RRT-GPU and Minecraft

by

Christen Ford

A Thesis

submitted in partial fulfillment  
of the requirements for the degree of  
Bachelors of Science in Computer Science   
Baldwin Wallace University

2018

Contents

[Chapter 1: Introduction 4](#_Toc509932570)

[Chapter 2: Literature Review 6](#_Toc509932571)

[1. Literature Review 6](#_Toc509932572)

[2. Conclusions 8](#_Toc509932573)

[Chapter 3: Research Question and Methodology 9](#_Toc509932574)

[1. Research Question 9](#_Toc509932575)

[2. Methodology 9](#_Toc509932576)

[3. Plan 10](#_Toc509932577)

[4. The A\* Search Algorithm 10](#_Toc509932578)

[5. The Rapidly Exploring Random Tree Algorithm 13](#_Toc509932579)

[6. Publication Possibilities 13](#_Toc509932580)

[Chapter 4: Findings 15](#_Toc509932581)

[1. Testing Environment 15](#_Toc509932582)

[2. A\* Statistical Summary 15](#_Toc509932583)

[3. RRT Statistical Summary 16](#_Toc509932584)

[4. RRT-GPU Statistical Summary 17](#_Toc509932585)

[5. F-Test Statistical Summary 17](#_Toc509932586)

[6. Z-Test Statistical Summary 17](#_Toc509932587)

[7. Conclusion 17](#_Toc509932588)

[Chapter 5: Conclusion 18](#_Toc509932589)

[1. Conclusion 18](#_Toc509932590)

[2. Future Work 19](#_Toc509932591)

[3. What I Learned from This Experience 20](#_Toc509932592)

[Appendix 21](#_Toc509932593)

[1. A\* Results 21](#_Toc509932594)

[2. RRT Results 22](#_Toc509932595)

[3. RRT-GPU Results 23](#_Toc509932596)

[4. F-Test for Equality of Variance: Two Populations 25](#_Toc509932597)

[5. Z-Test: Difference of Population Means 27](#_Toc509932598)

[References 29](#_Toc509932599)

## Chapter 1: Introduction

Path planning is a branch of artificial intelligence that is concerned determining routes for an artificial agent in both simulated and real environments. There have been many goals of path planning as well as many application domains. The most well-known involve a subset called shortest path algorithms. This type of path planning, as the name implies, is all about determining the shortest path from a source to a destination. Optimally, the best solution to this problem is to just follow a straight vector from the source to the destination. However, in practice, this is never achievable due to physical constraints such as obstacles and terrain or limits placed on the agent itself.

Another factor to consider when path planning is the number of dimensions to plan in. The obvious case and most easily approached is path planning in two dimensions. That is, path planning performed in a system that utilizes an (x, z) coordinate system. Many problems presented in two dimensions are easily solved by the various algorithms devised to do so, notably Dijkstra class algorithms. However, path planning becomes immeasurably more complex in terms of computation when the same problems are elevated to three dimensions. Unfortunately, this means that without advanced hardware, it becomes impractical to compute paths of any kind in three dimensions. This does not mean it cannot be achieved. Indeed, in constrained, regulated environments where inputs are limited, three dimensional paths can be determined. However, to have applicability to the real world, we must be able to at least model a spatially restricted sample of the world for pathing.

The final restriction on pathing in three dimensions is created by the enormous size of the sample space. With two dimensions of space, the sample space can be analyzed and a path can be computed in real time, regardless of complexity. If the sample space can be solved, a path can be determined. There currently exists only a small handful of path planning algorithms for computing paths in real time that are suitable for use in three dimensions. The most recent of these algorithms expands on the concept introduced by rapidly exploring random trees. As explained by Naderi et al. it extends the algorithm thorough constant resampling and rewiring to enable real time path planning (2015). This algorithm is known as rapidly exploring random trees or RT-RRT\*.

Given the recent advancements in power consumption for both central and graphical processing units. It is now possible to create a pathing algorithm that leverages advanced hardware to be able to create paths in three dimensions on the fly. While mobile computing has been a standby for decades, it has only recently reached a point where it matches the performance of traditional computing. This is further augmented by advancements in artificial intelligence as seen in platforms such as autonomous vehicles and the internet of things as well as robotics as showcased by Boston Dynamics.

Theoretically the improvements gained through distribution can be many. Most notable are improvements in the speed of calculations and the memory required to do so. Important is the fact that many modern graphical units involve computations across thousands of individual cores. Unlike a central processing unit which is better suited for throughput as opposed to speed, a graphics processing unit is capable by design of calculating at rates unachievable by a traditional processor. In addition, while central processing units contain multiple layers of localize memory cache, graphics processing units contain onboard with space thousands of times greater than the cache found in a traditional processor. Therefore, computation via the graphics pipeline weighs much more heavily on the development of artificial intelligence then does central processing. Cost is another factor to consider in this approach, however the cos of modern graphics hardware is now roughly equivalent to top tier computational units and is widely accessible to the public.

What this all leads to is the need for a new class of algorithms for a new world. Future algorithms will always build on those introduced in the past. This thesis serves to lay the framework for one such algorithm. This algorithm shall extend real-time rapidly exploring random trees to plan paths in real time, three-dimensional environments. It shall utilize advanced graphical hardware and processing techniques introduced in previous works. Finally, this algorithm shall be easily portable. Unlike several algorithms that utilize global data structures to compute paths, this algorithm shall be parallelized. This will allow us to compute paths across graphical cores, and possibly in the future across graphical units.

The goal of this thesis is to create an algorithm that under a set of specified criterions (specified in chapter three), can compute three dimensional paths. However, even if the algorithm fails to meet these criterion, it will still classify as a success. Much of the work done in three-dimensional path planning is experimental. It is not as thoroughly researched as two-dimensional path planning. This leads to many failures in this field that further result in advancements at later times. As stated earlier, the need for three-dimensional path planning is rapidly increasing. A generalized approach to the planning problem can be utilized to solve problems in realms outside of computer science.

