---|---|
| Voxels in model | 1780200 | 222525 | 27676 |
| Voxels in navigable space | 1298416 (73%) | 153144 (69%) | 19466 (70%) |
| Initial cells | 368 | 219 | 97 |
| Cells after merging | 141 (38%) | 80 (37%) | 56 (58%) |
| Ratio cells to navigable voxels | 1 : 9209 | 1 : 1914 | 1 : 348 |
| Portals | 206 | 84 | 56 |
| Graph vertices | 980 | 168 | 112 |
| Graph edges | 412 | 297 | 195 |

**Table 5.5:** Statistics of graph generation for flying drone.

Figure 5.6 shows the graphs that result from the graph generation. Each graph actually consists of 2 or 3 smaller disconnected graphs. There are multiple graphs for each model because there are multiple disconnected parts of navigable space. The first part of navigable space is the interior of building, the second part is the space above the building and the third part is the space in the attic.

A noticeable thing in the graphs is the vertical positions of the nodes. Nodes are created at the center of portals and most portals constitute a wall of voxels. The center of such a portal is somewhere in the middle between the floor and ceiling. However, there are variations between the portals and therefore there are also variations between the elevations of the portal centers. This results in the wavy pattern as can be seen clearly in the figure.



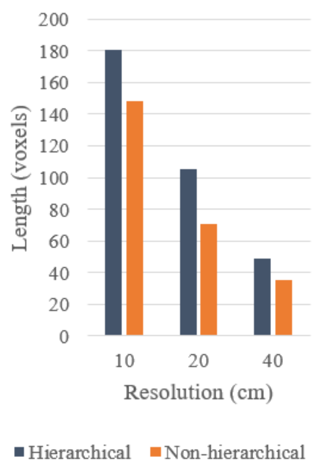
(a) (b)

(c)

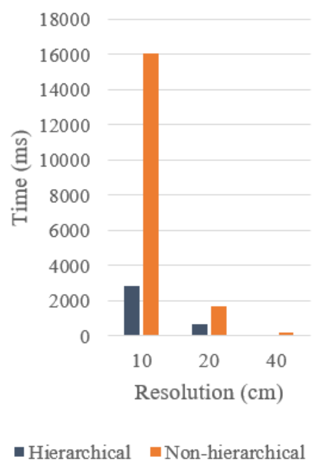
Figure 5.6: Generated graphs for flying drone on various resolutions: 40 cm (a), 20 cm (b) and 10 cm (c).

### 5.2.4 Path-finding

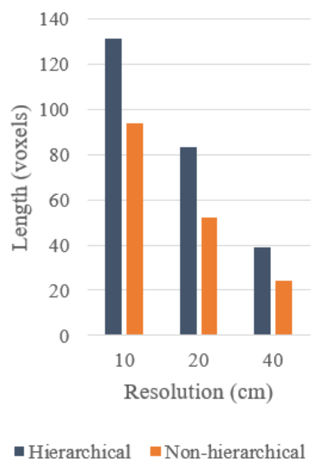
Figure 5.7 shows the quantitative results of the path-finding computations. For each path the length and computation time are plotted in a diagram. The computation time is an average of 10 runs. The exact numbers can be found in Table A.2.



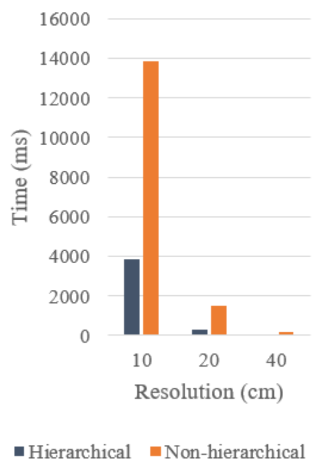
(a)



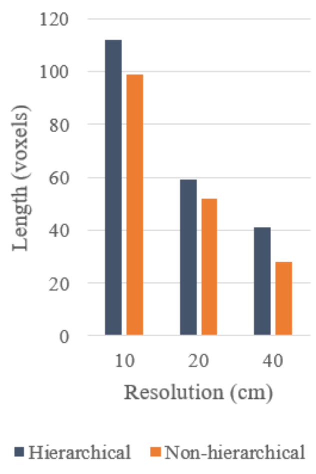
(b)



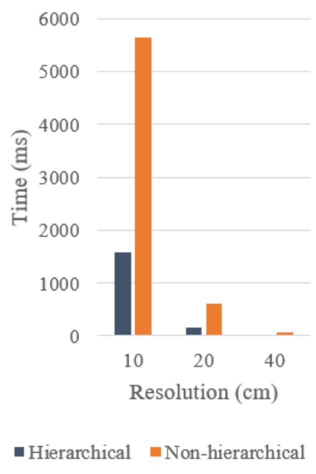
(c)



