ASTES

Advances in Science, Technology and Engineering Systems Journal
Vol. 7, No. 4, 49-58 (2022)
www.astesj.com
Special Issue on Innovation in Computing, Engineering Science & Technology

ASTES Journal
ISSN: 2415-6698

ARAIG and *Minecraft*: A Modified Simulation Tool

Cassandra Frances Laffan[*,1], Robert Viktor Kozin[1], James Elliott Coleshill[1], Alexander Ferworn[1], Michael Stanfield[2], Brodie Stanfield[2]

[1]*Computational Public Safety Lab, Department of Computer Science, Toronto Metropolitan University, Toronto, M5B 1Z4, Canada*

[2]*Inventing Future Technologies Inc. (IFTech), Whitby, L1N 4W2, Canada*

ARTICLE INFO

Article history:
Received: 10 May, 2022
Accepted: 19 July, 2022
Online: 27 July, 2022

Keywords:
Haptics
Octree
Pathfinding
*A**
Search and Rescue
Digital Games

ABSTRACT

Various interruptions to the daily lives of researchers have necessitated the usage of simulations in projects which may not have initially relied on anything other than physical inquiry and experiments. The programs and algorithms introduced in this paper, which is an extended version of research initially published in ARAIG And Minecraft: A COVID-19 Workaround, create an optimized search space and egress path to the initial starting point of a user's route using a modification ("mod") of the digital game Minecraft. We initially utilize two approaches for creating a search space with which to find edges in the resulting graph of the user's movement: a naive approach with the time complexity of $O(n^2)$ and an octree approach, with the time complexity of $O(nlogn)$. We introduce a basic A algorithm to search through the resulting graph for the most efficient egress path. We then integrate our mod with the visualization tool for the "As Real As It Gets" (ARAIG) haptic suit, which provides a visual representation of the physical feedback the user would receive if he were to wear it. We finish this paper by asking a group of four users to test this program and their feedback is collected.*

1 Introduction

As most people, readers and the general populace alike, are aware, the COVID-19 pandemic has hampered the plans of many researchers [1]; moreover, ongoing supply chain interruptions have compounded this issue. From research using specialized equipment to field testing, the past two and a half years have been full of ingenious "workarounds" to both workplace restrictions and material shortages. We feel our current project is no exception: in a time where our research, as introduced in our previous work [2], should be applied to tangible hardware and tested in physical environments, access to our lab and materials is limited. The need for corporeal results is clear and thus, we propose and implement a workaround in this paper. This work is an extended version of previously published conference proceedings: *ARAIG and Minecraft: A COVID-19 Workaround* which may be found here [3].

As touched upon in [2], the field of search and rescue is going to experience various changes over the coming decades. Canada, home to both this project and its researchers, is one of many countries currently feeling the metaphorical and literal heat of climate change's effects [4]. Natural disasters, such as the forest fires plaguing the Canadian prairie provinces [5, 6], will become more prevalent across the country and globally before climate action takes effect [4]. Thus, now is the time to act in terms of curbing future disasters while empowering emergency workers to safely and efficiently respond to high impact, low (though increasing) frequency events. While we cannot influence government actions, domestically or globally, we can certainly do our best to assist in preparing our first responders for future disasters.

One of the most dangerous situations a firefighter may face in disaster environments, particularly in enclosed spaces such as homes and buildings, is potentially growing disoriented and thus, lost [7]. In fact, there is ongoing research on this exact issue [7], as this is still an unsolved problem. The motivation for this research, in conjunction with the increasing incidence of natural disasters, is this issue: how can we utilize modern technology in a lightweight fashion to assist firefighters in navigating out of these low visibility environments? Due to the restrictions the pandemic has brought about, as well as ongoing supply chain issues, the initial answer to this question comes in the form of simulation.

