Real-Time, 3D Path Planning for UAVs

Jay Lanzafane
Boston University Mechanical Engineering
Jlanzafa@bu.edu

Abstract

The field of path planning is both heavily researched and utilized in current robotic systems. There is a distinct lack, however, in the development of 3D, real-time path planners. As the capabilities and involvement of robots in our lives continue to increase, they will rely more and more on autonomous control. Many of these systems will require path planning techniques falling into the realm of such 3D, real-time planners. We set out to explore this domain in more detail and create a solid foundation from which to analyze, compare, and develop these algorithms.

This paper will detail the creation of a testbed system capable of simulating 3D, real-time path planning problems with the capability to easily switch both the testing environments and algorithms. This requires the system to simulate the movement of a UAV through an arbitrary environment as well as the UAV's sensor readings, update the UAV's knowledge of the map based on these sensing capabilities, and then pass this information to the path planning algorithm. It tracks performance metrics and saves intermediate data throughout, allowing detailed analysis and the ability to examine each intermediate step of the process. Additionally, it provides powerful 3D visualization tools to not only view the environment and resulting path, but a step-by-step tool which renders the UAV's current position, the path it took to get there, its current knowledge of the map, and its intended path onward. This system provides a solid foundation to work from in order to test, evaluate, and compare current state of the art techniques and will hopefully aid in the development of future algorithms. Finally, we present preliminary results comparing an optimized version of the popular A algorithm against the purpose-built, 3D real-time path planning algorithm HD*.*

1. Introduction

UAVs have become incredibly capable and agile flyers. A quick look at the budding sport of drone racing will demonstrate UAVs navigating 3 dimensional obstacle courses at speeds relative to, and with the agility of, birds - one of natures most capable flyers. These drones, however, require human pilots to direct their flight. New research has led to control systems for UAVs which can follow similarly extreme trajectories, extending out to the limits of their physical capabilities [1]. Where birds, and drone pilots, have current autonomous systems beat is in their real-time path planning capabilities. If given a set of way points, the control algorithm in [1] could plan a trajectory from the starting way point through to the final way point. If operating within an unknown environment, however, these way points can not be known a priori and a 3D, real-time path planner is required to generate them. Current research in this area is not as well developed [2], [3], and autonomous UAVs are significantly slower at navigating an unknown, 3D obstacle field than birds or bats. The best example of high-speed flight through such an environment is seen in [3], and is a promising, though incomplete, start.

Real-time path planning is required for a large number of future applications - from drones aiding in search and rescue missions during natural disasters to the synergy of human-robot teams in manufacturing. Path planners are usually presented in the context of movement of a body through some environment, and in this project that is precisely the goal, but in the larger picture they can be abstracted to work on the manifolds which define the workspaces of many types of robots. As robots, and their autonomous roles within the world, become more and more advanced, they will increasingly need to be more aware and capable of reacting to their surroundings.

Path planning is, and has been, a heavily studied subject for many years. The simplest incarnation of the problem is the 2D, offline case - when the map is known in it's entirety. Solutions to this problem began back in 1956 with the formulation of Dijkstra's Algorithm. In 1968, the A* search algorithm was first described. A* is still an incredibly useful algorithm because it is both optimal and complete - it will always find the shortest path available and in the shortest number of executions possible. Implementations of A* can be found in a huge number of applications, from robotics, to image processing, and even in video games. Since it's conception, there have been a vast number

of different types of path planning algorithms developed. Suffice it to say that there are hundreds of approaches and techniques available, with different strengths, weaknesses, and intents. While the current body of work has the 2D, offline problem nailed down, there is considerably less development in the realm of 3D, real-time planners [2], [4]. This class of problem provides significant challenges not present in the 2D, offline case. First, the change to 3 dimensions both invalidates many types of planners and makes the search space exponentially larger. Meanwhile, the transition to real-time provides a number of challenges in and of itself. First and foremost is the constantly shifting nature of the environment - as more of the map is explored, or as the environment changes, previously planned paths need to be reassessed not only for fear of being invalidated by obstacles but for the emergence of new, more optimal paths opening up. Additionally, even when optimal planners are run in real-time, it is impossible to guarantee optimality, as the map is, by definition, not fully known. Each of these challenges has been explored individually, but typically not in conjunction with one another.

This combination of challenges presents a serious hurdle when it comes to creating fast and efficient algorithms. By their very nature, real-time planners often need to return results quickly in order for the processes they guide to proceed, whether that's a UAV navigating a dense forest or a robotic arm reacting to the changes in its environment. Having long computation times is extremely detrimental to these systems because their safe, maximum speed is limited by the computation time. As such, the effectiveness of autonomous systems which rely on path planners is inextricably tied to the performance of these algorithms.

This paper will detail the creation of a testbed system capable of simulating 3D, real-time path planning problems with the capability to easily switch both the testing environments and algorithms. This requires the system to simulate the movement of a UAV through an arbitrary environment, update the UAV's knowledge of the map based on its sensing capabilities, and then pass this information to the path planning algorithm. It tracks performance metrics and saves intermediate data throughout, allowing detailed analysis and the ability to examine each intermediate step of the process. Additionally, it provides powerful 3D visualization tools to not only view the environment and resulting path, but a step-by-step tool which renders the UAV's current position, the path it took to get there, its current knowledge of the map, and its intended path onward. This system provides a solid foundation to work from in order to test, evaluate, and compare current state of the art techniques and will hopefully aid in the development of future algorithms. Finally, we present preliminary results comparing an optimized version of the popular A* algorithm against the purpose-built, 3D real-time path planning algorithm HD*.

