Frustumbug: a 3D mapless stereo-vision-based bug algorithm for Micro Air Vehicles

R.S. Meester , T. van Dijk [*], C. De Wagter [†], G.C.H.E. de Croon [‡] [†]
Control and Simulation, Faculty of Aerospace Engineering
Delft University of Technology, The Netherlands

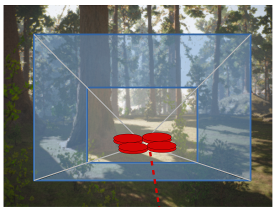
Abstract

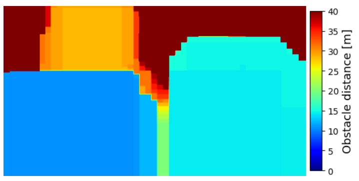
Obstacle avoidance is an important capability for flying robots. But for robots with limited resources, such as small drones this becomes particularly challenging. Bug algorithms have been proposed to solve path planning with only minimal resources. And stereo vision provides a rich description of the world for limited weight but typically has a limited Field of View (FoV) and is fixed to the drone frame to further reduce weight. Based on these, a computationally light 3D path-planning algorithm is proposed. The proposed algorithm is called Frustumbug and is based on the Wedgebug algorithm since this algorithm copes well with a limited FoV. Since Wedgebug only addresses 2D problems, the Local-ϵ-Tangent-Graph (LETG) is used to extend the path planning to 3D. Disparity images are obtained through an optimized stereo block matching algorithm. Frustumbug copes well with noisy range sensor data and includes 3D trajectories like reversing, climbing and descending maneuvers to avoid or escape local minima. The algorithm has been tested with 225 flights in two challenging simulated environments and achieved a success rate of 96%. Here, 3.6% did not reach the goal and 0.4% collided. Frustumbug has been implemented on a 20-gram stereo vision system and was tested in the real world on a MAV. This shows the potential for small drones to reach their targets fully autonomously based on very limited resources.

1 Introduction

The drone market keeps growing, with applications ranging from industrial inspection to search-and-rescue and package delivery. Since most of the applications have to be performed autonomously, autonomous obstacle avoidance is of the essence. Current solutions combine multiple sensors and are computationally expensive and memory intensive. An efficient, lightweight solution would require less energy from the battery, increasing the flight time while reducing weight requirements.

Over the past few decades, several distance measurement sensors have been studied. To allow for onboard path planning, vision-based sensors are the only promising candidate when looking at size and weight and power [Aswini et al., 2018]. To limit the computational resources used by the drone, a mapless method is preferred over a map-building method. Since the outdoor flight is assumed, building maps of large, complex environments requires high resources.


(a) Drone's viewing frustum



(b) Depth image obtained from the information inside frustum

Figure 1: Creation of the 3D viewing frustum. The drone is represented in red. The red dotted line indicates the traveled path. Depth image includes a safety margin around obstacles.

Ideally, the generated path should resemble the globally shortest path. Therefore, the drone's ability to move up or down to avoid obstacles needs to be exploited, hence this paper focuses on path planning in three dimensions. 3D mapless vision-based methods exist but are often combined with probabilistic roadmaps [e.g. Matthies et al., 2014 and Lee et al., 2021] or machine learning algorithms [e.g. Doukhi and Lee, 2022 and Grando et al., 2022]. Since there is a certain amount of randomness in these methods, the choice of paths can not always be explained, and may not always lead to the goal. Furthermore, these methods tend not to include strategies for escaping local minima, or use randomly generated sub-optimal paths to escape [e.g. Matthies et al., 2014 and Yu et al., 2018].

Bug algorithms are used for robots with limited computational resources since they plan their path using very little computation and memory. Another advantage is that these finite-state machines provide a strategy for escaping local minima. Wedgebug [Laubach and Burdick, 1999] is a two-dimensional bug algorithm that solved the problems for robots with a small Field of View (FoV), by rotating the sensor when more information is required. However, to create an accessible algorithm for the many drones and cameras without that capability, and to limit the necessary moving components, the algorithm should be designed without the requirement of camera rotation. When stuck in a local minimum, Wedgebug will follow the obstacle boundary until an intermediate waypoint is found which decreases the distance to the goal [Laubach and Burdick, 1999]. However, in 2D an

[*]J.C.vanDijk-1@tudelft.nl
[†]c.dewagter@tudelft.nl
[‡]g.c.h.e.decroon@tudelft.nl

obstacle can only be avoided in two directions: left or right. In 3D, the number of directions to avoid an obstacle increases to infinity, since any location on the obstacle boundary can be chosen. 3DBug [Kamon et al., 1996] reduces the necessary calculations by discretizing the continuous obstacle contour into a set of points. However, 3DBug assumes an infinite sensor range and a 360 degrees FoV. Furthermore, it has been tested on relatively simple environments where the locally best decisions mostly lead to the globally best paths.

The goal of this study is to create a 3D path planning algorithm, which can navigate a drone to a target while minimizing deviation from the nominal path using limited resources. We contribute to the existing literature by; (i) extending Wedgebug to operate in 3D, (ii) introducing additional navigation states for avoiding and escaping local minima, (iii) designing a novel waypoint selection method to increase robustness to noisy range sensor data, (iv) extensively testing the proposed algorithm in simulation and real-world, using a very light-weight stereo vision system. We will call it Frustumbug, as the frustum can be seen as the 3D extension of a wedge. Figure 1 shows how the viewing frustum is shaped (Figure 1a), along with its corresponding configuration space (Figure 1b). Our contributions take the next step towards a publicly available self-contained path planning package and allow for future extensions since each path planning decision follows from a logical decision captured in a decision tree, and each branch can be altered individually. Assumptions include outdoor flight during daylight, where GPS signal is available. Furthermore, the target and obstacles are assumed static and maze-like environments are not considered. The remaining part of the paper is structured as follows: first, section 2 shows the related work and section 3 explains how the chosen approach is implemented. Then, section 4 presents the experimental results from both simulation and real-world testing. This paper then discusses the results in section 5 and concludes in section 6.

2 Relevant Work

The relevant work is divided into two sections: we first look at the literature on distance measurement sensors to optimize our sensing, followed by literature on avoidance algorithms to complete the path planning package.

2.1 Sensing

Vision-based sensors are suitable for outdoor environments since they usually have enough texture to be captured with a camera [Matthies et al., 2014]. Deep/Machine learning algorithms are not considered for obtaining depth information, since these methods consist of multiple convolutional layers, which require large matrices to be stored and many computations to be made [e.g. Doukhi and Lee, 2022]. Moreover, the parameters of a neural network result from training the algorithm for a longer period of time. If it shows inconsistent results for repetitive image texture or low-textured regions, the parameters can not be easily adapted to improve the result. It would have to be trained again on an improved dataset, or with different hyperparameters.

Vision-based sensing can be done with one, two or more cameras. Monocular vision in obstacle avoidance is generally paired with optical flow methods. These have their drawbacks in obtaining disparity images compared to stereo vision: a larger image area needs to be searched to find matching pixels, estimates on position and attitude changes need to be included in the calculation and it is not able to sense distances in the direction of travel, since there is no optical flow in the focus of expansion [van Dijk, 2020]. Following the argument above, stereo vision is preferred since the added weight and required power are small; they can be as lightweight as 4 grams and be run on 168 MHz microprocessors [McGuire et al., 2017]. To limit the computational resources, a solution using more than two cameras is not considered.