Minecraft is a digital game which focuses on the exploration, and building, of randomly generated environments. The graphics, game mechanics and game world are simple: the player is placed, without warning or preamble, into the *Minecraft* environ-

[*]Corresponding Author: Cassandra Frances Laffan, George Vari Engineering and Computing Centre, 245 Church Street, +1 (647) 983-4070 & Cassandra.Laffan@ryerson.ca

ment, which is composed of "blocks", not dissimilar to voxels. The game is centred around navigating, manipulating and accumulating blocks while exploring a procedurally generated world. Since the world of *Minecraft* is so simple, and there are numerous frameworks which support modding the game (these programs will be referred to as "mods"), we have found it suits the above requirements for a simulation medium given our problem domain. Moreover, the game has a well established and active "modding" community [8, 9].

2 Related Work

There has been considerable research into simulations over the past few years, particularly for reasons outlined in this paper's introduction. It is of note, however, that the usage of digital games as an avenue for simulating experiments predates the pandemic and supply shortages. "Project Malmo", a project published in 2016 by Microsoft, is an earlier example of this. The authors write an Application Programming Interface (API) and abstraction layer for *Minecraft*. With it, they train an artificial general intelligence (AGI), to complete various tasks [10, 11]. The inclusion of [10] in our *Related Work* is due to their motivations for using *Minecraft* as their training medium:

- *The environment is rich and complex, with diverse, interacting and richly structured objects [10, 11].* The platform must offer a worldspace with which a user, or otherwise autonomous agent, can interact. This worldspace must be varied and robust.
- *The environment is dynamic and open [10, 11].* The platform needs to offer unique settings which allow us to mimic real-world environments.
- *Other agents impact performance [10, 11].* Other agents, such as AI or other humans, should be able to impact the simulation.
- *Openness [10, 11].* The platform should be cross-platform and portable.

Our ongoing research does not necessitate training AI in *Minecraft*. However, the above guidelines summarize why we believe the game is an optimal platform on which we can test our algorithms and simulate our experiments. The third point in the above list, touching upon how other agents impact performance, is discussed again in various sections of this paper.

Research into simulation, space division and construction of point clouds, graphs and pathways is limited, especially in the realm of search and rescue. Thus, we explore a more generalized approach to 3D pathfinding which has a lower time complexity than more conventional approaches to navigating Euclidean space. In that regard, the authors in [12] explore pathfinding using 3D voxel space in the digital game *Warframe*.

In [12], they propose utilizing an octree, comprised of voxels or "octants", to split the 3D world of *Warframe* into a low time complexity, searchable space which allows for more efficient pathfinding. Every octant is the centre point of a corresponding voxel. The program explores all 26 nearby voxels to find the next best space to move to. The authors determine the next available voxels via the following constraints: are there obstacles in the closest voxel? If so, the voxel is left out of the potential path as the agent cannot occupy the same space as another object. Is the voxel out in the "open", away from cover? It is undesirable for an agent to be out in the open, as it leaves it vulnerable to enemy attacks.

In [13], the authors implement an approach to processing point clouds in Euclidean space, much like what we are attempting to accomplish in our own research; in this case, they wish to navigate through buildings and other structures. The researchers observe that sorting and naively navigating through unordered point clouds can have needlessly high time and space complexities. They propose using an octree to circumvent these issues, allowing new buildings to be mapped internally with a lower demand for computation time. It should be noted that the point clouds these researchers are using are already constructed before data processing; our research differs in that, not only are we constructing the point clouds as a user navigates through the worldspace, but points are not pre-processed.

The authors in [13] use these point clouds to determine the location of obstacles and throughways, such as furniture or doors, respectively. They concern themselves more with identifying and labelling specific objects and terrains, such as stairs, whereas we are more concerned with efficient navigation. More precisely, our focus is on whether or not the firefighter's elevation has changed, as well as the most efficient egress path through a given point cloud and resulting graph.