2. Implementation

2.1. Overview

In order to accurately simulate 3D, real-time path planning, the testbed will require a number of major components. First and foremost, it must have a consistent geometry for representing both the environment and the position of the UAV as it moves through that environment. As the UAV explores more of the map, the system must also update the UAV's knowledge of the map as it pertains to its ability to plan paths. Of course, it requires the implementation of path planning algorithms to generate the paths. Finally, visualization techniques are required to intuitively portray information about the paths, explored environment, and current state of the path.

The basic chain of processing is straightforward - the UAV plans a path according to its current knowledge of the map, takes a step along that path, updates its knowledge of the map when applicable, and then repeats until it reaches the goal. These major components of the system are discussed below.

2.2. Geometry

As stated, there are a significant number of approaches to path planning. This project is focused on techniques which operate on uniform grids, in particular the A* algorithm and its more advanced variants. It is important to define the properties of uniform grids and their characteristics, as they are essential to understanding the algorithms and, later, optimizing their performance.

A uniform grid is a representation of a workspace where the defined area has been segmented into uniformly sized elements. For the purpose of path planning, it is easiest to define them as unit, Cartesian grids, where each element is a unit square in 2D or a unit cube in 3D. As such, any position within the workspace can be represented by a vector in $\mathbb{N}^3$.

Each element, or cell, has an associated group of neighbors. These neighbors can be defined in a number of ways. In a 2D, Cartesian grid it is common to define them in one of two ways: a 4-connected neighborhood or an 8-connected neighborhood. In the 4-connected case, cells that share a common edge are defined to be neighbors. These amount to the adjacent cells in the cardinal directions. In the 8-connected neighborhood, cells which share even a single point are also included within the neighborhood. This addition adds adjacent cells in the ordinal directions, as seen in figure 1.

Within 3D Cartesian grids there are a few more possibilities. 6-connected neighborhoods are defined by cells that share a common face (paralleling a 4-connected neighborhood in 2D). 18-connected neighborhoods are defined by

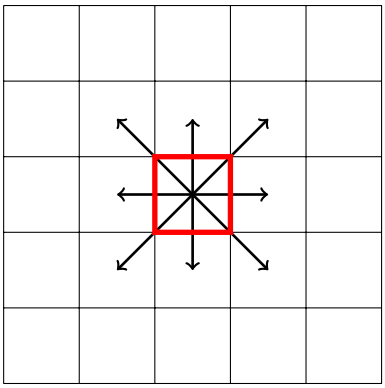

Figure 1. An 8-connected neighborhood in 2D. Neighbors are defined as any cell which touches an edge of the current cell (in the cardinal directions) as well as any cell which touches at just the corner point (in the ordinal directions).

cells that share a common edge (paralleling an 8-connected neighborhood in 2D). Finally, 26-connected neighborhoods are defined by cells that share a common point, opening up the 8 corners of the 3x3 cube centered on the starting node.

Movement across the grid is defined as a continuous series of cells which the mover will traverse on its way from the starting location to the destination. Any cell within this series is required to be a neighbor of the previous cell. This constraint limits the individual movements based on the type of neighborhood selected.

2.3. The Environment

As stated, one of the major goals of this testing system was the ability to quickly and easily change testing environments. To streamline the process, it is convenient to define a standard method for representing an environment. Given that the planner is working on a unit Cartesian grid, it makes sense that the environment should share that property. As such, the environments are defined as 3D matrices. Any cell that contains a value of 1 or greater is treated as obstructed, while values less than 1 are treated as unobstructed; resulting in a binary classification of the map.

To begin testing, a few baseline environments were generated. As this project was inspired by the graceful flight of birds through a forest, it seemed appropriate to generate environments which were representative of that particular challenge. The environments generated come from a random generator and are rough caricatures of forested environments with a few buildings. The size, density, and clutter are controlled by a few tuning parameters. While these generated environments do not comprise a complete test, they do provide a decent baseline for testing.

The first environment generated can be seen in figure 2. It is a 500x250x50 volume, totalling 6,250,000 nodes. The start and goal locations were chosen at points on opposite sides of the map in order to produce longer paths for testing, which can be seen later in figure 7.

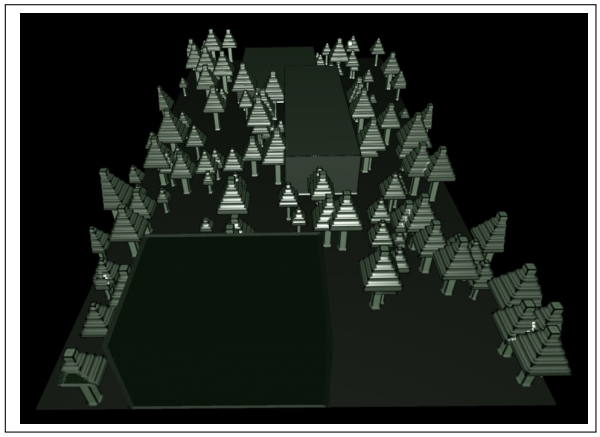

Figure 2. Environment 1, a relatively sparse environment with 3 short buildings

As another example, environment 3 can be seen in figure 3. This is a more densely cluttered environment and contains significantly taller buildings which can not be flown over. The heavy tree canopy is intended to hinder the options for vertical movement, as the UAVs starting position is just above the ground. This environment is 250x250x150, totalling 9,375,000 nodes.