## Chapter 2: Literature Review

### Literature Review

Creating three-dimensional paths directly in a three-dimensional plane is both temporally and spatially expensive. The direct approach to doing so is best left to high-powered machines as more widespread, cheaper computers cannot easily handle such a task. To solve this problem, we reintroduce the concept of spatial subdivision. This process takes a three-dimensional plane and splits it into a series of connected, two-dimensional planes using both voronoi diagrams and spatial subdivision. However, there is an additional drawback to this method. That is, computing voronoi diagrams is known to be exponential in run-time at the worst case and polynomial in run-time at the best case. To solve this, we leverage the fact that many household computers now contain discrete graphics processing units. They are built specifically to handle mathematically intensive work, that the central processing unit would otherwise struggle through. That said, Hoff et al. also present ways to compute voronoi diagrams utilizing GPU shaders. Camporesi and Kallman take the findings presented by Hoff et al. and apply them to computing shortest path maps. Finally, there is the issue of pathfinding in these newly created two-dimensional domains. The works presented by Ramires and Leonel, Leonel et al., Mitchell and Sharir, Naderi et al., and Burch and Weiskopf attempt to solve this.

As presented by Hoff et al., a voronoi diagram is simply a two-dimensional representation of a space that is split into voronoi regions utilizing carefully selected voronoi sites. The resulting boundaries, known in the literature as voronoi boundaries are, are then used by pathfinding agents. A common application of this technique is outlined in Hoff et al. to solve the piano movers’ problem, “The underlying idea is to treat the obstacles as sites. The voronoi boundaries then provide paths of maximal clearance between the obstacles” [1999]. As mentioned earlier, the issue with this technique is time. All algorithms that compute voronoi diagrams do so iteratively, the longer the time they can run for, the more accurate the representation of the space the resulting voronoi diagram will represent. A well-known algorithm for computing voronoi diagrams is known as Llloyds’ algorithm or more commonly, K-Means clustering. Hoff et al. utilize a parallelized version of this algorithm to compute voronoi diagrams using GPU shaders.

The techniques presented in Hoff et al. are then referenced in both Camporesi and Kallman [2014] and Mitchell and Sharir [2004] to further refine solutions to the three-dimensional pathing problem. Camporesi and Kallman build on the work done by Hoff et al. and present methods for computing shortest paths using GPU shaders. As stated in Camporesi and Kallman “Our method first relies on standard CPU algorithms for computing the shortest path tree of the obstacle set, and then applies the proposed shaders to encode the SPM in the frame buffer with arbitrary resolution” [2014]. This method has three stages. First, the environment space is preprocessed into discrete two-dimensional regions. Second, visibility graphs and shortest path trees are computed for each region. Finally, the shortest path map for each region is computed and the resulting paths for each region are adjoined together to create the overall shortest path map [2014]. Mitchell and Sharir apply voronoi diagrams to compute paths amongst stacked sets of axis-aligned polygonal shapes. They approach the problem by using spatial subdivision to represent the polygons as terrain. They then compute the shortest path using a topographical, top-down approach. This technique forces the agent to stick to flatter terrain, but could have the adverse effect of causing the agent to “sweep the terrain upwards” [2014] wherein the agent will generate a non-shortest path. As an aside to their work with terrain, Mitchell and Sharir also provide methods for computing shortest path distances over walls. They represent the wall as a series of interconnected lines in 3-space and then compute the shortest polygonal paths between them. The resulting sub paths create a path that appears to bend around the wall and is proven to be y-monotone.

Now that we have discussed methods for representing three-dimensional maps in two-dimensional space, we will discuss techniques for creating path from that space. Ramires and Leonel propose utilizing collision detection introduced in [2006] and later refined by Leonel et al. in [2008] to navigate three-dimensional space. They utilize a height based approach that automatically extracts needed information from the three-dimensional world and creates a minimum two-dimensional representation. “This is achieved by slicing the world with horizontal planes. For each slice, the height at which the slice was taken, as well as the height map obtained at that slice is kept” [2006]. This method works well in constrained, highly populated environments. When combined with the proper pathfinding algorithm, it can find paths quickly and efficiently. In their paper from 2008 Leonel et al. build on the information presented in their earlier paper and apply it to dynamic environments. The initial step in their process involves slicing to create spatial subdivision maps. After this step, they then compose the resulting planes together at connection points to create a single hierarchical navigation graph. This graph can then be utilized by A\*, in static environments, or in our case RT-RRT\* for dynamic environments. Even in a static environment RT-RRT\* is guaranteed to outperform A\* temporally, but will it will consume more space than A\*.

To Give some background into what a rapidly exploring random tree is, we present and review the findings in the work performed by Burch and Weiskopf They present algorithms both for computing rapidly exploring random trees and algorithms for visualizing them. Per Burch and Weiskopf an RRT as it is known in the literature is simply a tree that “… is computed incrementally by adding a new sample to the tree randomly, computing the least distant already existing sample in the tree by a distance function, and finally connecting both samples by as straight line that produces a new branch of the tree” [2013]. This process is repeated until the final path is computed. And since RRT is a probabilistic search method, it is faster than all the classical Dijkstra search methods and is also capable of parallelization. The visualization algorithms presented by Burch and Weiskopf allow for rendering an RRT on the screen as a graphical heat map, with earlier parts of the tree appearing on the screen more intensely than recently explored regions. The algorithm produces jagged edges around obstacles, but we are not concerned with the smoothness of the path, only its optimality.

Naderi et al. propose a modified version of Rapidly Exploring Random Trees call RT-RRT\* that can explore dynamic environments with adversaries. It works via incremental resampling and is explained best by Naderi et al. “…At each iteration, we expand and rewire the tree for a limited user-defined time. Then we plan a path from the current tree root for a limited used-defined amount of steps further” [2015]. While RT-RRT\* works great for dynamic environments, it has some drawbacks. First, is the spatial complexity of the algorithm. It stores the entire tree in memory and keeps it there until the path is found. For smaller maps, this is not an issue, but as the region RT-RRT\* is set to explore grows, the spatial requirements of this algorithm will grow with it, a drawback that is offset by the VRAM available on the GPU. Second, this method is optimized for bounded environments. An analysis of the map must be conducted beforehand to optimize RT-RRT\*. This is because it uses an ellipsoid method for resampling and rewiring. If the distances are too large, then this method will suffer from in time complexity.

### Conclusions

As discussed in Ramires and Leonel [2006] collision detection can be made to be efficient simply by utilizing a divide and conquer approach of slicing and pathing. Camporesi and Kallman [2014] provided a method of quickly generating shortest path maps using GPU shaders through a three-step process. They also provided some potential drawbacks to consider, and how to avoid them. Mitchell and Sharir [2004] provide some techniques for generating shortest path map on polygonal terrain by simply pathing above it. They also introduce an approach to pathing over obstacles, useful in certain game genres.