Further on in this paper, we implement an A* algorithm for searching the resulting graph with appropriate edges drawn between nodes. Two papers which are referenced in this publication for the implementation of A* are: *Algorithms and Theory of Computation Handbook* [14] and *Artificial Intelligence: A Modern Approach* [15]. Both publications act as guidelines for the implementation of not only A*, but other algorithms which are discussed in the *Future Work* section. The latter also provides guidance for both the time and space complexities of A*, which are necessary to explore given the problem domain.

3 Methodologies

The next sections are divided and ordered in a way mimicking the timeline of this project. First, methods for modifying *Minecraft* are outlined, as this is a necessary step for producing meaningful datasets and egress settings for our experiments. Next, we explore dividing the worldspace of *Minecraft* and ensuing datasets into a searchable graph with edges (at times referred to in this work as "adjacencies"). Splitting the worldspace and creating a graph in a timely manner emulates the urgency necessary in the real world environment for firefighters. Pathfinding through the graph follows the graph creation, as is the logical progression of events; again, efficiency in time complexity is discussed since urgency is one of the most important factors in creating the egress path for a first responder. Navigating through the *Minecraft* worldspace follows, alongside using the ARAIG visualization tool. Creating directions and output for the suit, while important, needs to be done with the intention of making them intuitive for first time users. Thus, a short survey is conducted with a small group of users to get initial

feedback on the simulated system.

4 Modding *Minecraft*

Minecraft follows a server-client network model. This means there are two avenues for modding the game: server-side or client-side [16]. The server-side handles logic, game state and updates from clients. The client-side handles rendering the game state and sending updates. This separation of concerns is important. If we wish to illustrate this concept in *Minecraft*, a creature, its location and where it moves is handled by the server. The information about said creature is sent to the client where it is rendered. The client can also send updates, for example, when the player wishes to perform an action such as jumping or hitting a block. In this respect, as it can only modify what it has access to, the client cannot control where the aforementioned creature is and the server cannot control how the aforementioned creature appears.

Server-side modding tends to be relatively straightforward, since the network interface is well understood and documented. There are numerous "community sourced" implementations and application programming interfaces (APIs) which extend the *Mincecraft* server. The API we use for this mod is Spigot [17], which provides a way to run code on events, such as player movement. It also enables programs to react to said events, namely cancelling the movement, recording coordinates or even running actions, such as smiting the player with a bolt of lightning.

Client-side modding is often more complex than its server-side counterpart. To extend the game, client-side mods have to "hook" directly into the "vanilla" client using a variety of complex methods. An example is runtime "bytecode" manipulation, where the Java runtime environment is used to modify compiled code while it is running [9]. *Minecraft* client code is often complex in nature as it deals largely with in-game rendering. In addition, the code is obfuscated, making it difficult to read and understand.

There are two big projects that make modding *Minecraft* easier: Forge [18] and Fabric [8]. Forge is more established and has a larger scope of supported modding functionalities. However, the consequence of this is that the framework takes longer to update and is less light-weight. Fabric, in contrast, is newer and lighter, using modern techniques and providing more low level control. In this project, we opt to use Fabric, as we prefer to keep the mod lightweight, allowing for a more agile approach to mod development.

Originally, we attempted to create a server-side mod. Our goal was determining if location data and its visualization is viable in *Minecraft*. Despite our success in proving its viability, the limitations of server-side modding in regards to visualization quickly made themselves known. As previously mentioned, the server has limited control as to what the client "sees". While the server can spawn creatures or create particles for visualization, we require more customization. Given these circumstances and our requirements, the client-side model is best suited for our mod. This allows us to use the same code to render our custom visualizations that the client uses to render the game.

Our *Minecraft* mod visualizes a graph data structure by drawing the edges as lines and the nodes as numbers in the *Minecraft* worldspace. This gives the user the ability to visualize the recorded coordinate points and the connections created between them. Coordinate points closer to the real world are now easier to collect; there is no need for random numbers or predetermined coordinates.