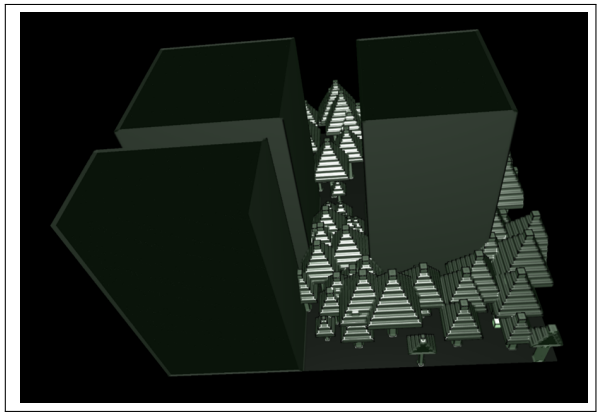

Figure 3. Environment 3, a dense tree canopy with tall buildings to heavily limit the effectiveness of vertical motion.

3. Algorithms

While a complete explanation and discussion of the algorithms is beyond the scope of this paper, an overarching explanation will be provided for the two primary algorithms. This should give enough background and intuition to understand the remainder of the paper and results. Citations are provided for much more detailed discussions of the algorithms, where appropriate.

3.1. The A* Algorithm

The A* algorithm is a heavily studied and utilized path planning algorithm. A wide range of resources exist which detail the machinery and nuances of the algorithm, such as [11], and as such only an introductory explanation will be provided here.

The algorithm requires a graph representing the environment, a start, and a goal location as inputs. In this particular case, the graph is assumed to be a Cartesian grid. The algorithm sequentially explores outward from the starting location towards the goal location by a process known as expansion. When a node is expanded, all of its neighboring nodes are initialized and added to the list of possible nodes to expand in the future, called the open list. When a node is initialized, a few values are computed. The first is the g-value, which is the distance traversed to reach that node from the starting node. Additionally, the heuristic, or h-value, is computed as the estimated distance of that node to the goal node. The h and g values are summed together to produce an F-score for each of the nodes added to the open list. The lower the F-score, the shorter the expected path to the goal. As such, the open list is sorted by the F-score so that the next node to be expanded will lie on the most promising path - which will have the lowest F-score. Nodes which have been expanded are taken off the open list and added to a closed list so to prevent them from being recomputed. Because the algorithm always follows the shortest path, it can return immediately upon finding the goal.

3.2. The HD* Algorithm

The HD* Algorithm is a combination of D* Lite, a variant of the D* Algorithm, and a Hierarchical planner. Again, full details of this algorithm can be seen in [2] and only a general overview will be covered here.

The first major component is the D* algorithm, which is a dynamic version of A*. D* was created to address the real-time planner problem, where the map is not known a priori and unexpected obstructions may be encountered in route. As explained, the algorithm will be rerun after every movement the robot takes. Fundamentally, D* attempts to reuse previous computation in an effort to speed up the path planning time. Since the starting location will change every time the robot moves, D* treats the destination as the starting node and the robot's position as the goal node. The intuition for this reversal is that the change in information is occurring locally around the robot as it uncovers more of the map. By planning from the goal to the robot, only the end of the path will be affected by those changes, versus the beginning of the path if planning from the robot to the goal. This allows a large portion of the path to be reused and thus reduces the total number of node expansions required. Note that this assumption is not a requirement, information of the environment can change anywhere within the map and the algorithm will still function properly. The performance advantage of D* will diminish, however, as less information can be reused.

In order to keep track of changes to the map, a new variable is added, called the right-hand-side (rhs) value. The rhs-value is similar to the g-value but is slightly more informed, as it considers the neighbors of the node as well - it looks ahead by one value. Anytime the rhs-value is not the same as the g-value for a given node, it is deemed inconsistent. Any node which is inconsistent is re-added to the open list in order to remedy the inconsistency. This process effectively handles the discovery of new obstacles and will re-plan the path accordingly - with the smallest alterations possible to retain path optimality.

The other component of HD* is the inclusion of a Hierarchical planner. The premise of hierarchical planners is simple, abstract the problem to a coarse grid, solve the rough path, and then refine the path on the level of the fine grid. Doing so will yield a significantly smaller search space for the coarse planning step and thus heavily speed up the process. The HPA* algorithm, Hierarchical Path-Finding A*, generates abstract levels of the original map, which is Level 0, by clustering nodes of the level below it (i.e. Level 1 is comprised of clusters of Level 0 nodes, and Level 2 is comprised of clusters of the Level 1 clusters, etc). It then uses a complex method of defining transition points between the clusters, which it uses to plan intra-cluster paths. This technique works particularly well in 2D, and can be read about in more detail in [10], but falls prey to the exponential change in complexity that the switch to 3D brings. As such, a slightly different clustering method is used within HD* and the transitions are dropped for another representation. Instead of using the borders of the clusters to define transition, HD* keeps track of the 3D cluster size. Neighbors of the cluster are then defined as the 26 neighbor nodes as defined previously, but scaled to a distance equal to the size of the cluster, as seen in in figure 4.

Finally, HD* does some path smoothing as part of the refinement process. This is done via a combination of line-of-sight checks and Catmull-Rom Splines. The details of this are not particularly important, however. It should also be noted, as a matter of completeness, that HD* uses D* Lite, which is not the same as D*. Despite the name, D* Lite is not actually based on the D* algorithm, but exhibits identical behavior. D* Lite is derived from LPA*, Lifelong planning A*. Suffice it to say that these algorithms, LPA*, D*, and D* Lite, though different in practice, share ideologies and common techniques to the level with which they have been described in this overview. Sven Koenig and Maxim Likhachev are the creators of LPA* and D* Lite, and much more in depth explanations can be found in [8] and [9], respectively.