Hoff et al [1999] show how voronoi diagrams can quickly be generated for both 2D and 3D worlds using graphics hardware. They do so by splitting the world at voronoi sites (represented as obstacles) and using the boundaries created between them to effectively path. Naderi et al. [2015] debut an advanced implementation of Rapidly Exploring Random Trees that allows paths to be computed in real time, dynamic environments. The algorithm they provide will constitute the pathing algorithm used in this thesis.

Leonel et al. [2008] showcased a method that demonstrated ow a 3D environment can be sliced into 2D chunks, that can then be connected through a single hierarchical navigation graph. This would enable easy traversal of the world by RT-RRT\*. Burch and Weiskopf [2013] show the beauty in rapidly exploring random trees. More importantly, they present novel techniques that will be utilized to help visualize the paths produced by RT-RRT\* in the 3D world.

## Chapter 3: Research Question and Methodology

### Research Question

Path planning in a multi-level, three-dimensional, dynamic environment is not easily solved. This has led to many different solutions to this problem that were created for specific application domains within this problem. Many solutions are not cross compatible. A solution for a specific application of this problem may not work for another, and in fact, this is often the case. Is it possible to adapt real-time rapidly exploring trees to better suit the needs of dynamic path planning in three dimensions? The algorithm should be able to compute paths on the fly Given this knowledge, we propose a method for solving the multi-level, three-dimensional, dynamic path-planning problem that will be generalized to apply to various application domains.

### Methodology

This research will utilize the Project Malmo framework for the test environment for this research. Initially established by Microsoft Research on June 1st, 2015. Project Malmo is a framework that utilizes the Minecraft game environment for creating artificial agents. It was chosen as Minecraft is a discretized, constrained environment, where in-game agents have predictable behaviors. In addition, Minecraft contains both a world generator and the ability for players to create their own worlds. As the focus of this research will be on developing the path-planning agent and not the world it will navigate, it makes sense to utilize a prebuilt framework.

The agent must be able to navigate a set of small to medium sized predefined worlds with the goal of navigating to the goal destination. Before it may be utilized on randomly generated worlds, the agent must meet the following criteria for all testing worlds:

1. The agent shall be able to navigate diverse, multi-leveled terrain, some of which is hazardous to the agent such as lava.
2. The agent shall be able to navigate around opposing agents, avoiding them when necessary.
3. The agent shall not exceed greater than 25% deviance from the optimal path to the goal. As each world contains a predefined layout and goal, the only contributing factor to deviance from the optimal path shall be opposing agents and hazardous terrain.

To further strengthen statistical analysis of the path planning algorithm, it shall be run across each map a total of 1,000 times. This should enable us to filter out any bias introduced into the population via a small sample size. Because the algorithm will be run on the provided NVIDIA GPU cores, time constraints on the actual testing process are not a concern.

As Minecraft is a complex, three-dimensional multi-level environment, preprocessing must be performed before the agent may path the world. This thesis shall utilize a recently released real time variant of rapidly exploring random trees (RT-RRT\*). RT-RRT\* creates an elliptical sample space to create the tree from. To do this in the Minecraft environment, we will leverage the distinct unit boundaries created by the (x, y, z) world system. Where each unit is called a block, and is exactly one meter in length, width, and height and is exactly sixteen pixels.

A connected hierarchical tree can be constructed at each z-level of the world, with transition points being single blocks that connect one z-level to the next. After this stage, three-dimensional RT-RRT\* will be able to function as it normally does in two-dimensions.

The machine utilized for this research is an MSI GT62VR 7RE with the following hardware:

* Windows 10 Professional
* Intel Core i7 7700HQ @ 2.8GHz
* 16 GB of DDR4-2400 Memory
* 480GB Intel 535 Solid State Drive
* NVIDIA GeForce GTX 1070 8GB

In addition, Project Malmo allows for writing agents in Java, C#, Python, and C++. However, for optimization and compatibility with all of Project Malmos’ features and NVIDIAs’ Compute Unified Device Architecture (CUDA) application programming interface, this thesis shall utilize the Visual C++ 2015 programming language.

### Plan

Preceding the actual implementation of the thesis, a few preliminary steps must be accomplished first, these include:

1. Install and configure Project Malmo on the development machine.
2. Create a data structure to represent the ellipsoidal tree of the search space.
3. Upscale RT-RRT\* to three dimensions.
4. Parallelize RT-RRT\* to work across multiple GPU cores.

After these steps are accomplished, the pathing algorithm will be hooked up to Project Malmo so that it may retrieve information from the game environment and send pathing information back to the agent playing the game. At this point, data collection can begin. In this stage, the pathing algorithm shall be run per the parameters outlined in the “Methodology” section. Once the algorithm is successful per the success criteria, it shall be allowed to run on several randomly generated worlds to gauge effectiveness (per the same pathing criteria for the predefined worlds) in times of true uncertainty. Results from testing on the predefined worlds shall be used as a baseline shall be used to gauge how well the agent performs on the random worlds.

This thesis will produce a three-dimensional version of real-time rapidly exploring random trees. It shall utilize Project Malmo as a testbed. To verify the algorithm, it will utilize several predefined, user-created worlds to ensure compatibility with the Minecraft environment. After this point, the algorithm shall be utilized on several randomly generated worlds. To ensure repeatability, the random world generated will be seeded with known Minecraft seeds for popular Minecraft worlds.

### The A\* Search Algorithm

To gauge the performance of RRT we required a comparable algorithm to compare against. It was the decision of the team to utilize the A\* path planning algorithm for this purpose. A\* is a vastly popular path planning algorithm. It finds use in video game environments, movies, robotics, and many other forms of entertainment and electronics. A\* is an offline path planning algorithm. In this regard, A\* is run prior to the agent making any decisions. Traditionally, A\* would find a complete path from the agents starting state to the goal state. In real-time environments, A\* is modified to plan for a limited amount of time to find a state that looks the best to reaching the goal state. For this discussion we will discuss the offline, non-real-time variant of A\*.

The A\* search algorithm is a modified version of Dijkstra’s search algorithm, known in the literature as Dijkstra’s algorithm. Both algorithms work on the premise of creating a path that adheres to some constraint. The most common use case for these algorithms involves finding shortest-paths. This technique is employed in this thesis to find paths using A\* search. Where A\* and Dijkstra’s algorithm differ is in the way they evaluate neighbor nodes. A\* and Dijkstra’s algorithm both operate on the premise of a heuristic or best guess metric. Dijkstra’s algorithm utilizes a null heuristic, meaning the best guess distance to the goal node is always zero. The performance of A\* is highly dependent on the heuristic that is used. Finally, much of what has been done with path planning involves two-dimensional path planning. Many path planning algorithms can be generalized to n-dimensional space.