(d)



(e)



(f)

**Figure 5.7:** Path-finding results (length and computation time) for flying drone. Path 1 (a & b), path 2 (c & d) and path 3 (e & f)

Figure 5.8 shows the path with the greatest difference in length between the hierarchical and non-hierarchical approach. This path (path 2) starts on the floor in the lounge room and ends on the floor in bedroom 2. The path of the hierarchical approach tends to follow the centerline of a storey because this is where the graph resides. The path of the non-hierarchical approach goes directly towards the ceiling, to the staircase and then over the floor towards the ending point. In this particular case is the path of the hierarchical approach 40% longer. In another case (path 3) is the path of the hierarchical approach only 13% longer.

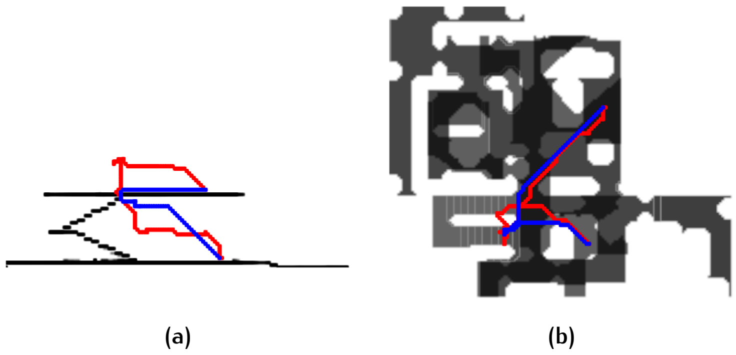


Figure 5.8: Path difference between hierarchical (red) and non-hierarchical (blue) approach. Side view (a) and top view (b).

## 5.3 ANALYSIS

### 5.3.1 Graph generation

*Cell generation*

The graph generation decomposes space into cells, identifies portals and derives a graph from these portals and cells.

The number of cells is dependent on the resolution of the input model. As stated in [Haumont et al., 2003], a high resolution model tends to have more local maxima in the distance field causing more cells to emerge when the watershed transformation is applied. The ratio between the number of generated cells and the number of voxels in each cell also increases with the resolution of the model. This is an important fact because the number of cells has a direct effect on the number nodes and edges in the derived graph.

The generated cells have irregular shapes and are seldom convex. They do not represent familiar building volumes like rooms. Their existence and shape depend on the computed distance field, but are also affected by the semantic labels. Each cell has to belong to a single semantic class (e.g., *floor*, *stairs* or *obstacle*) and therefore the semantic label acts as a constraint. Cell borders appear on locations where there is a transition between two semantic labels. For example, cell borders exist on the edge of a table because the cells above the table are labelled as *obstacle* while the surrounding cells are labelled as *floor*.

*Cell merging*

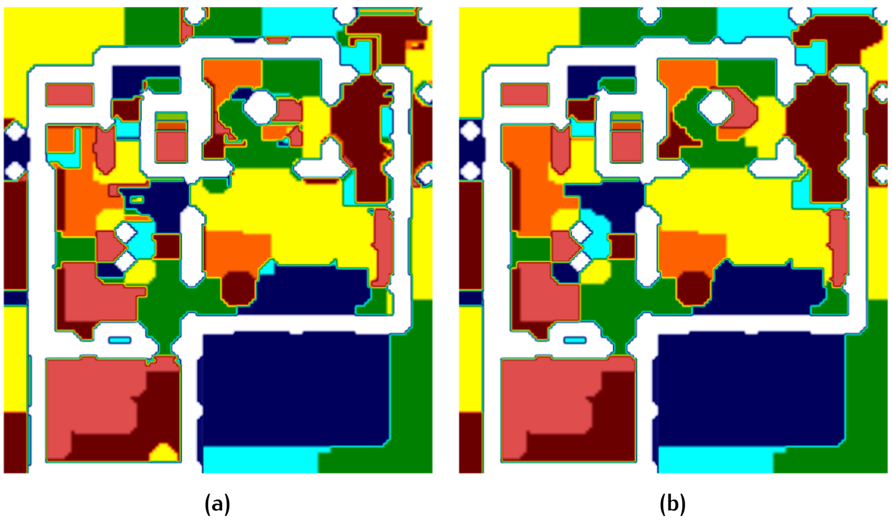


**Figure 5.9:** Cross section of cell decomposition for flying drone. Initial cells (a) and merged cells (b).

The cell merging process reduces the number of cells substantially. For the actor walking person around 75% of the cells remain and for the actor flying drone around 44%. Figure 5.9 shows the cell decomposition before and after the cell merging. Small cells are merged and the remaining cells become more regularly shaped.