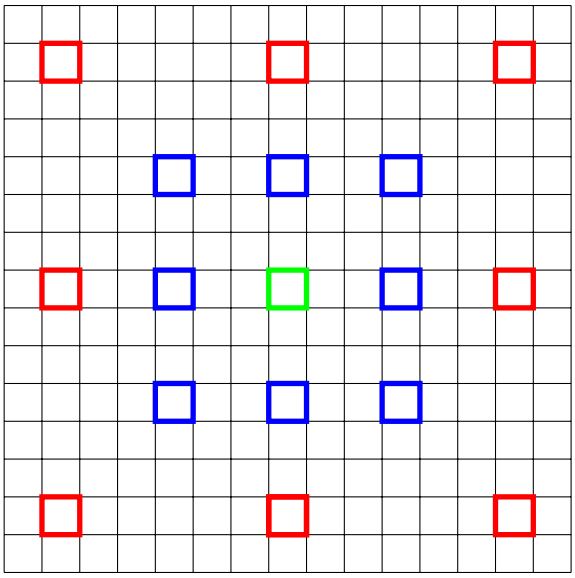

Figure 4. Clusters, and their neighboring points, are defined around the center node in green in this 2D example. The blue nodes represent a cluster of size 3, while the red nodes represnt a cluster of size 6.

4. Algorithm Optimization

In order for the algorithms to intelligently choose successor nodes to explore, they need access to accurate heuristics which effectively quantify how promising any given path is. Defining a strong heuristic is paramount to the efficiency of the search. If a heuristic overestimates the distance, then the planning algorithm will be 'tricked' into returning suboptimal paths. If a heuristic underestimates the actual distance, then the algorithm will spend excessive time searching for a better path that does not exist, significantly slowing the performance.

Since we are working within a uniformly gridded area, calculating the distance to the goal location from any arbitrary node is quite simple. The most obvious choice is the euclidean distance in three dimensions, seen in equation 1.

$$h = \sqrt{(\Delta x)^2 + (\Delta y)^2 + (\Delta z)^2} \tag{1}$$

Unfortunately, given the constraints imposed on movement within a grid, this distance isn't quite accurate. Consider the simple scenario presented in figure 5.

Here we consider 2 nodes within a uniformly spaced, unit grid. The green arrow represents the euclidean distance from the starting node S to the goal node G. The series of black arrows represent one optimal path from S to G. Because of the constraints on movement within a grid, it is impossible for the shortest realizable path to ever match the euclidean distance. As such, the euclidean distance actually underestimates the shortest path. As explained earlier, this

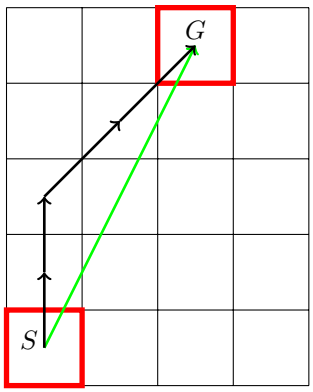

Figure 5. The euclidean distance underestimates the shortest possible path length because movement on a grid is constrained.

will cause the algorithm to spend excessive time searching for shorter paths which can not exist. Additionally, note that the path given in figure 5 is only one of many optimal paths which all share the same path length. On the surface, this is not a problem. Multiple paths can exist which have the same, minimum cost. As long as the algorithm returns any one of these paths, it has fulfilled its purpose. The problem, however, is in the processing time. At its heart, A* is a breadth-first search. This means that if there are a number of paths, N, that are all equidistant, then A* will explore all N of them in parallel. The 6 equidistant paths can be seen in figure 6. This occurs because the optimal path will have 2 vertical movements and 2 diagonal movements where the order of these movements does not matter.

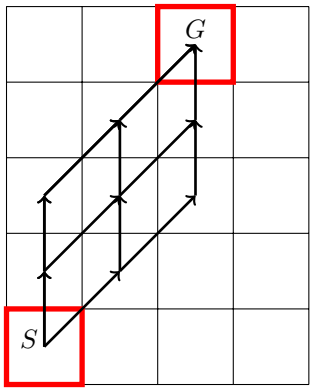

Figure 6. The 6 equidistant paths: (U, U, D, D), (U, D, U, D), (U, D, D, U), (D, U, U, D) (D, U, D, U), (D, D, U, U)

In this simple case, which requires only 4 moves, there are already 6 possible paths. As the goal location moves further from the start, the problem grows exponentially. Furthermore, these simple examples show only the 2-dimensional case, the transition to 3 dimensions exponentially increases search space as well. It is imperative, then,

that an ideal heuristic not only provide an accurate distance but also break ties between equidistant paths.

4.1. Accurate Heuristics

Ultimately, there is a disconnect between using the euclidean distance as the heuristic and the permissible movement on the grid. Because the euclidean distance is not informed by the allowable movements, it underestimates the effective distance in most cases. We can construct a more informed heuristic by starting from the permissible movements. To build up to an accurate 3D heuristic in a 26-connected neighborhood, it helps to begin with a 2D, 4-connected graph. In this case, only vertical and horizontal movement is permitted. We arbitrarily define the start location at (x_s, y_s) and the goal location at (x_g, y_g). The minimum travel distance between the start and goal location then becomes the Manhattan distance, also known as the L1 distance, seen in equation 2.