Stereo matching can be done either local or global, and sparsely or densely. Although global methods tend to generate better results in low-texture environments, it comes at a computational cost [Liu et al., 2020]. Furthermore, outdoor environments during daylight have enough texture for local methods. Sparse methods only calculate the disparity for interesting features, which decreases the computational cost. However, this would leave large gaps in the disparity image, which are unwanted. Therefore, a local dense stereo matching algorithm will be used to create the disparity image. Possible stereo-matching algorithm candidates can be compared by their run-time, used platforms, performance and code availability from common benchmarks KITTI [Geiger et al., 2012] and Middlebury [Scharstein and Szeliski, 2002]. The most promising candidates are Block Matching (BM) [Scharstein and Szeliski, 2002] and Semi-Global Block Matching (SGBM) [Hirschmuller, 2005], due to their fast execution time and performance in outdoor environments [Lyrakis, 2019]. In addition, it was concluded that BM is preferred over SGBM, since it is faster and does not have the tendency to fill the sky with incorrect large disparity values [Lyrakis, 2019].

Block Matching can be performed using various metrics for similarity, pre-/post-filters and block sizes. The Sum of Absolute Differences (SAD) consists of relatively simple calculations, resulting in a fast algorithm [Patil et al., 2013]. Filters are used to delete outliers, which can be caused by occlusions or untextured or repetitive image regions. These filters will be discussed in more detail in subsection 3.1.

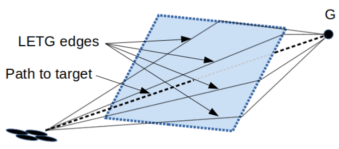
2.2 Avoidance

After the obstacles have been observed by the stereo camera, a path needs to be planned around them for avoidance. Path planning can be done by using either map-based or mapless methods. Map-based methods either start with a complete map of the environment (global map), or build them using sensors (local map), here referred to as map-building [Elmokadem and Savkin, 2021]. Mapless methods do not utilize this previously gathered information and show reflex-like behavior based on current sensor data [Elmokadem and Savkin, 2021]. Since complete maps of outdoor environments are hardly ever available, map-based approaches which require a global map are unrealistic for outdoor path planning.

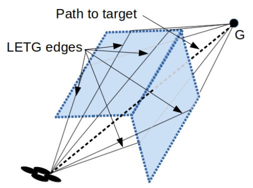
There is a trade-off in the decision for a map-building or mapless method. The objective of this research is to minimize deviation from the nominal path using limited resources. Map-building methods require more computational resources and memory, whereas mapless methods make reactive decisions using a relatively small amount of processing of the sensor data [Elmokadem and Savkin, 2021]. However, when stuck in a local minima, a mapless method does not use any information of previously sensed obstacles. It needs to scan the obstacles again and it could fail to identify an escape path already known to the map-building method. Hence, map-building methods could lead to more optimal paths. Nevertheless, only a small amount of dead ends is expected in outdoor environments, since obstacles can also be avoided by passing over them. Therefore, a mapless approach is chosen to limit computational resources.

Bug algorithms are computationally cheap mapless path planning algorithms, due to their simple reasoning in finding a way to the goal. For example, Bug2 [Lumelsky and Stepanov, 1986] draws a main line, or *M-line*, from start- to goal position and follows this line until an obstacle is encountered. The obstacle boundary is then followed until the *M-line* is met, and the motion to the goal is resumed. Wedgebug [Laubach and Burdick, 1999] is of interest because it uses a range sensor with a small FoV, and we also have limited FoV vision. It scans a 'wedge' to see if the goal is safe, or if an intermediate waypoint needs to be set. In case there is no solution inside the wedge, it will scan an adjacent wedge to obtain more information. This algorithm was used to move a planetary rover horizontally, i.e. in 2D. Extending it to 3D is necessary to utilize the drone's ability to move vertically. 3DBug [Kamon et al., 1996] is a bug algorithm that extends path planning to 3D by constructing a Local-ϵ-Tangent-Graph (LETG) which discretises the obstacle boundary. This is necessary since any obstacle boundary is a line with an infinite number of points. 3DBug assigns a finite number of points to the boundary and creates the LETG, which consists of the lines connecting the current- and the goal position to these waypoints, as shown in Figure 2. Figure 2a shows an example for a 2D plane and Figure 2b for a concave obstacle.

A reactive approach requires each disparity image to be used individually for path planning. Since the assumption that the robot can be modeled as a point robot does not hold, a safety margin needs to be included. This can be done with low computational costs and little memory by expanding the obstacles in disparity space to construct the configuration space (C-space) [Matthies et al., 2014], as presented in Figure 3. Each pixel in the disparity image is translated into its world coordinates and a sphere of a predefined expansion



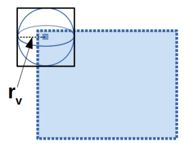
(a) 2D plane blocking the path



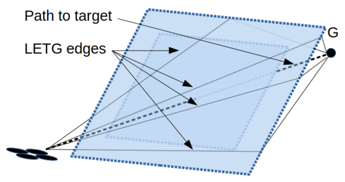
(b) Concave obstacle blocking the path

Figure 2: LETG edges used by 3DBug for an obstacle blocking the drone's path to goal, represented by the thicker line.

radius is drawn around it. An example for one pixel is shown in Figure 3a. Next, a rectangle that hides the sphere behind it is drawn just in front of the sphere, to simplify calculations. The area captured by each rectangle obtains a depth value equal to the pixel's original depth value minus the expansion radius. Using the C-space, the drone can be modeled as a point and it can safely choose an obstacle-free pixel in the disparity image, as shown in Figure 3b.



(a) Obstacle expansion for one pixel. Draw a sphere with expansion radius r_v and capture in a rectangle.



(b) C-space after obstacle expansion

Figure 3: Obstacle expansion in disparity space using expansion radius r_v, creating the drone's C-space.

3DBug makes two assumptions that do not hold: The range sensor has perfect readings and all obstacles can be modeled as polyhedral obstacles. The stereo camera will not provide perfect readings, which makes it difficult to model the obstacles as polyhedral. As a solution, a boundary tracing algorithm can be used to find the obstacle contours in the disparity image. The Moore-Neighbor tracing algorithm is often used, since it is fast and easy to implement [Reddy et al., 2012]. After creating the contour, the point that represents the locally shortest path can be calculated. Since boundary tracing could identify the (unsafe) concave edge in Figure 2b as safe, sudden points are searched within the image. These are defined in Pointbug [Buniyamin et al., 2011] as large sudden changes in depth readings. Hence, the sudden points will ensure that the drone chooses a waypoint around the obstacle instead of into it.

If the drone is stuck in a local minimum, it needs to apply a strategy to escape. Wedgebug halts the robot and scans additional wedges to find an intermediate waypoint [Laubach and Burdick, 1999] and goes into boundary-following mode. Here, the robot skirts the contour of the obstacle and leaves the boundary whenever the goal direction is free of obstacles. Other escaping strategies are proposed by potential field methods, for example introducing virtual obstacles that push the drone away [Lee and Park, 2003] or using rotational forces to steer around the obstacle [Sfeir et al., 2011]. However, the literature focuses on solutions for 2D problems, and 3D mapless obstacle avoidance algorithms tend not to mention they are getting stuck in local minima [e.g. Lee et al., 2021 and Oleynikova et al., 2015]. Due to the limited FoV of stereo vision also in the vertical direction, situations in which the drone cannot instantly see a vertical escape route can definitely occur.

3 METHOD

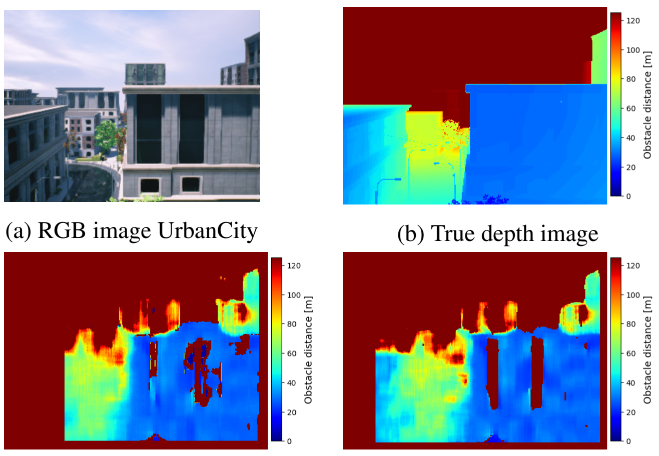
The method section first discusses the sensing, followed by the intermediate waypoint selection method and the finite-state machines. The latter is divided into the description of the Wedgebug states, changes in these states and the additional states for avoiding and escaping local minima.