### 5.3.2 Path-finding

### 5.3.3 Path length

It can be concluded from both tests that the length of the path computed using the hierarchical approach is longer compared to that of the non-hierarchical approach. This fact was also stated in [Botea et al., 2004]. The extent of this difference differs between the two actors. The difference is greater for the flying actor than it is for the walking person. This results from the fact that the computed path has to go through the nodes of the graph according to the methodology of [Botea et al., 2004]. The graph of the flying drone resides near the centerline of a storey in the middle between the floor and the ceiling and will therefore force the path through this area. This has a great influence on the path length for a flying actor (Figure 5.8). This problem is less prominent for the walking actor because this graph resides just above the ground surface exactly where the walking actor normally moves.

The graph forces the actor to move through a two-dimensional surface that coincides with the graph itself. Therefore, it does not really support actors that should be able to move up and down in space (e.g. flying actors).

### 5.3.4 Computation time

The purpose of the hierarchical approach is to reduce the time complexity. The hierarchical approach fulfils this purpose for both actors, but the time reduction is not as substantial as claimed in [Botea et al., 2004]. They claim to be 10 times faster than standard A*. The results show that the implemented methodology is 4 times faster for the flying actor and 3 times faster for the walking actor.

The time reduction depends on the ratio between the number cells and the number of voxels within each cell. It is not exactly clear what this ratio should be, but a high ratio appears to be more promising because this is the case in the 10 cm resolution model. The difference in computation time between the hierarchical and non-hierarchical approach is greatest in the 10 cm resolution model.

It is assumed that at some point, when the ratio becomes very high, the time reduction will decrease again. This means extra levels of abstraction are required. However, this is not implemented in the methodology. Only one level of abstraction is supported.

# 6 CONCLUSION, DISCUSSION AND FUTURE WORK

## 6.1 RESEARCH QUESTIONS

In this section the research questions are answered based on the results of this thesis. First the subquestions are answered and then the main research question.

*What kind of actors exist in an indoor environment?*

There are many different actors that require path-finding in an indoor environment. One example is a walking person that visits a building for the first time. The computed path could be presented to this person through a screen on a handheld device to enable navigation. Not all people in an indoor environment are walking. Some of them make use of a transportation device like a wheelchair, Segway or hoverboard. These people have a different mode of locomotion: driving.

In recent years there has been emerging a new group of non-human actors. These are the robots that fulfill some kind of function in a building like floor cleaning, guiding people or assisting people. Most of these robots drive, but some of them fly like Blue Jay (`https://www.bluejayeindhoven.nl/`) which is a drone supposed to help people in everyday life.

*What requirements does each kind of actor have on the computed path?*

Each actor has several characteristics that influence the required path. One of these characteristics is the size of the actor. The size of the actor limits the actor from going through spaces which are too small. For example, a person can not go underneath a table while a vacuum cleaner robot can.

Another characteristic is the mode of locomotion of an actor. There are many modes of locomotion possible like rolling, jumping, crawling and gliding, but the most obvious modes of locomotion are walking, driving and flying. An important aspect of the mode of locomotion is the available navigable space. Walking and driving actors are bound to a ground surface and therefore the navigable space of these actors can be seen as a two-dimensional surface in three-dimensional space. In contrast, the navigable space of a flying actor is truly three-dimensional as the actor can move up and down.

Apart from these two physical characteristics, there is also a third less descriptive characteristic that has influence on the path-finding. This is the actor's notion of the best path. The notion of the best path varies between actors and is sometimes hard to define. For example, the best path for a walking person may be the shortest path, but may also be the path with the least turns or the path through the most appealing surroundings. Other actors have a more easily determinable best paths like that of a vacuum cleaner robot. This path should visit every location at least once to ensure that the entire floor is cleaned.

A path-finding method should adhere to the characteristics of an actor to find the most suitable path.

*What parameters can describe the required path of an actor?*

Parameters that describe the requirements of each actor for path-finding are defined to generalize the problem. These parameters allow a generic path-finding solution that covers all aspects described by the parameters.