$$h = |x_g - x_s| + |y_g - y_s| \tag{2}$$

We can now extend this definition to the 2D, 8-connected grid where diagonal movements are permissible. In the 4-connected case, every move had a length of 1. Diagonal moves, however, have a length of $\sqrt{2}$. One diagonal move can replace a combined horizontal and vertical move and shortens the distance by $2 - \sqrt{2}$. As such, it is beneficial to take as many diagonal moves as possible. The maximum number of beneficial, diagonal moves is easily calculated as $minimum(\Delta x, \Delta y)$, where $\Delta x = |x_g - x_s|$ and $\Delta y = |y_g - y_s|$. The Manhattan distance, then, can be shortened by the product of the number of beneficial moves and the distance saved by making those moves, which results in the octile distance seen in equation 3.

$$h = \Delta x + \Delta y - (2 - \sqrt{2}) * min(\Delta x, \Delta y) \tag{3}$$

To extend this model to the 3D, 26-connected neighborhood is not so challenging. Of course, a new Δz variable must be added to account for the added third dimension, and where horizontal and vertical movements described the x and y motions, the z motions will be called perpendicular. The Manhattan distance follows in an equivalent manner as $h = \Delta x + \Delta y + \Delta z$ in a 6-connected neighborhood. The 2D diagonal moves from the octile distance open up diagonal moves constrained to the xy, yz, or xz planes, which facilitate an 18-connected neighborhood. There is one additional type of move, however, which is a 3D diagonal. This 3D diagonal is a change in x, y, and z. Similarl to the 2D diagonal, this move replaces a combination of the 3 moves - horizontal, vertical, and perpendicular. Taking a 3D diagonal saves a distance of $3 - \sqrt{3}$, and as before, taking the maximum number of beneficial, 3D diagonals will reduce the path length. The number of maximum, beneficial 3D diagonals becomes the $minimum(\Delta x, \Delta y, \Delta z)$. The number of 2D diagonals now needs to be recalculated. The 3D diagonals will remove the need for 2D diagonals across one axis, and as such, 2D diagonals will only be required within one plane, if at all. That axis is determined by the first minimum. To find the number of 2D diagonals, the second minimum is used. In other words, if Δx, Δy, and Δz were sorted in increasing order in a list called $deltas$, the number of 3D diagonals is the first value and the number of 2D diagonals is the second value. Given these definitions, the 3D octile distance can be defined as seen in equation 4.

$$h = \Delta x + \Delta y + \Delta z - (3 - \sqrt{3}) * deltas[0] - (2 - \sqrt{2}) * deltas[1] \tag{4}$$

While this may seem like an unnecessarily detailed discussion of heuristics, it is surprisingly impactful on the the speed of the algorithm. As a simple demonstration, consider an empty 3D grid with a goal defined such that $(\Delta x, \Delta y, \Delta z) = (70, 45, 20)$. To compare performance, A* is first run with the euclidean distance. Subsequently, the heuristic is switched and the path recomputed. The resulting running time and, most telling, the number of expanded nodes are shown in table 1.

Heuristic	Running Time (s)	Expanded Nodes
Euclidean	3096	43873
3D Octile	1.47	525

Table 1. Performance comparison between using either the Euclidean distance or the 3D Octile distance as the heuristic.

The scrutiny placed on defining accurate heuristics is clearly justified. The number of expanded nodes is about 2 orders of magnitude less in this example, and the computation time is more than 3 orders of magnitude less (this occurs because the computation time is not O(N) with respect to expanded nodes). The benefit provided by this more accurate heuristic is not static, but depends on the combination of $\Delta x, \Delta y, \Delta z$. This is most obvious when two of the deltas are 0. In that case, the euclidean distance returns the same value as the 3D octile and the change in heuristic provides no benefit. Of course, only a very small percentage of configurations exist where the euclidean distance is equivalent to the 3D octile, and this change in heuristic will universally improve the performance of the algorithm. Also of note, as the paths become longer (as they are in all of our test cases) the effect becomes more and more pronounced. This change in heuristic benefits not only the implementation of A*, but also HD*.

4.2. Tie-Breaking

In addition to an accurate heuristic, the search space can be reduced further by tackling the need for tie-breaking. This is done not in the heuristic itself, but inside the A*

algorithm as an adjustment to the F-score. Ties occur because every node that lies on any of the shortest possible paths has the same F-score. Again, this is not problematic because of the number of equally optimal paths, but only because it slows down the search when all optimal paths are explored in unison. If the F-score of all the tied nodes can be slightly perturbed, the algorithm can be encouraged to explore a subset of them at a time. If the F-score is altered by too large a margin, however, then it will no longer correlate accurately with path length and the algorithm will be at risk of returning sub-optimal paths. As long as the perturbations remain below the threshold of one move, then this can be prevented and optimal paths will be guaranteed.

Recall that the F-score is a combination of both the g value and the h value, where the g value represents the distance traveled to the current node and the h value is the heuristic measurement from the current node to the goal. One method of tie-breaking is to favor those nodes with larger g values - those which are further away from the start. By doing this, the algorithm is encouraged to follow optimal paths for as long as they remain optimal. In some sense, it allows the algorithm to function as a depth-first search approach with the condition that the path it is exploring remains one of the best possible choices. If that path intersects an obstacle, for example, it's F-score will increase by much more than these small perturbations and the algorithm will then backtrack and begin exploring other routes. This is exactly the functionality desired - if an obvious, easy solution exists, the algorithm should find it quickly and be done. If not, then the algorithm will not be negatively affected by these perturbations, as they are incapable of outweighing the cost of an indirect movement. To create a slight perturbation with these characteristics, the F-score is modified as seen in equation 5.