3.1 Sensing

The rectified stereo image pair is fed to the stereo matching algorithm to generate the disparity image. The depth in meters can be obtained by multiplying the stereo base by the focal length, and dividing by the disparity in pixels. Figure 4 shows the result of the stereo BM algorithm using SAD. The RGB image in Figure 4a is taken from the simulated environment UrbanCity[1]. Its corresponding true depth image is shown in Figure 4b. The generated depth image is shown in Figure 4c shows the result using BM's default parameters. To improve the result, a grid search is performed to find the parameter combination giving the best disparity images. For this, a dataset of 100 images with ground truth depth values is generated from the open-source simulation software AirSim[2]. To quantitatively compare the performance of the parameter combination, each disparity image was evaluated for completeness, accuracy and noise. The parameters are for block size, texture filter, uniqueness ratio filter, speckle filter, left-right consistency check. An extra filter has been added in an attempt to further reduce noise: morphological opening. The generated depth image using these optimized parameters is shown in Figure 4d.

The next step is to expand the obstacles in disparity space, to create the C-space. As suggested by the authors, look-up tables for obstacle expansion coefficients are generated pre-flight for efficiency [Matthies et al., 2014]. The method is shown in Figure 5. Figure 5a is a small cutout of the optimized parameters' depth image showing two pixels as an example of obstacle expansion. The width of r_v has to be at least half the width of the drone to include a large enough safety margin. Expanding every pixel of an image with a pixel resolution of 240x320, results in the C-space shown in Figure 5b. To further improve performance, pixels are only

[1]https://www.unrealengine.com/marketplace/en-US/product/urban-city
[2]https://microsoft.github.io/AirSim/



(a) RGB image UrbanCity (b) True depth image

(c) Depth image using BM with default parameters (d) Depth image using BM with optimized parameters

Figure 4: True- and stereo Block Matching depth images, belonging to an RGB image from simulated environment UrbanCity.

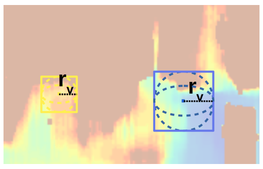
expanded if their depth value is below a threshold, and the images are downscaled. In the example of Figure 5c, a depth threshold of 30 meters is used, speeding up the runtime by a factor 2.5. Figure 5d shows the image downscaled by a factor 100, yielding a 24x32 resolution, and is 110 times faster than the complete C-space. The thresholded and downscaled image shown in Figure 5e is 210 times faster than the complete C-space, with satisfactory results. Figure 5f shows what happens to the C-space if the BM parameters are not optimized: the noisy pixels on the nearby building cover the entire FoV after obstacle expansion, making path planning impossible.

The best parameters for a dataset are not necessarily the best parameters for each individual image. For example, some images could have highly repetitive textures and would need different values for the uniqueness ratio parameter. Therefore, if the drone is not able to move due to poor stereo matching, Frustumbug will change the uniqueness ratio temporarily, in an effort to escape the current location.

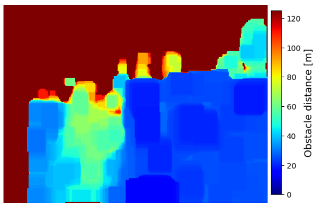
3.2 Waypoint selection

Wedgebug and 3DBug are both finite-state machines (FSM). They set intermediate waypoints to avoid obstacles, and move to the goal position when possible. The basis of the avoidance strategy is shaped by Wedgebug.

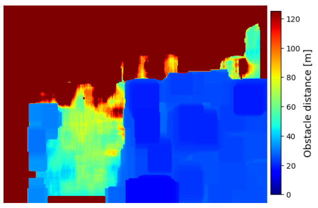
The novel waypoint selection method, shown in Figure 6, finds the locally best waypoint based on its sensor data. The goal- or current waypoint pixel must always be located inside the FoV, else the drone will hover and scan. The goal pixel is shown in white in the middle of each image. In this example, the goal is blocked by an obstacle at around 30 meters and a waypoint with a clearance of at least 35 meters will be selected. The first step is to identify the 'safe' pixels, as shown by the white mask in Figure 6a. Second, the mathe-



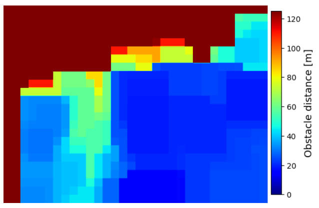
(a) Expansion radius r_v



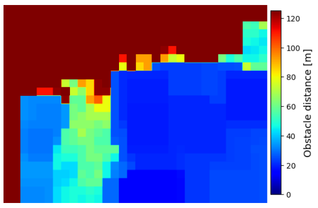
(b) Complete C-space



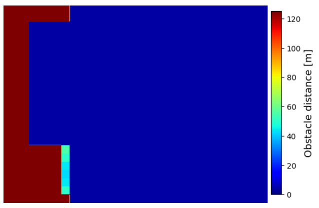
(c) Thresholded C-space (2.5x faster)



(d) Down-scaled C-space (110x faster)



(e) Thresholded and Down-scaled C-space (210x faster)



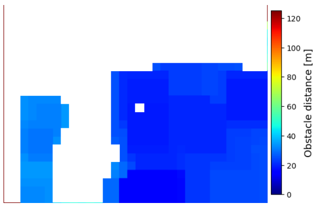
(f) Thresholded and Down-scaled C-space default parameters

Figure 5: Comparing C-space efficiency measures results

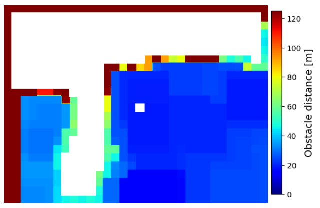
matical erosion of this mask is performed, yielding the eroded mask in Figure 6b. Since the obstacle expansion in disparity space is extremely sensitive to noise, as shown in Figure 5f, its expansion radius can not be too large. The mathematical erosion is performed to ensure a large enough safety margin, in a noise-robust manner. Third, the edge of the eroded mask is calculated, shown in Figure 6c. To get the locally shortest path, the obstacle needs to be avoided with the smallest deviation possible. Hence, the best waypoint will be located on the edge.

Since the Moore-Neighbor Tracing Algorithm requires a starting point for each boundary it traces, the novel method is preferred because it is able to find multiple safe areas. In addition, it makes it simpler to include an extra safety margin. Since a mask also makes no distinction between waypoints around or into concave obstacles, the fourth step is to look for sudden points. These are defined as pixels that have neighboring pixels have a change larger than 20% in their depth value, and are shown in Figure 6d.

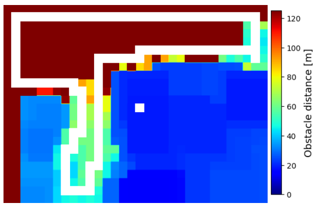
The last step is to find the best waypoint according to a cost function. For this, the obstacle boundary is discretized to pixel resolution: each pixel indicates a new possible waypoint. The cost function minimizes the pixel distance to the sudden points within the eroded mask (if they exist) and the goal pixel, shown in Figure 6e. Wedgebug places the waypoint at a finite sensor range [Laubach and Burdick, 1999]. Frustumbug places the waypoint beside the obstacle it is avoiding, to allow for goal scanning upon arrival.