The classical implementation of A\* utilizes a priority queue (referred to in the literature as the open list or frontier) data structure to choose successor nodes. Upon initialization of the algorithm, the starting node is placed on the queue with priority zero. The algorithm then repeats the following procedure until either all nodes have been evaluated, or the goal is found. A\* utilizes three primary functions to generate paths. The first *f(n)* is determines the overall value of the node currently being evaluated.

For A\* to produce correct paths the heuristic used must be consistent and admissible. A heuristic in A\* search is said to be consistent if for every neighbor node of each vertex in the path the estimated distance remaining to the goal plus the cost of reaching the neighbor is less than or equal to the actual distance remaining to the goal. With the additional constraint that the goal node should always have a heuristic of zero. Mathematically, **consistency** can be represented as:

Where…

* *h* is the heuristic distance function
* *c* is the cost function that determines the cost of moving from one node to the next
* *n* is the node currently being evaluated
* *a* is the action taken at node *n*
* *n’* is the successor node generated from having taken action *a* at node *n*

For the condition of admissibility to hold true, the heuristic value must never overestimate the cost to reach the goal node from any node in the graph. The result of these two constraints will cause A\* to never evaluate unnecessary nodes, thereby producing the optimal path. Mathematically, **admissibility** can be represented as:

Where…

* h is the heuristic distance function
* c is the cost function
* n is any node in the search space
* g is the goal node

The heuristic and cost functions vary per use case. To successfully find paths, a reasonably appropriate heuristic function must be employed. The cost function can be implemented in such a way to encourage certain paths, while discouraging others, as was done in this thesis.

The heuristic and costs functions employed in this thesis appear as follows:

Where…

* n is a node in the search space
* n’ is any descendant of n
* g is the goal node

There are several distinctions in with these functions that were chosen specifically for this thesis. First, we weight the y-dimension in the heuristic function. Minecraft utilizes a three-dimensional coordinate system for placement of objects and entities. The x and z axes determine horizontal position, while the y axis determines vertical position. This led to a natural adaptation of the heuristic function where we bias the y-axis to ensure the agent properly traverses between floors. However, there are some issues with this approach.

Our 3D A\* implementation suffers from the locality problem. Wherein, the agent will become locally trapped on floors due to the way neighbor evaluation is handled. We have corrected for this situation with careful design of the maps utilized in the thesis, but because of this problem, the implementation is not fully portable to other maps. Another potential solution for this problem would involve increasing A\*s’ scanning range (the area around the agent where neighbors are evaluated). In the current implementation, the agents’ scanning range is 3x3x3 wherein the agent evaluates only adjacent cells on the floor below, the current floor, and the floor above it for a total of twenty-six cells (the current cell is never put up for evaluation). Increasing the scanning range would dramatically increase the number of evaluated cells thereby increasing evaluation time.

The other problem that our A\* implementation suffers from is tie-breaking. We do not break ties in this implementation. In cases where there are multiple shortest paths (as is the case in the last map), the agent will consistently choose the first shortest path it comes across and will never utilize any other. A possible solution to this problem would involve a multi-step process.

First, we evaluate all neighbor nodes for their f-score. After this we are presented with a few options. We could just randomly choose from the best nodes. However, this would require us to iterate through all the successors twice. On the first iteration we calculate all f-scores while simultaneously finding the max f-score. On the second iteration, we mark all successor nodes that match the maximum f-score we found during the previous iteration. We would then randomly pick a successor from this sample.

The second solution involves modifying the h-score to act as the tie-breaker. This is as the h-score times the sum of one plus a weighting factor *p*. We then proceed to calculate the f-scores. Across the literature, this is the more common way of breaking ties with A\*. The issue with this approach is that it risks creating an inadmissible. To reduce this risk, the literature recommends:

### The Rapidly Exploring Random Tree Algorithm

A rapidly exploring random tree (RRT) is a randomized path planning algorithm. The algorithm is initialized with a starting point, a maximum distance between nodes, *epsilon*, and a maximum number of iterations. At each iteration, a point, *rand*, is randomly sampled from the search space. Then for each node, *p*, already in the tree, a distance function is used to find the nearest node, *nn*, in the tree to the randomly sampled point. A line-to function is used to determine whether to attach *nn* to *rand* or to generate a new point along the line created from *rand* to *nn*. This functionality is based on whether the distance between *rand* and *nn* is less than *epsilon*. If so, *rand* is used, otherwise the newly the algorithm generates a new point from *nn* to *rand*.

Our implementation differs from the standard implementation of RRT by utilizing NVidias’ CUDA library to accelerate the distance function using a discrete graphics card (GPU). In our implementation, the distance between *rand* and all neighbors contained in the tree is calculated simultaneously on the GPU. We utilize the Numpy and PyCuda libraries to make working with CUDA accessible within Python. We utilize the PyCuda ElementWise Kernel which allows for the simultaneous application of some predefined function to a given data set. The result of this kernel is a two-dimensional array of distances from *rand* to all the points in the tree. We then flatten the array to a single dimension and treat it as a parallel array of the tree. We then find the min index in the array and utilize the corresponding node in the array containing the tree as *nn*. We call this implementation RRT-GPU.

The Python implementation of this agent has some flaws that require discussion. First, the overhead incurred by utilizing the PyCuda library and Numpy dominate any potential gains from accelerating RRT over smaller maps. Second, the CUDA requires precisely aligned data sets to work with that adhere to strict standards based on the users GPU specifications. This affects the thesis in two ways: First, as it stands, the direct implementation of this thesis is not 100% portable to other hardware setups without some additional configuration; Second, due to the requirements of CUDA, GPU acceleration is only utilized every eight iterations of the algorithm. If this was not followed, the GPU would have to deal with jagged arrays that would require a complex striding algorithm to properly apply the distance function. This translates to increased run times for RRT-GPU.

This also impedes the algorithms ability to maintain and reconstruct paths. RRT-GPU keeps track of two arrays, an array containing the tree, and an array containing the distances as calculated by the ElementWise Kernel. PyCuda allows us to send and receive C structs to and from the GPU. However, this requires a custom Python class that contains a specialized wrapper. We did not opt for this approach and instead implemented a function in our Node class that converts the position contained within the node to a float3 vector object PyCuda can work with.