First of all, some parameters have to describe the shape and size of the actor. The actor is a three-dimensional object and therefore the bounding box of the actor can be represented by three parameters: width, length and height. The number of parameters can be reduced to one or two parameters also. For example, if two parameters are used, then the length and width of the actor are represented by the same parameter and therefore they are always equal. If the length of the actor is included in the parameters and the length of the actor is greater than the width of the actor, then the orientation of the actor has to be considered also. In this scenario the overall width of the actor increases when the actor makes a turn due to its length. That means a fourth parameters must be included to represent the rotation of the actor around the vertical axis. However, Canny [Canny., 1988] stated that the complexity of finding a shortest path is exponential to the number of degrees of freedom. It is therefore better to represent the shape and size by only two parameters (diameter and height) instead of four. This reduces the path-finding problem substantially, but does not support actors that are longer than they are wide (or vice versa).

Other parameters have to describe the mode of locomotion or, more specifically, the navigable space for a certain mode of locomotion. To realize this, the space must be semantically labelled by one of the following classes: *above floor*, *above stairs* and *above obstacle*. If the space is semantically labelled in such a manner, then it is possible to declare the navigable space for each mode of locomotion based on the semantic classes. For example, driving actors can navigate through space labelled as *above floor* and walking actors can also navigate through space labelled as *above stairs*. In this way the navigable space of an actor can be represented by three Boolean parameters. One for each semantic class. This approach supports the three most common modes of locomotion: walking, driving and flying.

The other characteristic, notion of the best path, can not be expressed by a parameter. This notion varies too much between actors and each notion would require a different implementation. Therefore, it was chosen to consider the shortest path as the best path because this type of path is applicable for many applications. As stated in [Zlatanova et al., 2014], the most commonly used strategies for path-finding are the shortest distance and shortest time.

*In what data structure should the voxels be stored to facilitate pathfinding?*

Voxels can be stored in any data structure that permits accessing a voxel by its XYZ-coordinates. One solution is to store voxels as a one- or three-dimensional array. If a three-dimensional array is used, then the X, Y and Z coordinates correspond to the indices of the array. If a one-dimensional array is used, then the index of the array has to be calculated with the following formula:

$$index = x + (y * dimX) + (z * dimX * dimY)$$

In which $dimX$, $dimY$ and $dimZ$ are the extent of the array in the three dimenions.

Constructing such an array will allocate memory for all voxels of the volume. This may be desired, but for some applications it is not the most effective way of storing voxels.

Another solution is to store voxels using a hash table. A hash table is a data structure that permits storing values by custom indices (e.g. keys). In such a data structure the tuple (x, y, z) can be used as a key to access a voxel. An advantage of a hash table is that it is able to store only a subset of the total number of voxels. For example, all non-empty space voxels may be stored in a hash table while the empty space voxels are omitted. This can reduce the memory consumption substantially.

The vtkImageData format of VTK makes use of a one-dimensional to store the voxels. A hash table data structure is used in some of the implemented Python scripts. For example, each generated cell of the cell generation script is stored as a hash table. The implementation of a hash table in Python is called a dictionary.

*What is the influence of the model's resolution on the path-finding?*

The resolution of the model is a very important characteristic. It defines how much detail can be represented and it determines the time complexity of the path-finding problem. The higher the resolution of the model, the more accurately the environment can be represented, but the greater the time complexity of the path-finding problem is.

It was tried to automatically assign one of the semantic labels (*floor*, *stairs*, *obstacle*) to all voxels. The methodology therefore – extracting horizontal surfaces, segmentation and slope estimation – works best on the 10 cm model although it is not completely fail proof. Small objects up to 25 centimetres are still labelled as stairs.

The size and shape of the actor are incorporated in the path-finding method by dilating the model. This dilation is done based on the diameter and height of the actor. The extent of the dilation is always a multitude of the model's resolution. Therefore, the dimension of an actor can be represented more precise in a high resolution model.

The results of the dilation and semantic labelling suggest that a high resolution model is better because it can better represent the environment and the actor within it, but it is also computationally more expensive to find a path in a high resolution model. Therefore, there is a tradeoff between a good representation and fast path-finding.

*What implementations could improve the performance of the pathfinding method?*

The performance of a path-finding method consists of multiple aspects including memory consumption, optimality of the path and time complexity for finding the path. A* [Hart et al., 1968] is a commonly used shortest path algorithm and is also implemented in the methodology of this thesis. The memory consumption of this algorithm can be lowered by Iterative Deeping A* [Korf, 1985] which is an extension of A*. The optimality of the path is improved in Theta* [Nash et al., 2007]. Theta* performs line-of-sight analysis during expansion of the navigation front to check whether following nodes are reachable. If so, then a straight path is drawn to the following. This eliminates the jagged pattern of a path in a grid.