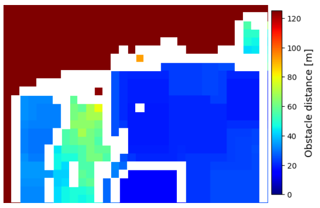
(a) Safe pixels mask



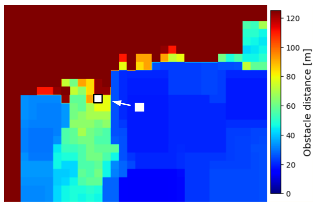
(b) Eroded mask



(c) Eroded mask edge



(d) Sudden points



(e) Selected waypoint

Figure 6: Waypoint selection procedure

3.3 *Finite-state machine - Wedgebug states*

Similar to Wedgebug and 3DBug, Frustumbug is also a finite-state machine. A simplified version is shown in Figure 7. The Wedgebug FSM is shown by the blue, green and yellow shapes, which are connected by solid lines. Frustumbug expands the FSM by adding reversing, climbing and descending states, indicated by the dashed grey lines. These can be activated from motion to goal, motion to waypoint or boundary following. The states with a solid boundary indicate a moving robot, whereas the dotted boundaries indicate a stationary robot, used for scanning. For the drone, stationary means hovering.

Wedgebug uses state ***Motion to Goal*** (*MtG*) to move the robot directly towards the goal. If the goal is reached, the algorithm halts. If an obstacle is detected, a new waypoint will be searched using the method described above. If found, state *MtG* transitions to state ***Motion to Waypoint*** (*MtW*). If not, it transitions to state ***Scanning Waypoint*** (*SW*). State *MtW* moves the robot to the waypoint and checks if state *MtG* is possible when the waypoint is reached. If the path to the waypoint becomes unsafe because of a previously unseen obstacle, a new waypoint is searched. If this waypoint is not found, state *MtW* transitions to state *SW*. In other words, state *SW* is activated when there are no safe waypoints visible in the current wedge. The robot halts and scans additional wedges to find a new waypoint that can still decrease its distance to the goal. If no such waypoint can be found, state *SW* transitions to state ***Scanning Boundary*** (*SB*). State *SB* keeps scanning additional wedges until a new waypoint is found. If none are found, the goal is deemed unreachable. Otherwise, state *SB* transitions to state ***Boundary Following*** (*BF*), which moves the robot around the obstacle boundary until a transition to state *MtG* is possible, or until there exists a leaving point: a waypoint which has a smaller distance to the goal than the previously visited coordinates, making a transition to state *MtW* possible [Laubach and Burdick, 1999]. The algorithm stores the direction in which the obstacle boundary is being followed (CW/CCW) to prevent backtracking. If the goal is reachable, Wedgebug guarantees global convergence [Laubach and Burdick, 1999].

3.4 Finite-state machine - Changes to Wedgebug states

Since Frustumbug obtains depth information from imperfect sensor readings, it is not able to identify independent obstacle boundaries. In other words, it is not sure whether it has been following the boundary of the same obstacle, or if it has jumped to different ones (e.g. it is hard to identify which branch belongs to which tree in a dense forest). This causes Frustumbug to lose its global convergence guarantee. Moreover, if an unseen obstacle appears within a short time after transitioning from state *BF* into state *MtW*, it cannot be said with certainty if this is the same obstacle whose boundary it was just following, or if it is a new obstacle. If it is the same, it should follow the boundary in the same direction (CW/CCW). Therefore, the positive direction around an obstacle is remembered until a distance is traveled that exceeds a threshold, currently implemented as 20 meters.

Frustumbug is designed for a fixed stereo camera, hence it can not rotate its sensors like Wedgebug. Whenever it has to scan an extra wedge, it will rotate the drone around the vertical axis (yaw). To minimize the chance of crashing into obstacles while the camera is facing away from the direction of motion, this is only performed when hovering stationary. If the drone is moving, it has to come to a stop first. To transition from state *MtW* to state *MtG*, the drone first needs to stop and scan the goal. This is done via the state ***Scanning Goal*** (*SG*). Furthermore, it required a design change for state *BF*, since it cannot scan the goal direction while moving along the boundary. Frustumbug divides state *BF* into two states: ***Boundary Following - Waypoint*** (*BFW*) and ***Boundary Following - Turning*** (*BFT*). When state *SB* has found a waypoint, it transitions to state *BFW*, which moves the drone to the waypoint. Upon arrival, there is a transition to state *BFT*, which rotates the drone to scan the goal and/or the obstacle boundary to plan a new waypoint. The distance between the waypoints results from a trade-off between stopping too often and missing opportunities where the goal direction was safe. Unfortunately, states *BFW* and *BFT* can lead to long paths if they miss safe goal direction opportunities and the continuous

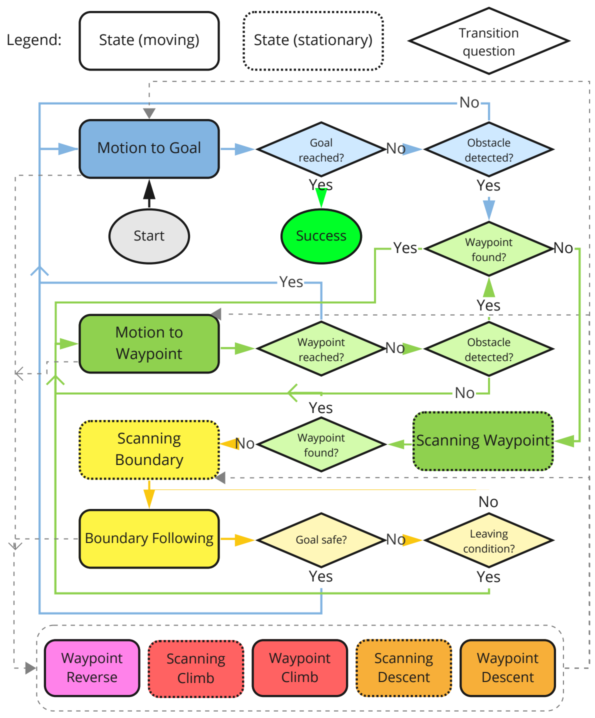


Figure 7: Simplified Frustumbug FSM

stopping, rotating and going can be time-demanding.

To decrease the frequency in which states *BFW* and *BFT* are activated, state *SW* is allowed to scan for waypoints that do not necessarily decrease the distance to goal, as long as they are within a 180 degrees FoV when facing the goal. Since Frustumbug already lost its global convergence guarantee due to being unable to identify independent obstacle boundaries, this rule is implemented to improve the overall performance. Further improvements are made by allowing state *SW* to always scan a left wedge and a right wedge, unless a fixed direction around the obstacle has been specified. This is implemented to prevent missing out on good opportunities to avoid the obstacle. Wedgebug would stop scanning as soon as it finds a waypoint in the left or right wedge [Laubach and Burdick, 1999].

To prevent looping around the same obstacle, Wedgebug has implemented a loop detection: when a location on the obstacle boundary is visited twice, a loop is detected [Laubach and Burdick, 1999]. A two-dimensional loop might not be found in a three-dimensional environment, since the shape of an object is not always constant over height, and the altitude of the drone can change during state *BFW*. Since Frustumbug can not clearly distinguish independent obstacle boundaries, it is possible that it will follow the boundary of multiple obstacles. Hence, even if a loop is found, the maneuver can be time-consuming. Therefore, the state *MtG* is activated by state *BFT* once the *M-line* is crossed. Instead of using the *M-line* from start- to goal position, Frustumbug defines this line as the line between the start of the state *SB* and the goal position, since the *M*-line does not play a role in the other states.