### Publication Possibilities

This thesis shall be publishable in various journals and conferences that deal with artificial intelligence. Several publication possibilities include:

1. Journal of Artificial Intelligence Research
2. The Annual Symposium on Computational Geometry
3. The International Conference on Motion in Games
4. Artificial Intelligence: An International Journal
5. AI & Society: Journal of Knowledge, Culture, and Communication
6. Applied Intelligence: The International Journal of Artificial Intelligence, Neural Networks, and Complex Problem-Solving Technologies

## Chapter 4: Findings

### Testing Environment

The purpose of this section is to reintroduce the testing environment utilized in the thesis as well as the final list of software packages. Data presented here was collected on an MSI GT62VR 7RE laptop with the following specifications:

* Windows 10 Professional Version 1709
* Intel Core i7 7700HQ @ 2.80 GHz
* 16GB DDR4 2400 MHz Ram
* Samsung MZNLN256HMHQ 256 GB SSD Primary Drive
* Intel 535 480 GB SSD Data Drive
* Nvidia GeForce GTX 1070 8GB @ 1650 MHz

In addition, this thesis utilizes the following software packages:

* Project Malmo 0.31.0
* Python 3.6.4
* PyCuda 2017.1.1
* NumPy 1.14.0
* SciPy 1.0.0
* Pandas 0.22.0
* Seaborn 0.8.1

Finally, we have incorporated a hand built three-dimensional world and collision detection system, meant to simulate the Minecraft environment in a meaningful way. This was done as Project Malmo has no way to extract complete world states from the agent. It was also done to allow us to plan offline, as opposed to having the agent plan in real-time as the Minecraft simulation was running. We previously attempted this before building the world system and it caused minute but highly impactive timing errors to occur between the Minecraft server and client as well as the Project Malmo agent.

We have conducted a thorough analysis of all four maps, the results of which are presented in the following sections. We first start by summarizing the statistics we have gathered across all of the algorithms. We then follow up with a summary of an f-test we conducted to determine if there was a significant difference in the variances of the RRT and RRT-GPU populations. We then proceed to an analysis of the RRT and RRT-GPU populations using a z-test to determine if there was a significant difference between the means of the two populations. Finally, we present a conclusion.

Note that all the data discussed in the following sections is present in full detail in the appendix.

### A\* Statistical Summary

Recall that we do not break ties with A\*, we merely select the first best node available for expansion at each iteration of the algorithm. This presented us with data that is identical across path length, number of heading changes, and total path degrees. However, we did observe a variance across run times. This is likely due more to fluctuations in speed of the processor during data gathering than it is the algorithm itself. In this regard, our A\* implementation produced the optimal path for each map on each iteration. This means that aside for the run time, all the other metrics pertaining to A\* can be used as a valid benchmark for RRT and RRT-GPU.

For map one we observed a mean run time of 0.1213 seconds. Given that A\* must evaluate all potential neighbors at each iteration, it incurs a run-time penalty in this regard. Map one saw a mean path length of 99.9706 units. We also observed a mean number of heading changes at 48. Finally, we saw a mean total path degree of 96.9516. Aside for the run-time which had a standard deviation of 0.0040, the standard deviation of the path length, number of heading changes, and total path degrees was either zero or almost zero. This indicates no change in the paths generated across map one by A\*. This proves A\* consistently found the optimal path.

With map two we observed a mean run-time of 0.1328 seconds, likely due to the slightly increased complexity over map one. We observed a slightly larger mean path length of 100.8406 units. There was also a slight increase in the mean number of heading changes and path degrees, due to the increased path complexity over map one. We also saw the same general spread of the data for run‑time with a standard deviation of 0.0056, the other metrics had the same standard deviations as map one.

Map three boasted a mean run-time of 0.1575 seconds, with the same level of complexity as map two, we expected similar results between the two. All other metrics for this map are identical to the metrics observed for map two. We observed a slightly smaller standard deviation for the run-times of the third map over the second, indicating a smaller spread, however, the difference is not significant.

Map four saw a significant boost in run-time over all prior maps with a mean run-time of 1.4893 seconds. This map is far more complex than the previous three maps as it enforces the agent to both ascend and descend to reach the goal. The path length presented a mean of 65.0957 units. While the mean number of heading changes was 37 and the mean total degrees was 58.8086. The path length exhibits a standard deviation in the range of the other maps, as do the other metrics.

### RRT Statistical Summary

We chose to stick with the standard implementation of RRT by Steven M. Lavalle. We utilize a python list to store our tree and iterate through it to find the nearest neighbor for tree additions. This method incurred a slight penalty in run-time over utilizing a C-based array, however the impact was negligible. We do however substitute all Python math library calls for Numpy’s math library. Numpy’a math library is implemented in C and is highly optimized for efficiency and accuracy. Given that we make multiple calls to it on each iteration, we felt this was a significant improvement over the default implementation.

We observed a mean run-time of 0.016885 seconds on map one utilizing the RRT algorithm. This is 786% decrease in mean run-time over A\*. This combined with a standard deviation of only 0.014143 seconds indicates that RRT clearly outperforms A\* on this map. RRT also carried a mean path length of 94.107885 units; 7.758 mean heading changes; and 91.150887 mean path degrees. Given that RRT is a randomized path planning algorithm, we expected a significant difference between the standard deviations of RRT and A\*, which is what was observed.

For map two, RRT carried a mean run-time of 0.031848 seconds. Accounting for a 494% decrease in run-time over A\* on the same map. RRT carried a significantly higher run-time standard deviation on this map then it did on map one at 0.051747 seconds. RRT had a mean path length of 96.884981 units, like map one; 9.101 mean heading changes; and 92.151255 mean path degrees. For these metrics, both the mean and standard deviation are near analogous to map one.

Across map three, RRT produced a mean run-time of 0.048252 seconds, a 326% decrease in run‑time over A\*. This is longer than both map one and map two. However, it is important to remember that map two and map three share the same design. The only difference is that the position of the goal is inverted. Likewise, RRT saw a run-time standard deviation of 0.109778 seconds. This is double what was observed on map two and indicates that RRT may struggle more when facing ascension problems than descension problems. Path length displayed a mean of 97.434895 and a standard deviation of 17.274995. RRT displayed a mean number of heading changes of 7.208 and a standard deviation of 2.986914. Finally, RRT gave a mean 95.237213 and a standard deviation of 16.687743.