$$F = g + (1 + \alpha) * h \tag{5}$$

Before this change, there are a number of tied nodes with equal F-scores - but their g and h values are not necessarily equivalent. By slightly increasing the second term by a factor α, those nodes with larger g values (and thus lower h values) will be penalized less. As such, paths with larger g values will come out slightly ahead, prioritizing exploration of paths which are further from the starting location. Determining α is relatively simple. Movements on the unit grid cost at a minimum 1. The total change to the F-score must be less than this, and that would be facilitated by $\alpha = 1/pathlength$. Since the path length is not known at the start, it is acceptable to use the heuristic to estimate what the shortest path is and then use some multiple of that. For example, equation 6 will almost certainly work except for the most convoluted of mazes.

$$\alpha = \frac{1}{10 * h(start, goal)} \tag{6}$$

Even then, it would require enormous path lengths to throw off the optimality of the path by just one sub-optimal move within the path (i.e. it may take a combined vertical and horizontal movement instead of a diagonal). As such, it is safe to adjust the F-score as seen in equation 5 with α as defined in 6. To investigate the efficacy of this tie-breaking approach, we reconsider the previous example of an empty 3D grid with a goal defined such that $(\Delta x, \Delta y, \Delta z) = (70, 45, 20)$. The F-score is adjusted according to the above equations and run in the same manner, with the results presented in table 2.

Heuristic	Running Time (s)	Expanded Nodes
Euclidean	3096	43873
3D Octile	1.47	525
3D Octile + α	0.102	70

Table 2. Performance improvements from implementing the tie-breaking alpha value.

Again, performance is enhanced considerably by this subtle change. As can be seen in the table, the running time is down by an order of magnitude as are the number of expanded nodes. The combination of accurate heuristics and tie-breaking adjustments are paramount to the performance of these types of algorithms, and take running times that would be considered unworkable into the realm of relatively fast.

5. Visualization

Given the size of the environments, standard scientific plotting tools are insufficient. They are simply not capable of rendering such large datasets, and an alternative approach had to be found. After evaluating many options, we settled on the Vispy library. This is a python library which provides powerful wrappers and functions to interface with a GPU through OpenGL. This is, computationally, many orders of magnitude more efficient than using standard plotting tools. By leveraging the graphics card, with its own native functions, the challenge of displaying volumes of this scale reduces to solely an implementation problem. There were quite a few hurdles to clear, not least of which was setting up an environment with compatibility between all the interconnected libraries. After an exhaustive search, the installed environment for visualization is in python 3.4 with the dev0.50 Vispy release.

This combination of tools is used in a number of useful functions. The first, and most obvious choice, is simply to display the environment. Given that these are large data structures in three dimensions, it is impractical to try and

understand their properties without such visualization techniques. This process is quite simple. The environment was generated on a unit Cartesian grid, where nodes which contained obstacles were given values greater than zero. This allows the use of the display volume function, treating the 3D array as a volume. By setting the zero values to transparent, it is easy to render the environments as seen earlier in figures 2 and 3.

The next function follows the exact same premise but displays the path on top of the environment. This allows for quick sanity checks on whether the algorithm is working properly - whether it is effectively avoiding obstacles and taking paths which are conceivably near-optimal. The same volume rendering technique is used from the previous function and then the path is overlaid on top, as seen in figures 7 and 8.

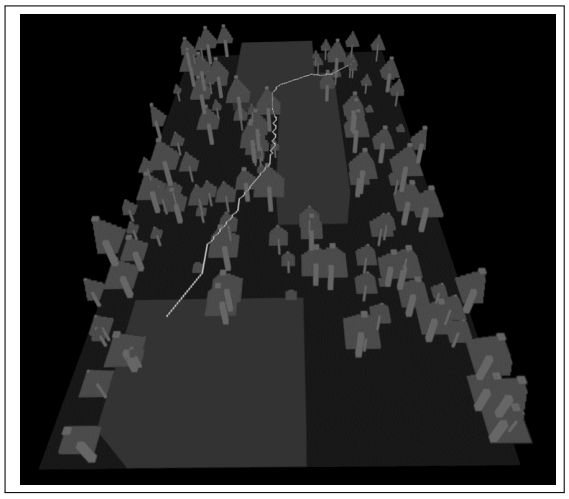

Figure 7. View of the path generated when testing environment 1, overlaid on the transparent map view.

The most interesting of the display functions is the execute path function. This function allows the user to understand not just the final results of a test, but each individual step within the process. This function displays, up to a given time step, the portion of the map that the UAV has explored, the path that the UAV has followed to that point, and the path it intends to execute from that point onward. The time step can be increased or decreased within the visualization, so the user can effectively watch the planner make decisions and explore the map. This provides further checks on the operation of the testbed but is also insightful when it comes to understanding the challenges that real-time planners face. A few snapshots are included in figures 9, 10, and 11 to demonstrate the visualization created by such a function. As can be seen in figure 9, the planner is attempting to go around what it perceives to be the end of the building. As it discovers more of the building, in figure 10, it has to replan the path as the previous path would have collided with