The time complexity of a path-finding algorithm depends on the size of the search space. The search space can be reduced by utilizing a hierarchical data structure. Most hierarchical path-finding methods perform two steps. First, a graph (tree) is built from the navigable space. Secondly, a path is found by searching the graph.

One solution for hierarchical path-finding is HPA* from [Botea et al., 2004] described in Section 2.2.3. HPA* offers hierarchical path-finding in a two-dimensional grid and many hierarchical path-finding algorithms have been developed based on HPA*. Hierarchical Annotated A* (HAA*) [Harabor and Botea, 2008] extends HPA* by considering the dimensions of actors as squares, DT-HPA* [Li et al., 2012] extends HPA* by using a decision tree for hierarchical subdivision and Hierarchical Path-Finding Theta* (HPT*) [van Elswijk et al., 2013] extends HPA* by combining it with the shortest path algorithm Theta*. Another similar algorithm for hierarchical path-finding is Partial Refinement A* (PRA*) [Sturtevant and Buro, 2005]. This algorithm creates abstraction by mapping regions of 2x2 tiles to tiles on the next level.

All of these algorithms (HPA*, PRA* and its descendants) do not work in a 3D environment. Their methodology for creating abstraction is targeted on 2D grids. A single level abstraction has been realized in this thesis research by deriving a graph from a cell decomposition.

## 6.2 CONCLUSION

*Is it possible to develop a single, uniform path-finding method that is applicable for different kind of actors?*

A path-finding method has been developed that supports different kinds of actor. For each actor the size and mode of locomotion are taken into consideration. The size of the actor is represented by two parameters: diameter and height. That means the width and length of each represented actor are always equal.

This path-finding method involves deriving a graph from a voxelized model. This graph enables hierarchical path-finding and reduces the computation time for path-finding substantially. Constructing the graph is an elaborate process that has to be performed for each distinct actor. This makes the path-finding method only suitable for a single actor in a static environment. A semantically labelled input model is required for the graph generation. A methodology for semi-automatic semantic labelling is proposed in this thesis. The semantic labelling process labels all voxels correctly in a high resolution model as long as there are no obstacles with the same height as a stair in the staircase.

The graph is derived from a cell decomposition of the navigable space. This cell decomposition is realized by performing a watershed transformation on a distance field of the input model. The semantic labels form an extra constraint in this cell generation porocess. All voxels belonging to the same cell must have the same semantic label. That means the cell decomposition depends on the geometry and semantics.

The path-finding method has been tested for two different actors: a walking person and a flying drone. The results of the walking person are promising. The computation time is reduced substantially compared to a non-hierarchical approach and the difference in path length is small. The results of the flying drone are a bit less promising. Although the computation time is reduced even more using the hierarchical approach, the difference in path length is a lot bigger. The path is far from optimal in some situations. It can be concluded that current methodology does not work well for flying actors because of this big difference with the optimal path, but it does work well for walking actors.

The methodology has not been tested for driving actors as thoroughly as for the walking and flying actors, but it is expected that the results for a driving actor will be very similar to that of a walking actor. Both actors have similar properties and a similar navigable space. The only difference is that the space above the stairs does not belong to the navigable space of a driving actor.

## 6.3 DISCUSSION

### 6.3.1 Semantics

Three different semantic classes are used: *floor*, *stairs* and *obstacle*. These classes are used to determine the navigable space of an actor based on its mode of locomotion. Labels that correspond to these classes are assigned to voxels and generated cells. It is hereby possible to incorporate these semantics in the path-finding methodology.

The semantic classes are not thematic semantics. They do not tell something about the identified meaning of a space, but rather about the geometrical properties of the space. The semantic labels are also automatically derived from the geometric model. One could therefore argue that they are not really semantic classes.

It is however possible to incorporate thematic semantics by the same means. Each cell can be assigned one or more labels. These labels will be propagated onto the graph during the graph generation. That means the edges and nodes receive the same labels. It is then possible to perform path-finding based on these labels. For example, if each cell is assigned a label holding the room number, then each node and edge within that space acquires the same label. This makes it possible to find a path between two cells based on their room number.

It must be noted that the cell decomposition works based on the distance field and the three semantic classes. Space is subdivided on the transitions between two semantic labels. These labels can therefore be used to subdivide space into meaningful cells (e.g., one cell for the food corner). If all semantic labels are omitted, then space is subdivided based only on the distance field (geometry).

### 6.3.2 Shape of actor

The shape and size of the actor have to be considered in the path-finding method to support different kind of actors. It was chosen to dilate model based on the actor's properties. This makes it possible to consider the actor as a single point during the path-finding, but it has consequences for the represented shape of the actor.