Finally, on map four we observed a mean run-time of 0.138354, a 1,076% decrease in run-time over A\* on the same map. The run-time on map four carried a standard deviation of 0.286411 seconds. The other metrics were significantly different from those observed on the other maps. First, the mean path length was 54.031288 units with a standard deviation of 15.617457. Second, the mean number of heading changes was 7.431 with a standard deviation of 2.960406. Finally, the mean path degrees were 52.206334 with a standard deviation of 15.222530. The agent starts closer to the goal in this map than in any other map.

### RRT-GPU Statistical Summary

### F-Test Statistical Summary

### Z-Test Statistical Summary

### Conclusion

## Chapter 5: Conclusion

### Conclusion

This thesis showcased three-dimensional RRT and a GPU accelerated variant of RRT. We sought to determine whether a GPU accelerated version of RRT could outperform its unaccelerated counterpart. To realize this, we utilized the Project Malmo framework by Microsoft Research as a basis to work from. We then built a world and collision detection system for the agents to work with. This was necessary as the level of information the agents needed to function was not easily retrievable from the framework. After this, we created four maps for our agents to path across. Using A\* as a benchmarking algorithm, we then proceeded to gather data on RRT and RRT-GPU regarding run time, path length, number of heading changes, and total degrees in the path. This was repeated 1,000 times to generate an appropriate population to for statistical testing. The results of our testing show that, in our specific scenarios, accelerating RRT does not yield significant improvements upon run time. This was antithetical to what we sought to achieve. Nonetheless it provided valuable insight into the nature of RRT as a viable path planning algorithm.

In a practical sense, RRT must know of its immediate surroundings. In this thesis, our maps were small enough to allow us to load all geometry data for a given map into the agent. Realistically, RRT would also need a scanning function to allow it to dynamically load and unload immediately surrounding world geometry data. RRT utilizes a global node list and as such was not amicable to parallelization. Much of the overhead incurred with RRT-GPU stems from the translation between the data containers used by Numpy and the data containers used by PyCuda. In addition, the size of our data set, and the parameters we chose to use with RRT-GPU along with proper Cuda practices meant that the algorithm was not guaranteed to even run on the GPU in every instance. These factors translated into a slight increase in run time of RRT-GPU compared to RRT.

Given what RRT and RRT-GPU set out to achieve, we can conclude that in two-dimensions both algorithms perform well enough to act as suitable path planning algorithms. Their randomized nature means that unless post-smoothing is applied after the path is generated, the path will not be minimally optimal. Likewise, paths generated by both algorithms are not realistic. Both minimal path length and path realism are exceedingly important in applications involving computer graphics or motion planning.

For applications that involve real-time motion planning RRT and RRT-GPU are significantly faster than A\* for numerous factors. First, A\* must evaluate all adjacent neighbors (that have not already been seen or evaluated) before choosing the best cell for evaluation. RRT does not need to do this, instead it samples randomly form a predefined sampling space and branches to a nearby node. RRT has no evaluation requirement of neighbors. Second, unlike A\* which must move from neighbor to neighbor, RRT can jump around the search space. Paths generated by RRT tend to contain significantly less nodes as a result. Likewise, we can bias RRT with a slight probability to sample the goal immediately. This means that at each iteration RRT has a small chance to instantly find the goal, skipping possibly many, intermediary steps.

RRT does have drawbacks to its use. First and foremost, it requires full knowledge of the search space. A\* can naturally discover the search space as it proceeds through its algorithm. For RRT to exhibit the same behavior, it must be modified with a localized scanning algorithm to force it to sample only from discovered space. This is in fact what RT-RRT\* does with its ellipsis sampling algorithm. Second, it does not produce optimal paths. Paths are both jagged and longer than they need be. We discussed how post-smoothing can alleviate some of the issues created in this process, however, it is not a cure all. Instead we suggest utilizing RRT in three dimensions to generate way points for the A\* algorithm. These way points can act as sub goals for pathing agent which would reduce A\*s ability to be trapped in a localized area of the map.

Finally, we merely focused on parallelizing the nearest neighbor functionality of RRT. Like A\*, RRT is a serial algorithm. Based on our analysis, this functionality was the most logical piece of RRT to parallelize. However, we were also able to determine that doing so yielded no substantial improvements in run time over standard RRT.

### Future Work

This thesis provides a thorough examination of rapidly exploring random trees applied to finding paths in a simulated game environment. This is not the only application of the algorithm. Given the proper constraints, RRT could also be utilized to lay trace paths for printed circuit boards. It could also be utilized as a route planning agent for autonomous vehicles, albeit, with significant restrictions. As it stands, RRT is good at finding paths quickly. However, the resultant paths are non-sensical most of the time. The algorithm does not follow any pathing paradigm such as shortest path or achieving realism. Future research around RRT could be centered on restricting the algorithm to these paradigms for application to previously discussed cases.

Our implementation of this thesis is written in Python with embedded C CUDA code. While we have tried to squeeze as much performance out of the engine as we could, this thesis would greatly benefit from being written in native C++ code with direct use of the CUDA library as opposed to utilizing a wrapper such as PyCuda. Of course, this would only be beneficial alongside an increased search space. Future work involving this thesis should make all attempts to implement this thesis natively if possible.

Finally, we were unable to implement RT-RRT\* as we had originally planned. It turned out to be too complex for what we were seeking to gain. Instead we chose to focus on enhancing RRT with hardware acceleration. We would still like to implement RT-RRT\* and integrate hardware acceleration into that algorithm. In addition, we already know that RT-RRT\* functions fine in two-dimensions, we are not however, certain that RT-RRT\* functions appropriately in three-dimensions. We would like to design a similar set of experiments that assess its application to 3-space.

We believe it is possible to parallelize RRT in entirety. This however, requires world geometry information be loaded onto the discrete graphics card. In addition, the algorithm would require the means to communicate between threads on the graphics card. This is necessary for two reasons: first, all threads need to know the current state of the tree; second, a master thread should exist to pool together the results of the slave threads. This fell outside of the scope of this project and we lacked the necessary mastery of high performance computing to do so, however this is a viable option for future endeavors.

### What I Learned from This Experience

This thesis revolves around computational geometry and high-performance computing. I utilized scientific computing libraries such as NumPy, PyCuda, and Pandas. All the software packages utilized in this thesis carry real world relevance. NumPy and Pandas are used daily in professional environments to analyze vast quantities of data for varied purposes. As a wrapper for Nvidias’ CUDA runtime environment, PyCuda allows users to complete simultaneous complex transformations on data utilizing a discrete graphics card in a fraction of the time it would take a central processing unit. This has proven a valuable experience in gaining insight into the uses of some of the most commonly used libraries in the Python programming environment.