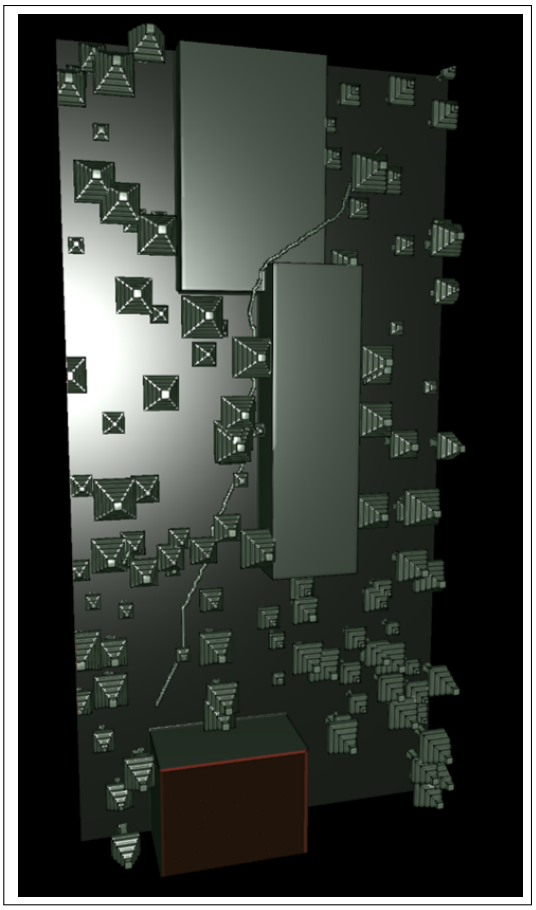

Figure 8. Another perspective of the path generated when testing environment 1, overlaid on the opaque map view to clearly see the building traversal.

the building. Again it plans on going around the building at what it now perceives as the edge. In figure 11, after discovering even more of the building, it adapts its approach as the prospect of flying around the building is less promising than flying over. At this point, it begins to head up and over the building, as seen earlier in figure 7 and 8.

Utilizing a powerful openGL package for the visualization enabled the ability to display these large 3D environments, as demonstrated, with the added ability to interactively rotate, pan, and zoom. This allows the user to fully explore the environment in its entirety, as well as the ability to analyze the temporal dependence.

6. Preliminary Results

With the testbed system complete, a few basic tests were done in order to compare baseline performance metrics for the two implemented algorithms. They were tested across three different generated environments, with varying levels of difficulty. It should be noted that, in all cases, both plan-

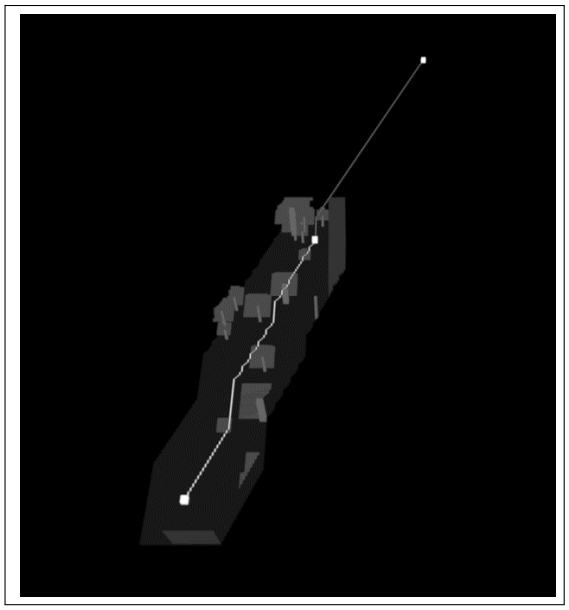

Figure 9. Mid-simulation result showing relative time 0. In view are the UAV's current position, path it took to get there, knowledge of the environment, and future path forward.

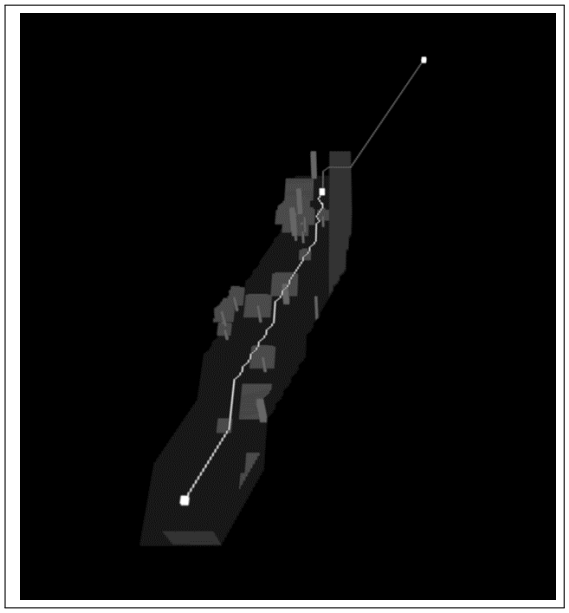

Figure 10. Mid-simulation result showing relative time 1.

ners found very similar paths - and path lengths deviated between the two by less than 5%. The performance statistics of interest were the mean path-planning time, the maximum path-planning time, and the number of nodes expanded over the course of the run. The raw times are useful metrics because they are most applicable to the problem at hand. Recall that the maximum speed the UAV could traverse the environment is limited by the path computation time. It is also

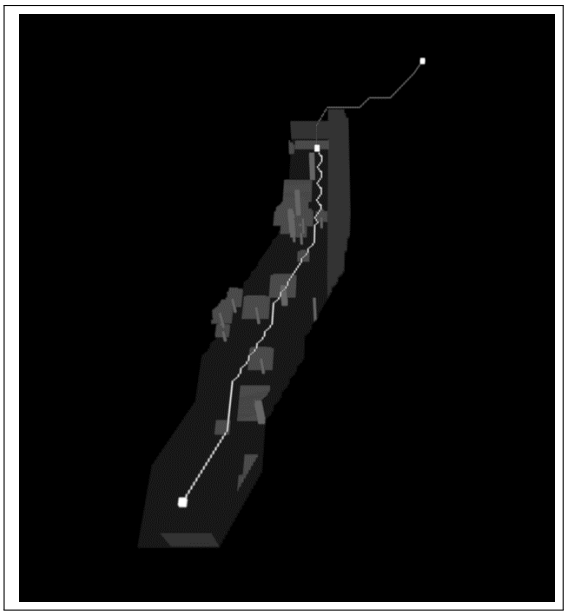

Figure 11. Mid-simulation result showing relative time 2.

useful to compare maximum computation time, as it represents the worst case scenario for a given trial. The number of nodes expanded is a more abstract metric, but helps to illuminate how much work each algorithm needs to perform in order to find the goal. It should be noted that the number of nodes expanded does not correlate linearly with the running time. The resulting mean and maximum running times for A* and HD* can be seen in table 3 and the number of expanded nodes in table 6.