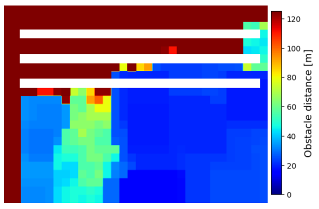
3.5 Finite-state machine - Additional states

To further decrease the frequency in which states *BFW* and *BFT* are activated, Frustumbug introduces the possibility to reverse the drone backward along previously visited GPS points using state ***Waypoint Reverse*** (*WR*). This state looks for escape points that would steer the drone around the obstacle in an earlier stage, using different parameter settings. Perhaps it did not see the obstacle before due to a stereo mismatch. Just in case it is mismatched again, the obstacle location is remembered as a single pixel, which is redrawn in the incoming images while reversing. One pixel is enough since it will be expanded in disparity space to create the C-space. State *WR* is only activated if the previously visited GPS coordinates are located on a straight line, since the drone is going there blindly and turning radii can be different than on the way there.

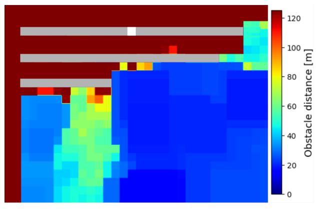
Next, a distinction has been made regarding obstacle size when reaching local minima: the C-space could be blocked by a farther away large obstacle (note that these pixels are still unsafe) or a nearby smaller obstacle which caused the obstacle expansion to cover the entire FoV, similar to what happened in Figure 5f. In case of a small obstacle, it is preferred to reverse and avoid using state *WR*, or to find a new waypoint using state *SW*. However, for a large obstacle, a new waypoint may not be found by states *WR* or *SW*. Instead, the altitude is increased in an effort to avoid the obstacle by going over it.

The process for selecting the best waypoint for climbing is shown in Figure 8. Due to the limited FoV it is not possible to move straight up, hence the drone will climb using a small flight path angle. To limit the required memory and computation when scanning its surroundings, only three pixel rows are checked, shown in white in Figure 8a. In the 32x24 image, these are rows 3, 6 and 9. The waypoint will not be selected in the top rows (0, 1 and 2), since it would move out of FoV quickly when the drone pitches forward to move, and obstacle presence could not be checked. For each white row, the best pixel's column is stored if the number of safe pixels exceeds the threshold for a safe climb. The best pixel's column is the column with the largest distance from any obstacle in that row. Figure 8b shows the safe pixels in grey, as well as the best pixel in white. If a safe pixel exists in the highest row, it is chosen first, followed by the middle row and then the bottom row. Preference is given to the climbing directions that are closer to the goal direction.

The goal direction is always scanned first for a possible waypoint to climb. If not found, state ***Scanning Climb*** (*SC*) is activated. It will first scan the 180 degrees in the goal direction to find a suitable climbing waypoint. If no waypoints are found, it will scan the remaining 180 degrees. If a way-



(a) The three scanned rows in white



(b) Waypoint for the quickest and safest climb in white. Grey shows obstacle-free waypoints

Figure 8: Method for selecting the best waypoint for a climbing maneuver

point is found, state ***Waypoint Climb*** (*WC*) will move the drone towards this waypoint. Else, state *SB* is activated. A climbing maneuver always consists of at least 2 segments to minimize the deviation from the nominal path: halfway the desired height difference, a new waypoint will be searched. It is possible that Frustumbug uses more than 2 segments to obtain the desired height difference if a previously unseen obstacle blocks the waypoint in state *WC*. A new segment is then added, using the method described above.

The climbing maneuver is created to allow the drone to avoid obstacles by flying over them, followed by a descending maneuver. Therefore, the drone is instructed to keep flying at a higher altitude until the obstacle is passed. Since individual obstacle boundaries can not be distinguished, this is implemented as a minimum distance to be flown toward the goal at a higher altitude. Once arrived, the goal direction is always scanned first for a possible waypoint to descend. If not found, state ***Scanning Descent Forwards*** (*SDF*) is activated. The method is similar to selecting climbing waypoints, but rows 14, 17 and 20 are scanned for descent.

State *SDF* only scans the 180 degrees FoV towards the goal. This state is normally used after the obstacle has been passed because descending backward could send the drone back to the wrong side of the obstacle. If no waypoint for descent is found, the drone will keep flying towards the goal at the same altitude for a few meters and transition to state *SDF* again. Else, state ***Waypoint Descent*** (*WD*) will move the drone towards this waypoint. Upon arrival, the goal direction will be scanned to see if the goal is inside the FoV. If not, state ***Scanning Descent Backwards*** (*SDB*) will be activated, which will scan the 180 degrees in the opposite direction of the goal. If a waypoint exists here, state *WD* is activated to execute the path. Else, state *SDF* is activated, which will try to find a waypoint to descend the drone toward the goal.

In case the drone is located nearby and above the goal, or if state *BFT* noticed that the goal is located below its FoV, it has no clear preference between states *SDF* or *SDB*. Hence, state ***Scanning Descent Either*** (*SDE*) is designed such that it will first scan in the goal direction. If no waypoint is found, it will continue to scan until a waypoint is found or until a full revolution is completed, similar to what state *SC* does for climbing. A note to this: if the drone realizes it is above an obstacle (e.g. a roof), it will keep moving towards the goal until this obstacle is passed. It attempts to identify this situation by checking if the pixels blocking the goal have a roughly linearly increasing depth value since this is usually the case when above buildings or large trees.

In total, the drone can adapt to 14 different states on its way to the goal position. We are aware that these FSMs can come across as quite elaborate. However, in complex environments, there are many possible failure modes that need to be accounted for, and we would like to present the states in detail to keep a clear overview of the drone's behavior during flight. Note that each path planning decision can be changed easily, allowing for quick user-preferred changes or upgrades.

4 Experimental Results

The proposed algorithm has been tested both in simulation and on a real drone. First, results from simulation environments UrbanCity and Forest are presented, followed by the real-world results. A number of start- and goal positions are generated for both UrbanCity and Forest environments. The generated paths are stored, from which the success rate and the path length can be obtained.

4.1 Simulation UrbanCity

UrbanCity is an environment built in Unreal Engine 4. Open source project UnrealCV included several commands to interact with the environment. The environment simulates an urban city corner and contains buildings, trees, roads and street signs. Start- and goal positions are chosen throughout the environment: 11 on ground level and 3 on top of buildings. These 14 locations yield 182 possible paths to be traveled. Since UnrealCV is only able to set static camera poses, drone dynamics are not included. To navigate through the environment, the camera location is moved by 1 meter in the direction of the waypoint each time. Only the yaw angle is changed, not pitch nor roll.

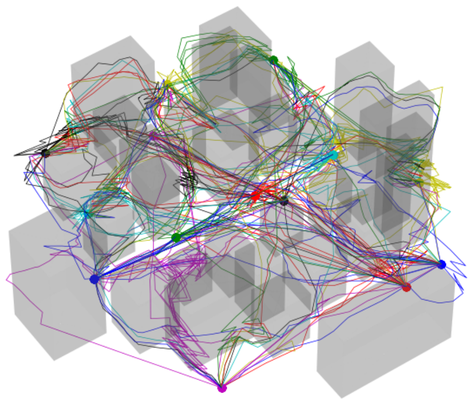
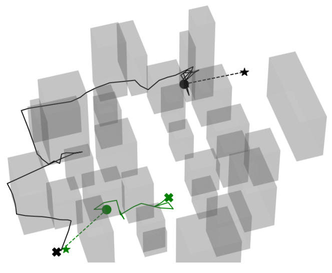
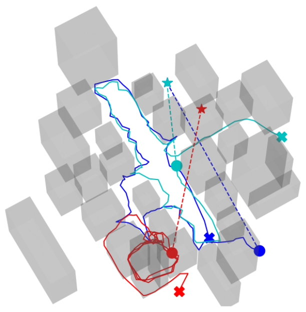


Figure 9: All successful paths in UrbanCity. The paths are color-coded similar to their goal position.