Two different values are used for the dilation. One affects the horizontal plane and one the vertical plane. That means the represented width and length of the actor are determined by the same value. The actor is therefore represented as long as it is wide. This results in the fact that the shape of the actor is represented by a cylindrical shape. The diameter of this cylinder corresponds to the actor's width and length.

This methodology is not capable of representing an actor that is longer than it is wide (or vice versa). That situation introduces a new problem: the orientation of the actor. In such scenario is the navigable space of the actor dependent on its orientation. That means the shape and size of the actor have to be represented by 4 parameters (width, length, height and orientation) instead of 2 (diameter and height). One solution for this approach is the multidimensional approach of [Verbeek et al., 1986]. This approach makes a 4-dimensional model in which the fourth dimension corresponds to the actor's orientation. However, this approach has a time complexity that is exponentially greater due to the extra dimension [Canny., 1988].

### 6.3.3 Applicability for different kind of actors

As stated in [Zlatanova et al., 2014], each actor has certain *preferences* like the type of path. It was chosen to consider the shortest path as preferred path because different kinds of paths can not be realized by a single, uniform path-finding method.

The methodology suffices for a vast variety of actors like walking people, people in wheelchairs, robots, drones, etcetera, but some actors may require additional information in the form of semantics. For example, a waiter in a restaurant may require some information about the kitchen and the dining tables. These semantics can be added to the methodology with some minor modifications as described in Section 6.3.1.

The methodology utilizes a graph to perform path-finding on level 1. The actor has to move through the nodes of the graph according to the methodology of [Botea et al., 2004]. This works very well for a walking or driving actor because the graph coincides with its navigable space, but this does not work well for flying actors. The navigable space of a flying actor is far greater than the area surrounding the graph and the actor should be able to move away from the graph (e.g., up and down). This results in a non-optimal path for flying actors bacause the path is drawn towards the graph which resides in the middle of a storey between the floor and ceiling. However, this may be a desired property. It may be desired that a drone flies on this elevation to keep maximum clearance from the floor and ceiling. It is also be possible to adapt the current methodology to set a preferred flying height. For example, if the graph portals are constructed in such a way that they are always located 2 meters above floor level, then this is the elevation where the graph will be constructed and where the flying drone will be forced to fly through.

A model is constructed for each distinct actor. This model consists of a graph and a collection of cells. Creating this model is an elaborate process and requires some time. In the current implementation, it may take up to an hour because it requires some tweaking by the user and the watershed transformation algorithm is not built on performance. The model is also static. It does not offer capabilities to be updated when the environment changes. The path-finding method is therefore suitable for applications in which the properties of the actor and environment are known in advance and do not change over time.

## 6.4 FUTURE WORK

### 6.4.1 Supporting different notions of the best path

The path-finding method only supports one type of path: the shortest path. This path is the suitable for many actors, but there are also other actors with a different notion of the best path. Here are two examples:

FIREFIGHTER When a firefighter sweeps a building he moves along the walls of the building. Such a path can not be computed using a shortest path algorithm like A* and requires a different approach. In this approach the navigation front of the path-finding method should spread out along the walls of the building. This requires the algorithm to know where walls are and therefore the semantic labelling of the current implementation should be extended by a fourth class: *wall*.

VACUUM CLEANER ROBOT A vacuum cleaner robot should cover the whole ground surface and preferably visit each location once. Such a path is the direct opposite of the shortest path and is called the Hamiltonian path. Computing this path requires a different algorithm like the Minram algorithm [Thompson and Singhal, 1985].

### 6.4.2 Maintain preferred flying height

For a flying actor like a rotary-wing drone it is more energy consumptive to move up than it is to move down or in a horizontal direction. It is therefore better to maintain a fixed elevation. A path with a fixed elevation is probably also easier to fly because vertical movement can be neglected most of the time.

One solution that could be investigated is by combining two navigation fronts. The first navigation front guides the actor towards the end point and the second navigation front guides the actor to the preferred elevation. By accumulating the two navigation fronts a new navigation front arises that tends to push the actor to the end point and the preferred flying height.

### 6.4.3 Supporting elevators

Supporting elevators requires adaptation of the current methodology, but should be feasible. First of all, the graph should incorporate the elevators. That means new portals must be created at the locations of elevator doors and edges must connect these portals through the elevator shaft. Path-refinement should not occur when the path enters the elevator shaft. This can be accomplished by introducing a fourth semantic class: *elevator*. This class would obtain the numerical label 4. This label should be assigned to cells that reside within the elevator shaft. The path-finding algorithm is then able to skip path-refinement if it encounters a cell with this label. Detecting elevator doors and semantically labelling the space in the elevator shaft may not be possible in an automated manner. It may require manual intervention.