More importantly, this thesis helped enriched my knowledge on parallel programming. This type of programming differs from sequential programming so much that it took me roughly four months just to understand the basics around it. Thankfully, PyCuda saved me from many of the messy underpinnings of working within this paradigm. My first attempts at parallelization were crude manipulations of Nvidia sample programs. As I continued my foray into writing CUDA code by hand, I realized that there had to be a better way. I eventually stumbled upon a set of functions within the PyCuda documentation that encapsulate almost all the setup for accessing the data and allowed me to simply pass the graphics card the data I wanted to work with and a function to apply to it.

The issue then shifted to representing that data in a meaningful way so that I could allow the agent to utilize it. I ended up utilizing PyCudas built in vector objects to pass multiple data points in a single object. This worked well enough, but as mentioned earlier in the thesis, this technique was a contributor to the increase in run time of RRT-GPU over RRT. The other contributing factor was the slowdown introduced by translating the data from the graphics card to main memory. Unfortunately, our implementation resulted in a large amount of transfers between the graphics card and main memory (every eight iterations of RRT-GPU).

## Appendix

### A\* Results

**Map One: Flat Land Pathing Agent**

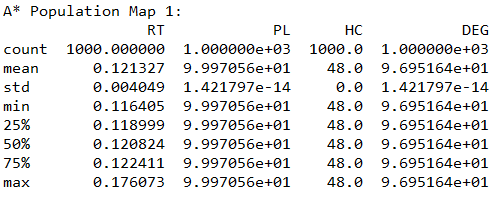


Figure 1: Map One A\* Population Data

**Map Two: Climbing Agent**

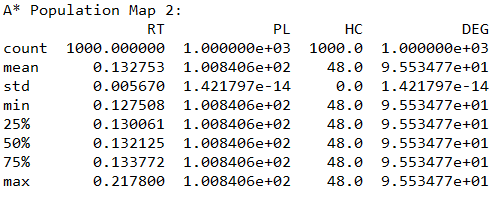


Figure 2: Map Two A\* Population Data

**Map Three: Descent Agent**

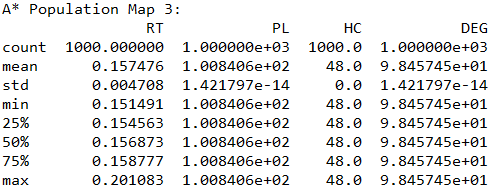


Figure 3: Map Three A\* Population Data

**Map Four: Mixed Agent**

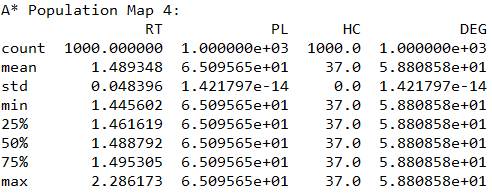


Figure 4: Map Four A\* Population Data

### RRT Results

**Map One: Flat Land Agent**

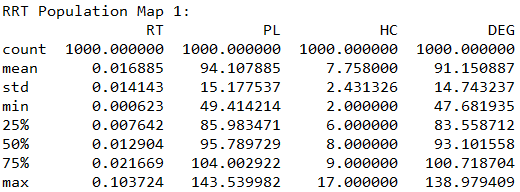


Figure 5: Map One RRT Population Data

**Map Two: Climbing Agent**

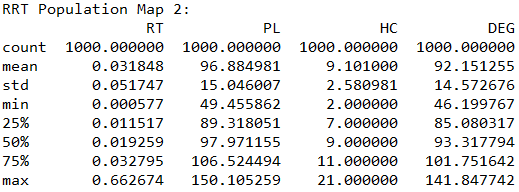


Figure 6: Map Two RRT Population Data

**Map Three: Descent Agent**

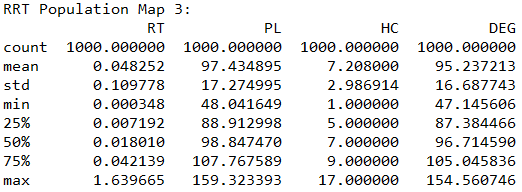


Figure 7: Map Three RRT Population Data

**Map Four: Mixed Agent**

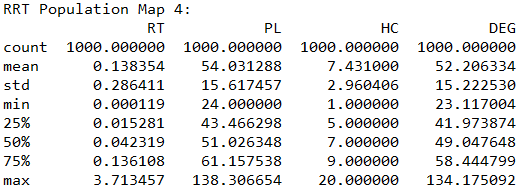


Figure 8: Map Four RRT Population Data

### RRT-GPU Results

**Map One: Flat Land Agent**

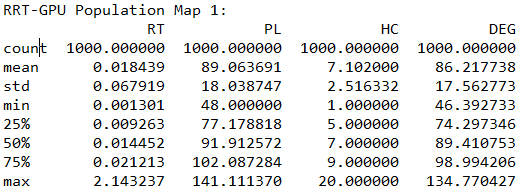


Figure 9: Map One RRT-GPU Population Data

**Map Two: Climbing Agent**

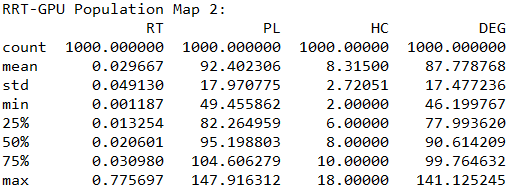


Figure 10: Map Two RRT-GPU Population Data

**Map Three: Descent Agent**

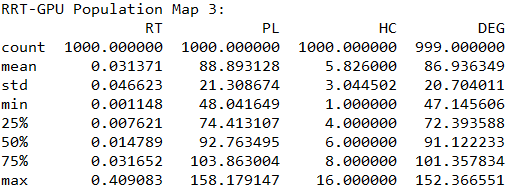


Figure 11: Map Three RRT-GPU Population Data

**Map Four: Mixed Agent**

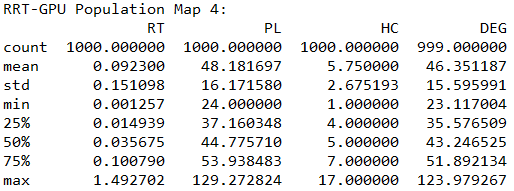


Figure 12: Map Four RRT-GPU Population Data

### F-Test for Equality of Variance: Two Populations

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Map One: Flat Land Agent (α=.05)** | | | | |
|  | **Hypothesis** | **Rejection Region** | **F-Statistic** | **Conclusion** |
| **Run Time** |  | F < 0.901 | 0.0043 | Very Strong Evidence to Reject the Null |
| **Path Length** |  |  | 0.708 | Weak Evidence to Reject the Null |
| **Heading Changes** |  |  | 0.934 | No Evidence to Reject the Null |
| **Total Degrees** |  |  | 0.705 | Moderate Evidence to Reject the Null |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Map Two Descent Agent (α=.05)** | | | | |
|  | **Hypothesis** | **Rejection Region** | **F-Statistic** | **Conclusion** |
| **Run Time** |  | F < 0.901 | 1.109 | No Evidence to Reject the Null |
| **Path Length** |  |  | 0.701 | Moderate Evidence to Reject the Null |
| **Heading Changes** |  |  | 0.9 | No Evidence to Reject the Null |
| **Total Degrees** |  |  | 0.695 | Moderate Evidence to Reject the Null |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Map Three Climbing Agent (α=.05)** | | | | |
|  | **Hypothesis** | **Rejection Region** | **F-Statistic** | **Conclusion** |
| **Run Time** |  | F < 0.901 | 5.544 | No Evidence to Reject the Null |
| **Path Length** |  |  | 0.657 | Moderate Evidence to Reject the Null |
| **Heading Changes** |  |  | 0.963 | No Evidence to Reject the Null |
| **Total Degrees** |  |  | 0.65 | Moderate Evidence to Reject the Null |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Map Four Mixed Agent (α=.05)** | | | | |