In total, 177 out of 182 goals were reached successfully, leading to a success rate of 97.3 %. Figure 9 shows all the successful paths. Each path has the same color as its goal, however since there are only 7 colors for 14 goals, every color appears twice. The 5 unsuccessful paths are shown in Figure 10, of which the first two are shown in Figure 10a. Here, the crosses represent the starting points, the dots are where it failed and the stars are the goals. The black line used state *WC* to avoid an obstacle at a higher altitude near the end. Upon arrival, the goal was still inside FoV near the bottom, but was blocked by the top of the building, hence state *MtW* was activated. A small distance later, the goal went out of FoV and state SDE was activated, which found a waypoint in the direction it just came from: in the front of the building. The drone got stuck in this loop until a timeout was reached. The green line collided with a building, due to a temporary change in value for the uniqueness ratio parameter, as explained in the last paragraph of subsection 3.1. After a turn, the drone was facing a different obstacle, which could not be detected using the temporarily increased uniqueness ratio. However, without this change in the uniqueness ratio parameter, dozens of simulations got stuck.



(a) Paths 1 and 2

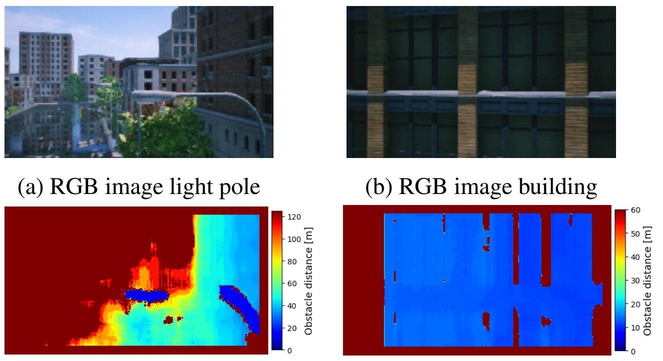


(b) Paths 3, 4 and 5

Figure 10: All unsuccessful paths in UrbanCity. A cross represents a starting point, a star represents the goal and the dot is where the simulation failed.

The three other unsuccessful flights, shown in Figure 10b, all ended in state *BFT* or *BFW* at the timeout. The blue and cyan lines could not find a path in between the buildings and continued to follow the boundary of a growing group of obstacles. The red line initially starts boundary following almost from the start, but gets stuck in a loop around one of the higher buildings. All three did not cross their respective *M*-lines. They initially did not start climbing because they were looking into small alleys, meaning that the minimum percentage of nearby pixels was not reached.

In subsection 3.5 a distinction between small and large obstacles was made, which would both have the entire FoV covered after obstacle expansion. Figure 11 shows an example to explain both possibilities using on-board data. Figure 11a shows a nearby light pole and Figure 11b shows a farther away building. This is supported by their depth maps:



(a) RGB image light pole (b) RGB image building

(c) Depth image light pole (d) Depth image building

Figure 11: Examples of nearby (light pole) and farther away (building) obstacles, both blocking the C-space. The light pole would trigger a transition to either state *SW* or *WR*, depending on the current state. The building would trigger a transition to state *SC*.

Figure 11c shows only a few nearby pixels in blue, whereas Figure 11d shows a large obstacle. After obstacle expansion, the nearby pixels cover the complete FoV for both. To transition to state *SC*, the percentage of nearby pixels in the depth image needs to exceed a threshold (currently 80 percent), else there will be a transition to either *SW* or *WR*, depending on the current state. For Figure 11a, state *SW* was activated, because the camera had just turned and there was no straight line segment, so the transition to state *WR* was not allowed. For Figure 11b, state *MtG* transitioned into state *SC*.

The path lengths are compared to their default path length. Note that the default path is not equal to the globally optimal path. Since it is complicated to get the globally shortest path out of UrbanCity, the default path length is defined by the path length that is obtained if the drone would simply climb high enough, fly over all obstacles, and descend to the goal, using the FoV restricted climbing maneuver. On average the path lengths are 75.8 % of their default path lengths, meaning that Frustumbug has found an average shortcut of 24.2 %. Results are shown in Figure 12. All 6 flights exceeding 115 % have activated states BFW and BFT during flight.

4.2 *Simulation Forest*

The open-source simulator AirSim is developed by Microsoft and contains an outdoor environment called Forest. It includes many trees, branches, bushes, rocks and hills, and is shown in Figure 13. The green dot indicates the starting position, and the 8 orange dots indicate the goal positions. Some of the goals are located near trees, but all are reachable. The paths are flown at 3 different altitudes, and from low to high altitude and vice versa, resulting in 40 paths. The area with the purple dots consists of fewer trees and more hills, hence it is used to test Frustumbug's ability to deal with ground elevation changes. In total, 43 paths are generated. The arrows indicate that the waypoint location is

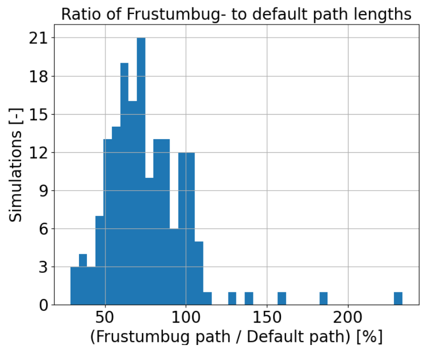


Figure 12: Frustumbug's path lengths histogram

outside of the image. Drone dynamics are included in this simulation, making it more representative for outdoor flight.

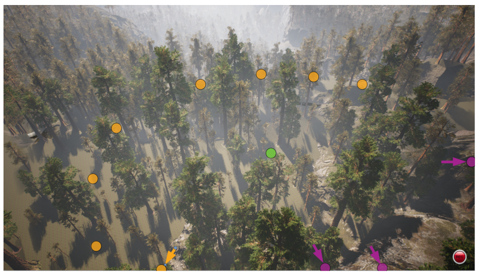
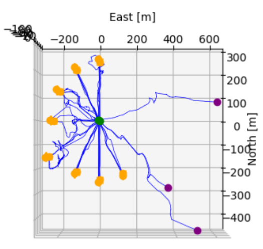


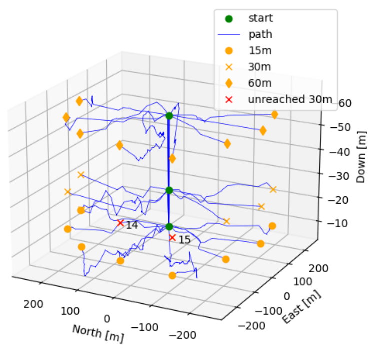
Figure 13: Forest environment. The green dot represents the start position, the orange and purple dots show all the goal positions. Arrows indicate they are off the map.

39 out of 43 goals were reached successfully, leading to a success rate of 90.7 %. Figure 14 shows all the successful paths and Figure 15 shows the 4 unsuccessful paths. Figure 14a starts with the top view. Note that the left side of this plot is the most cluttered part of the environment, with a large number of trees located close together. Figure 14b shows the 24 paths where start- and goal positions have the same altitude. Two goals (14 and 15) are not reached, both in the cluttered area. Figure 14c shows the 16 paths that go from low to high altitude and vice versa. Here, two goals (34 and 38) were not reached, of which one was located in the cluttered area. Figure 14d shows the three waypoints far into the forest, in the area containing hills. All three were reached by the proposed algorithm.