### 6.4.4 Testing for driving actor

The path-finding method has not been tested thoroughly for a driving actor. It is expected that the results of a driving actor will be very similar to that of a walking actor because they are closely related.

### 6.4.5 Path-smoothening

The found path tends to be jagged because of the inherent orthogonal structure of the voxelized model. This jagged path is more apparent in low-resolution models and may not be optimal for some applications. In [Bandi and Thalmann, 2000] and [Botea et al., 2004] a solution is offered to overcome this problem. They smoothen a path by enumerating the path positions and checking whether subsequent path positions can be reached in a straight line. If this happens, the straight line replaces the initial path. The resulting path is smoother and closer to the Euclidian straight line path as can be seen in Figure 6.1.

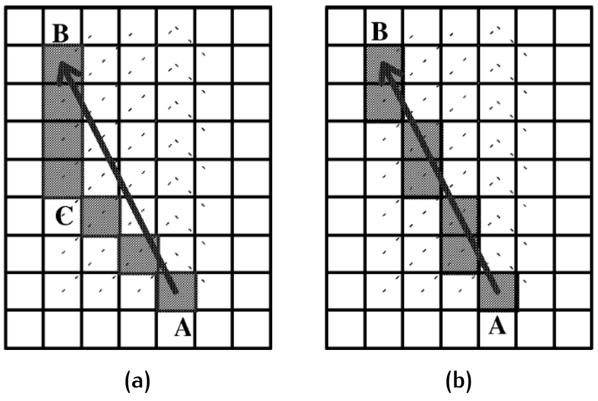


**Figure 6.1:** Path-smoothening. Jagged 8-connectivity path (a) and smooth path (b).

# BIBLIOGRAPHY

Andújar, C., Vázquez, P., and Fairén, M. (2004). Way-finder: Guided tours through complex walkthrough models. *Computer Graphics Forum*, 23(3 SPEC. ISS.):499–508.

Bandi, S. and Thalmann, D. (2000). Path finding for human motion in virtual environments. *Computational Geometry*, 15(1-3):103–127.

Borgefors, G. (1986). Distance transformations in digital images. *Computer Vision, Graphics and Image Processing*, 34(344):344–371.

Botea, A., Müller, M., and Schaeffer, J. (2004). Near optimal hierarchical path-finding. *Journal of game development*, pages 1–30.

Canny., J. F. (1988). *The complexity of robot motion planning.* MIT Press.

Cui, X. and Shi, H. (2011). A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130.

Dijkstra, E. W. (1959). A Note on Two Probles in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271.

Doran, J. E. and Michie, D. (1966). Experiments with the graph traverser program. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 294, pages 235–259. The Royal Society.

Dorst, L. and Verbeek, P. (1986). *The constrained distance transformation: A pseudo-euclidean, recursive implementation of the Lee-algorithm*, pages 917–920. Elsevier Science Publishing.

Goldstein, R., Breslav, S., and Khan, A. (2014). Towards voxel-based algorithms for building performance simulation. In *Proceedings of the IBPSA-Canada eSim Conference*.

Grevera, G. J. (2007). Distance transform algorithms and their implementation and evaluation. In *Deformable Models*, pages 33–60. Springer.

Harabor, D. and Botea, A. (2008). Hierarchical path planning for multi-size agents in heterogeneous environments. pages 258–265.

Hart, P., Nilsson, N., and Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths.

Haumont, D., Debeir, O., and Sillion, F. X. (2003). Volumetric cell-and-portal generation. *Computer Graphics Forum*, 22(3):303–312.

Jones, M. W., Bærentzen, J. A., and Sramek, M. (2006). 3d distance fields: A survey of techniques and applications. *Visualization and Computer Graphics, IEEE Transactions on*, 12(4):581–599.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109.

Latombe, J.-C. (1990). Robot motion planning (the kluwer international series in engineering and computer science).

Li, Y., Su, L.-M., and Li, W.-L. (2012). Hierarchical path-finding based on decision tree. In *International Conference on Rough Sets and Knowledge Technology*, pages 248–256. Springer.

Meijers, M., Zlatanova, S., and Pfeifer, N. (2005). 3D geoinformation indoors: structuring for evacuation. *Proceedings of Next generation 3D city models*, pages 21–22.

Nash, A., Daniel, K., Koenig, S., and Felner, A. (2007). Theta^*: Any-angle path planning on grids. In *Proceedings of the national conference on artificial intelligence*, volume 22, page 1177. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

Rabin, S. (2000). A* speed optimizations. In *Game Programming GEMS*, pages 264–271. Charles River Media, America.