|  | **Hypothesis** | **Rejection Region** | **F-Statistic** | **Conclusion** |
| **Run Time** |  | F < 0.901 | 3.593 | No Evidence to Reject the Null |
| **Path Length** |  |  | 0.933 | No Evidence to Reject the Null |
| **Heading Changes** |  |  | 1.225 | Moderate Evidence to Reject the Null |
| **Total Degrees** |  |  | 0.953 | No Evidence to Reject the Null |

### Z-Test: Difference of Population Means

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Map One: Flat Land Agent (α=.05)** | | | | |
|  | **Hypothesis** | **Rejection Region** | **P-Value** | **Conclusion** |
| **Run Time** |  | Z < 1.645 | 0.2360 | No Evidence to Reject the Null |
| **Path Length** |  | 1.96 | 0.0154 | Strong Evidence to Reject the Null |
| **Heading Changes** |  | 1.96 | 0.0129 | Strong Evidence to Reject the Null |
| **Total Degrees** |  | 1.96 | 0.0155 | Strong Evidence to Reject the Null |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Map Two: Descent Agent (α=.05)** | | | | |
|  | **Hypothesis** | **Rejection Region** | **P-Value** | **Conclusion** |
| **Run Time** |  | Z < 1.645 | 0.4902 | No Evidence to Reject the Null |
| **Path Length** |  | 1.96 | 0.0054 | Very String Evidence to Reject the Null |
| **Heading Changes** |  | 1.96 | 0.0787 | Weak Evidence to Reject the Null |
| **Total Degrees** |  | 1.96 | 0.0071 | Very Strong Evidence to Reject the Null |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Map Three: Climbing Agent (α=.05)** | | | | |
|  | **Hypothesis** | **Rejection Region** | **P-Value** | **Conclusion** |
| **Run Time** |  | Z < 1.645 | 0.4981 | No Evidence to Reject the Null |
| **Path Length** |  | 1.96 | 0.0077 | Very Strong Evidence to Reject the Null |
| **Heading Changes** |  | 1.96 | 0.0034 | Very Strong Evidence to Reject the Null |
| **Total Degrees** |  | 1.96 | 0.0077 | Very Strong Evidence to Reject the Null |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Map Four: Mixed Agent (α=.05)** | | | | |
|  | **Hypothesis** | **Rejection Region** | **P-Value** | **Conclusion** |
| **Run Time** |  | Z < 1.645 | 0.4990 | No Evidence to Reject the Null |
| **Path Length** |  | 1.96 | 0.0227 | Strong Evidence to Reject the Null |
| **Heading Changes** |  | 1.96 | 0.0285 | Strong Evidence to Reject the Null |
| **Total Degrees** |  | 1.96 | 0.0230 | Strong Evidence to Reject the Null |

## References

A. Ramires Fernandes and L. Deusdado. 2006. Efficient conservative collision detection for populated virtual worlds. In Ibero-American Symposium on Computer Graphics – SIACG(06), Santiago de Compostela, Spain.

Carlo Camporesi and Marcelo Kallmann. 2014. Computing shortest path maps with GPU shaders. In Proceedings of the Seventh International Conference on Motion in Games (MIG '14). ACM, New York, NY, USA, 97-102. DOI=http://dx.doi.org/10.1145/2668064.2668092

Joseph S. B. Mitchell and Micha Sharir. 2004. New results on shortest paths in three dimensions. In Proceedings of the twentieth annual symposium on Computational geometry (SCG '04). ACM, New York, NY, USA, 124-133. DOI=http://dx.doi.org/10.1145/997817.997839

Kenneth E. Hoff, III, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver. 1999. Fast computation of generalized Voronoi diagrams using graphics hardware. In Proceedings of the 26th annual conference on Computer graphics and interactive techniques (SIGGRAPH '99). ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 277-286. DOI=http://dx.doi.org/10.1145/311535.311567

Kourosh Naderi, Joose Rajamäki, and Perttu Hämäläinen. 2015. RT-RRT\*: a real-time path planning algorithm based on RRT\*. In Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games (MIG '15). ACM, New York, NY, USA, 113-118. DOI: https://doi.org/10.1145/2822013.2822036

Leonel Deusdado, A. Ramires Fernandes, and Orlando Belo. 2008. Path planning for complex 3D multilevel environments. In Proceedings of the 24th Spring Conference on Computer Graphics (SCCG '08). ACM, New York, NY, USA, 187-194. DOI=10.1145/1921264.1921302 http://doi.acm.org/10.1145/1921264.1921302

Michael Burch and Daniel Weiskopf. 2013. The aesthetics of rapidly-exploring random trees. In Proceedings of the Symposium on Computational Aesthetics (CAE '13), Stephen N. Spencer (Ed.). ACM, New York, NY, USA, 45-52. DOI=10.1145/2487276.2487285 http://doi.acm.org/10.1145/2487276.2487285