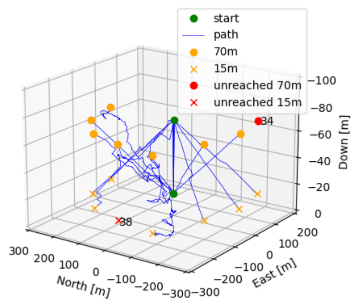
Ideally, the generated path lengths are compared to the globally shortest path lengths. However, calculating these would require accurate obstacle data, which is not available. Furthermore, the climbing and descending method presented in UrbanCity would yield unrealistically long default paths, since the trees are generally not avoided by climbing over them. Therefore, the paths had to be analyzed qualitatively. It was found that any deviation it took from the nominal path



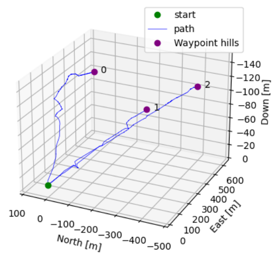
(a) Top view of all paths



(b) Paths at constant altitude



(c) Paths with changing altitude



(d) Paths into the hills

Figure 14: All successful flights in Forest

was supported by obstacle presence.

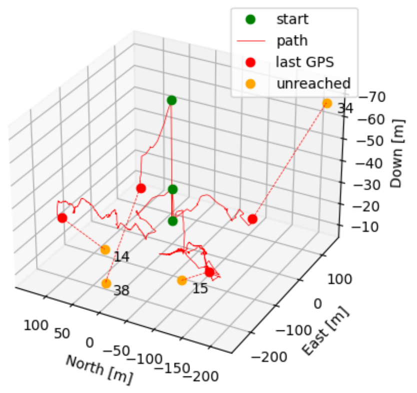


Figure 15: Four unsuccessful paths in Forest

The four unsuccessful paths are caused by three different reasons. The paths shown in Figure 15 belong to the red markers in Figure 14b and Figure 14c. Paths 34 and 38 both got stuck for the same reason: states *BFW* and *BFT* did not find an escape point in any of the wedges. Path 14 was working its way around the trees, but got stopped by a timeout. Path 15 transitioned many times from state *BFT* to state *MtW*, and from state *MtW* back to *SB* (followed by *BFW* and *BFT*). This was because the positive direction around the obstacle boundary was forgotten by the time the transition to *SB* happened. In other words, the threshold explained in subsection 3.4 was too low for this run, since it kept going around the same group of obstacles in different directions, until the timeout was reached.

4.3 *Real World*

For real-world testing, two JeVois[3] cameras were connected to one ARM-A7 processor, responsible for the navigation. The setup is shown in Figure 16, where the stereo camera is shielded in tin foil to block electromagnetic radiation.

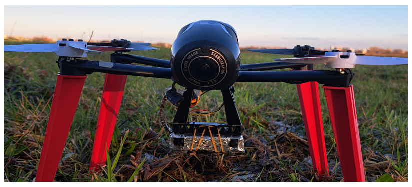


Figure 16: Experimental setup: JeVois stereo camera mounted to the bottom of the drone.

The testing is performed in the CyberZoo facility at the faculty or Aerospace Engineering at the TU Delft. The drone obtains accurate position and orientation estimates via the motion capture system Optitrack. In the experiment shown in Figure 17, obstacles are placed in a V-shape to create a local minimum.

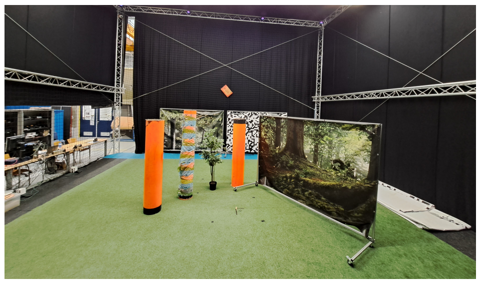
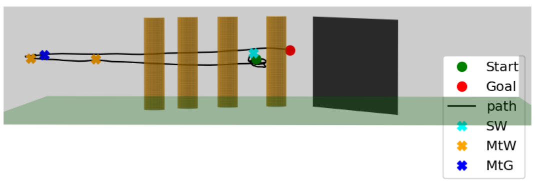


Figure 17: Real world testing environment: CyberZoo facility at the TU Delft. Obstacles are positioned to create a local minimum.

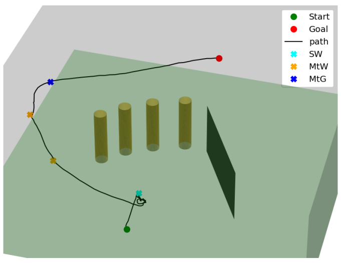
The results are shown in Figure 18, where Frustumbug has created a safe path around the obstacles. Figure 18a uses a low elevation angle to show that the height is approximately constant throughout the flight. Figure 18b shows a better perspective of the generated path. The path starts at the green dot, and the drone moves to the goal using state *MtG*. Once the obstacle is detected, the drone stops and hovers, because a waypoint is not found. State *SW* (cyan cross) starts scanning for a waypoint, and on the second left scan, a waypoint is found and the drone moves there. At both the orange crosses an obstacle (the CyberZoo edge) is detected and a new waypoint is found while in the state *MtW*. At the dark blue cross,

[3]http://jevois.org/

the waypoint is reached and state *MtG* moves the drone towards the goal. A waypoint above the obstacles was not found due to the limited vertical FoV.



(a) Low elevation angle



(b) High elevation angle

Figure 18: Result of a CyberZoo flight test. State *SW* finds a waypoint around the group of obstacles. State *MtW* steers clear of the CyberZoo edge. State *MtG* moves the drone towards the goal.

Another experiment, where the drone only had to avoid one obstacle, is shown in Figure 19. It starts at the green dot in state *MtG*, and the obstacle is detected at the orange cross. However, in this experiment, a waypoint is immediately found, hence no transition to state *SW* is necessary. State *MtW* moves the drone towards the waypoint. Upon arrival, the state transitions to *MtG* to complete the last segment to the goal. Note that these real-world tests do not test all the possible state transitions. Since the simulated environments have already proven that the algorithm can reach its goal through challenging environments, the main focus of the real-world tests is to show its feasibility on resource-limited systems.

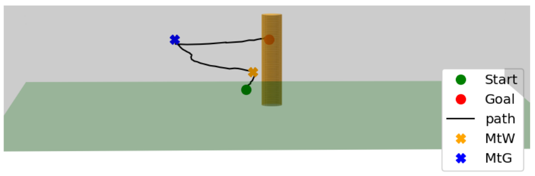


Figure 19: Result of a simple CyberZoo flight test. In state *MtG*, the obstacle is detected and a waypoint is found. State *MtW* moves the drone to the waypoint and a transition to state *MtG* follows.

5 Discussion

The results have shown that Frustumbug is able to reactively plan its path from start- to goal position with an acceptable path length, using limited computation and memory.

Previous research had various simplifications and limitations. 3DBug assumed that the range sensor has perfect readings over a 360 degrees FoV and that all obstacles could be modeled as polyhedral. In addition, it was only tested in simulation. Other studies on reactive path planning in 3D did not explicitly mention escaping local minima or used randomly generated sub-optimal paths to escape. Frustumbug has been demonstrated in complex, realistic environments. It has found solutions for the assumptions that do not hold in these environments, and escapes local minima by taking logical decisions. The strengths of the proposed algorithm are the high success rate and the small size, weight and power. Furthermore, every path-planning decision is made according to a large decision tree. This means that any choice made by the algorithm can be examined afterward and changed individually. Moreover, the stereo matching parameters can be altered mid-flight if the incoming image shows signs of low texture or highly repetitive texture.

The limitations of the algorithm come in twofold, which will be discussed along with a few recommendations.