Sturtevant, N. and Buro, M. (2005). Partial pathfinding using map abstraction and refinement. In *AAAI*, volume 5, pages 1392–1397.

Thompson, G. L. and Singhal, S. (1985). A successful algorithm for the undirected hamiltonian path problem. *Discrete applied mathematics*, 10(2):179–195.

van Elswijk, L., Sprinkhuizen-Kuyper, I., and Wiedijk, F. (2013). *Hierarchical Path-Finding Theta^*.*

Vandapel, N., Kuffner, J., and Amidi, O. (2005). Planning 3-D path networks in unstructured environments. *Proceedings - IEEE International Conference on Robotics and Automation*, 2005(April):4624–4629.

Verbeek, P., Dorst, L., B.J.H., V., and Groen, F. (1986). Collision avoidance and path finding through constrained distance transformation in robot state space. In *Robot State Space*, pages 634–641. International Conference, Amsterdam.

Xiong, Q., Zhu, Q., Zlatanova, S., Du, Z., Zhang, Y., and Zeng, L. (2015). Multi-Level Indoor Path Planning Method. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XL-4/W5(May):19–23.

Yuan, W. and Schneider, M. (2010). Supporting 3D Route Planning in Indoor Space Based on the LEGO Representation. *Proceedings of the 2Nd ACM SIGSPATIAL International Workshop on Indoor Spatial Awareness*, November:16–23.

Zlatanova, S., Liu, L., Sithole, G., Zhao, J., and Mortari, F. (2014). *Space subdivision for indoor applications*. OTB Research for the Built Environment, Delft.

# A PATH-FINDING RESULTS

| Path | Resolution | Hierachical | Length | Time (microsec.) |
|---|---|---|---|---|
| 1 | 10 | No | 178 | 435000 |
| | | Yes | 190 | 219000 |
| | 20 | No | 91 | 71400 |
| | | Yes | 102 | 45300 |
| | 40 | No | 48 | 23400 |
| | | Yes | 52 | 19000 |
| 2 | 10 | No | 139 | 333300 |
| | | Yes | 157 | 75000 |
| | 20 | No | 72 | 60600 |
| | | Yes | 101 | 34300 |
| | 40 | No | 38 | 16000 |
| | | Yes | 45 | 16900 |
| 3 | 10 | No | 99 | 156000 |
| | | Yes | 102 | 64400 |
| | 20 | No | 52 | 24700 |
| | | Yes | 54 | X |
| | 40 | No | 28 | X |
| | | Yes | 28 | X |

Table A.1: Path-finding results for walking person.

| Path | Resolution | Hierachical | Length | Time (microsec.) |
|---|---|---|---|---|
| 1 | 10 | No | 148 | 16033100 |
| | | Yes | 181 | 2837300 |
| | 20 | No | 71 | 1686700 |
| | | Yes | 105 | 654700 |
| | 40 | No | 35 | 192200 |
| | | Yes | 49 | 71700 |
| 2 | 10 | No | 94 | 13822300 |
| | | Yes | 131 | 3858300 |
| | 20 | No | 52 | 1493900 |
| | | Yes | 83 | 321900 |
| | 40 | No | 24 | 148600 |
| | | Yes | 39 | 43500 |
| 3 | 10 | No | 99 | 5644800 |
| | | Yes | 112 | 1578100 |
| | 20 | No | 52 | 607900 |
| | | Yes | 59 | 159400 |
| | 40 | No | 28 | 64300 |
| | | Yes | 41 | 29600 |

**Table A.2:** Path-finding results for flying drone.

# B REFLECTION

This thesis presents a new path-finding method. This method meets requirements that have not yet been described in literature before. First of all, the method targets a 3D representation of an indoor environment. Secondly, it supports different types of actors based on their dimensions and mode of locomotion. Thirdly, a hierarchical data structure has been used to reduce the computation time and enable (near) real-time path-finding.

This method is developed as a proof-of-concept to demonstrate whether or not it can meet the predetermined requirements. The method has been developed by extending existing concepts from literature.

Within the domain of geomatics is indoor path-finding an important topic for certain applications. One example is the realisation of a navigation system for people visiting a building. Next to that, also non-human actors like domotica robots require indoor path-finding for fulfilling their functions in a building. Applications like these will be emerging more and more in the future.

In this thesis research knowledge has been applied from various MSc Geomatics courses. Most notably Python programming and 3D modelling. Python programming has been used to implement algorithms from literature that perform 3D spatial operations. Knowledge of 3D modelling plays a crucial role in these algorithms.