	Mean Time (ms)		Max Time (ms)	
Environment	A*	HD*	A*	HD*
1	422	24	1618	104
2	560	58	5273	221
3	719	29	7825	95

Table 3. Comparison of the mean and max running time between A* and HD*.

	Expanded Nodes	
Environment	A*	HD*
1	65460	8951
2	83045	21117
3	87944	8326

Table 4. Comparison of the number of node expansions between A* and HD*.

Clearly, the HD* algorithm excels in this problem space when compared to A*. In all cases for the mean path finding time, HD* beats out A* by at least an order of magnitude.

In the maximum path finding time, HD* has an advantage that approaches two orders of magnitude in some cases. Of course, the explanation for this is clear when referring to the number of node expansions, where HD* holds a significant advantage.

These results are not particularly surprising. HD* was designed specifically for this type of problem, while A* is not particularly tailored for it. HD* reuses past calculations, which drives down the number of nodes that need to be recomputed. Additionally, HD* gains the benefit of hierarchical planners, which helps to further reduce the number of node expansions.

Note that these tests were run on a Samsung Notebook 9 Pro laptop with an I7 6700HQ quad core processor at 3.3GHz (though none of the code is set to run in parallel). It is important to note that this is a significantly faster system than what is available on-board the average UAV.

7. Future Work

Having created a testbed and done some preliminary testing, there are a number of questions now open for discussion and further exploration. Of course, the primary goal is to facilitate the development of better, faster algorithms. To begin this process, it would be wise to identify an entire complement of testing environments to evaluate performance across. Having a wide range of sizes, complexities, and difficulties will help identify the strengths and weaknesses of particular techniques. For example, HD* benefits from hierarchical planning, but it would be interesting to see if extremely cluttered environments reduce this benefit (as the planner is unable to abstract the environment as effectively). Implementing other purpose-built 3D, real-time planners would also be of interest. Hopefully, by testing different techniques against one another, the strengths of each algorithm can be revealed and built upon.

Currently, there are no guarantees that the path planning algorithms will produce paths that are flyable by a UAV. Future iterations would take into account the UAV's state vector to either remove neighboring nodes which are not attainable given the current state, or penalize nodes which would require drastic deceleration. This would require a change from the binary map (obstructed vs. unobstructed) to a cost-to-go representation.

8. Conclusion

Given the vast number of future applications that will require 3D, real-time path planning algorithms, there is a clear need for their continued development and exploration. With this need in mind, we created a testbed system capable of simulating these 3D, real-time problems in a manner that is fully extendable to allow for easy swapping of both the testing environments and the algorithms. It simulates the movement through, and exploration of, the environment while passing that data to the path planner. Relevant metrics and information are saved to allow for detailed analysis, and a powerful set of visualization tools is built in. We use this system to compare the performance of A* and HD* to demonstrate the functionality of the testbed and to showcase one 3D, real-time path planning algorithm which holds some promise for future applications. This system will facilitate further testing and exploration, and hopefully lead to the development of fast, efficient algorithms to enable robots of the future.

References

[1] D. Mellinger and V. Kumar, *Minimum Snap Trajectory Generation and Control for Quadrotors*, in Proc. of the IEEE Conf. on Robotics and Automation, 2011.

[2] M. Solomon and H. Xu, *Development of a Real-Time Hierarchical 3D Path Planning Algorithm for Unmanned Aerial Vehicles*, University of Maryland, 2016.

[3] A. Barry and R. Tedrake, *Pushbroom Stereo for High-Speed Navigation in Cluttered Environments*, in Proc. of the IEEE Conf. on Robotics and Automation, 2015.

[4] F. Yan, Y. Zhuang, and J. Xiao, *3D PRM based Real-time Path Planning for UAV in Complex Environment*, in Proc. of the IEEE Conf. on Robotics and Biomimetics, 2012.

[5] S. Aine and M. Likhachev, *Truncated Incremental Search: Faster Replanning by Exploiting Suboptimality*, in Proc. of the AAAI Conf. on Artificial Intelligence, 2016.

[6] Q. Wang, A. Zhang, and H. Sun, *MPC and SADE for UAV Real-Time Path Planning in 3D Environments*, in Proc. of the IEEE Conf. on Security, Pattern Analysis, and Cybernetics, 2014.

[7] J. Carsten, D Fergusen, and A. Stentz, *3D Field D*: Improved Path Planning and Replanning in Three Dimensions*, in Proc. of the IEEE Conf. on Intelligent Robotics and Systems, 2014.

[8] S. Koenig, M. Likhachev, and D. Furcy, *Lifelong Planning A**, Computer Science Department, USC, 2006.

[9] S. Koenig, M. Likhachev, *D* Lite*, College of Computing, Georgia Institute of Technology, 2005.

[10] A. Botea, M. Muller, J. Schaeffer, *Near Optimal Hierarchical Path-Finding*, Department of Computing Science, University of Alberta, 2004.

[11] Amit's A* Pages. `http://theory.stanford.edu/~amitp/GameProgramming/`