First, the choice of using a stereo camera with a small FoV limits the drone's local environment information. For example, during boundary following, the drone has to stop often to scan extra wedges. Furthermore, it results in small climbing and descending angles in states *WC* and *WD*. Increasing the FoV is a possible solution, but comes with issues. Increasing the FoV while keeping the same image resolution will decrease the focal length, which decreases the largest depth that can be measured. The stereo base could be increased to counteract, but for small drones, this is not ideal. To keep a constant focal length, the image size needs to be increased, but this will increase the computation. Another solution is to allow separate rotation of the stereo camera, similar to Wedgebug. However, this would introduce more moving components that can fail and puts more requirements on drones that would like to use Frustumbug. A third solution would be the use of additional lightweight sensors, for example, an HC-SR04 SONAR. Since it has an extremely small FoV it could not be used for normal path planning, but it could scan to see if there is an obstacle above the drone. If not, the FoV limited climbing maneuver can be avoided. Pointing a stereo camera up would be excessive since detailed information about obstacles above the drone is not required. The SONAR requires only 15 mA (JeVois: 800 mA), but adds an extra weight of 8.7 grams per sensor [4].

Second, the c-space expansion could hide any escape points in narrow streets, or local areas with a high obstacle density. Future work could focus on running two instances of

[4]https://www.adafruit.com/product/3942

the c-space algorithm, with both a small and large expansion radius. The small expansion radius could be used temporarily to escape the situations described above. However, the larger expansion radius is the default for path planning, to keep a big enough safety margin from obstacles for changes caused by wind or GPS inaccuracy.

Further improvements can be found when looking at the results. The black path in Figure 10a failed because it did not identify that it was above an obstacle that needed to be passed, as described in subsection 3.5. In this particular flight, the drone was squeezed between two buildings of different heights, and no linearly increasing depth values were found in the direction of the goal. Future work could focus on designing an improved version of this function. The same rationale applies to the green path: it failed because the trick to ignore repetitive texture caused it to not recognize another building. Future work could use image characteristics to have a variable parameter set for the stereo-matching algorithm.

Looking at Figure 10b, improvements can be made in the decision for state transition. For example, state *SB* is activated when the obstacle is relatively small (and state *SW* could not find a waypoint), but once state *BFW* or *BFT* realizes that the obstacle is in fact quite large, the state should switch to state *SC* and not stay in *BFW* or *BFT*, as was seen in Figure 10b. The same figure also shows a red path getting stuck in a loop around the same building, hence implementing something equivalent to a 3D loop detection seems useful after all.

6 CONCLUSION

Frustumbug is a cheap, lightweight three-dimensional path-planning package for small drones. It reaches more than 90% of its goals in complex environments, is robust to noisy range sensor data and runs smoothly on a 20-gram stereo vision system with limited memory and computation. Furthermore, it has been extensively tested, both in simulation and real-world.

Our paper takes a step towards a publicly available path planning package. Since it does not rely on heavy processing power, it will be less demanding for the battery and onboard computer, increasing valuable flight time.

REFERENCES

[1] N Aswini, E Krishna Kumar, and SV Uma. Uav and obstacle sensing techniques–a perspective. *International Journal of Intelligent Unmanned Systems*, 2018.

[2] Larry Matthies, Roland Brockers, Yoshiaki Kuwata, and Stephan Weiss. Stereo vision-based obstacle avoidance for micro air vehicles using disparity space. In *2014 IEEE international conference on robotics and automation (ICRA)*, pages 3242–3249. IEEE, 2014.

[3] Junseok Lee, Xiangyu Wu, Seung Jae Lee, and Mark W Mueller. Autonomous flight through cluttered outdoor environments using a memoryless planner. In *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1131–1138. IEEE, 2021.

[4] Oualid Doukhi and Deok Jin Lee. Deep reinforcement learning for autonomous map-less navigation of a flying robot. *IEEE Access*, 10:82964–82976, 2022.

[5] Ricardo Bedin Grando, Junior Costa de Jesus, Victor Augusto Kich, Alisson Henrique Kolling, and Paulo Lilles Jorge Drews-Jr. Double critic deep reinforcement learning for mapless 3d navigation of unmanned aerial vehicles. *Journal of Intelligent & Robotic Systems*, 104 (2):1–14, 2022.

[6] Yang Yu, Wang Tingting, Chen Long, and Zhang Weiwei. Stereo vision based obstacle avoidance strategy for quadcopter uav. In *2018 Chinese Control And Decision Conference (CCDC)*, pages 490–494. IEEE, 2018.

[7] Sharon L Laubach and Joel W Burdick. An autonomous sensor-based path-planner for planetary microrovers. In *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No. 99CH36288C)*, volume 1, pages 347–354. IEEE, 1999.

[8] Ishay Kamon, Ehud Rivlin, and Elon Rimon. 3dbug: A three-dimensional range-sensor based globally convergent navigation algorithm. Technical report, Computer Science Department, Technion, 1996.

[9] Tom van Dijk. Self-supervised learning for visual obstacle avoidance. Technical report, Micro Air Vehicle Lab (MAVLab), TU Delft, mar 2020. Technical report.

[10] Kimberly McGuire, Guido De Croon, Christophe De Wagter, Karl Tuyls, and Hilbert Kappen. Efficient optical flow and stereo vision for velocity estimation and obstacle avoidance on an autonomous pocket drone. *IEEE Robotics and Automation Letters*, 2(2): 1070–1076, 2017.

[11] Hua Liu, Rui Wang, Yuanping Xia, and Xiaoming Zhang. Improved cost computation and adaptive shape guided filter for local stereo matching of low texture stereo images. *Applied Sciences*, 10(5):1869, 2020.

[12] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *2012 IEEE conference on computer vision and pattern recognition*, pages 3354–3361. IEEE, 2012.

[13] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International journal of computer vision*, 47(1):7–42, 2002.

[14] Heiko Hirschmuller. Accurate and efficient stereo processing by semi-global matching and mutual information. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 807–814. IEEE, 2005.

[15] Alexios Lyrakis. Low-cost stereo-based obstacle avoidance for small uavs using uncertainty maps. Master's thesis, Delft University of Technology, 2019.

[16] Suyog Patil, Joseph Simon Nadar, Jimit Gada, Siddhartha Motghare, and Sujath S Nair. Comparison of various stereo vision cost aggregation methods. *International Journal of Engineering and Innovative Technology*, 2(8):222–226, 2013.

[17] Taha Elmokadem and Andrey V Savkin. Towards fully autonomous uavs: A survey. *Sensors*, 21(18):6223, 2021.

[18] V. Lumelsky and A. Stepanov. Dynamic path planning for a mobile automaton with limited information on the environment. *IEEE Transactions on Automatic Control*, 31(11):1058–1063, 1986. doi: 10.1109/TAC.1986.1104175.

[19] P Rajashekar Reddy, V Amarnadh, and Mekala Bhaskar. Evaluation of stopping criterion in contour tracing algorithms. *International Journal of Computer Science and Information Technologies*, 3(3): 3888–3894, 2012.

[20] Norlida Buniyamin, W Wan Ngah, Nohaidda Sariff, Zainuddin Mohamad, et al. A simple local path planning algorithm for autonomous mobile robots. *International journal of systems applications, Engineering & development*, 5(2):151–159, 2011.

[21] Min Cheol Lee and Min Gyu Park. Artificial potential field based path planning for mobile robots using a virtual obstacle concept. In *Proceedings 2003 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM 2003)*, volume 2, pages 735–740. IEEE, 2003.

[22] Joe Sfeir, Maarouf Saad, and Hamadou Saliah-Hassane. An improved artificial potential field approach to real-time mobile robot path planning in an unknown environment. In *2011 IEEE international symposium on robotic and sensors environments (ROSE)*, pages 208–213. IEEE, 2011.

[23] Helen Oleynikova, Dominik Honegger, and Marc Pollefeys. Reactive avoidance using embedded stereo vision for mav flight. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 50–56. IEEE, 2015.