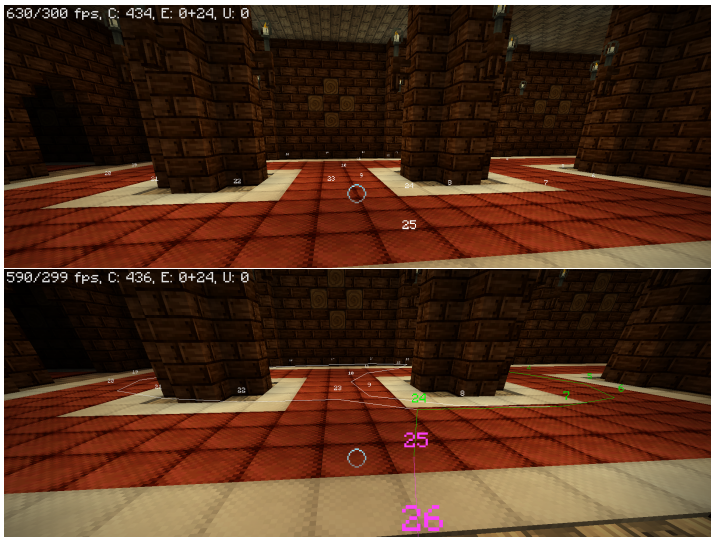

Figure 1: An example of how the *Minecraft* mod visualization tool renders the player's pathway before and after the pathfinding algorithm is run.

As stated above, *Minecraft* is a Java based game. However, in order to seamlessly interface with the ARAIG simulation software, the pathfinding program is in C. Thus, the *Minecraft* mod and the pathfinding algorithms must be split into separate programs, with communication being done over a network socket using a simple protocol. This protocol can be described as a request-response byte encoded message protocol. The first byte is the message type and the rest is the message body in the request. In contrast, the reply is comprised of only a message body. For example, to get the current location from the mod, the pathfinding program sends a *GET_LOCATION* byte identifier. The server then responds with 4 big-endian byte encoded floating point numbers containing to the x, y, z, and yaw components respectfully.

The control flow of the mod is as follows:

1. **Capture location data:** The mod has two commands, */start* and */stop*, the user can enter in the "chat bar" to control the recording of coordinate points. This emulates a GPS device by providing the coordinates of a wearer to an external machine. The location data is stored as a list of coordinate points. While the user is recording his path, a numbered node is placed in the worldspace and recorded as a set of coordinates.

2. **Transfer captured location data:** The pathfinding program then requests all of the recorded coordinate points from the mod, allowing processing to begin.

3. **Transfer live location data:** Additionally, the pathfinding program queries the user's live location data from the client. This allows for live tracking of the user along the path so that egress instructions are updated accordingly in real time.

4. **Send draw updates:** Finally, in response to the captured and live location data, the pathfinding program sends draw

commands to the client. The commands include: drawing or deleting a line between points, changing the colour of a given line, and changing the colour of the points.

While our initial belief of having the data collection and visualization as separate from the pathfinding program may result in unnecessary complexity, the resulting mod is, in fact, the opposite. Using two separate programs for data processing and data visualization allows the code to be more organized and decoupled. One of the benefits for this is the pathfinding program could be restarted and debugged without needing to restart the *Minecraft* client. Since the two are decoupled, both can be developed at different paces.

5 Creating the Graph

Two approaches are outlined here for graph creation, which is necessary for finding an egress path given points in 3D space: a naive approach which has a time complexity of $O(n^2)$ and an octree approach which runs in $O(nlogn)$ time.

5.1 Naive Implementation

Algorithm 1 is the initial approach for creating the shortest egress pathway possible given the user's nodes.

Algorithm 1: *add_node*(*graph*, *node*) Adds a node to the graph. Then, adds it to the adjacency arrays of any existing nodes within a radius *R*.

```
Input: A graph with all previous nodes already inserted
       and connected: graph, a newly created node to
       insert into the graph: node
foreach existing_node ∈ graph.nodes do
    if distance(existing_node, node) ≤ R or
     existing_node == graph.nodes[−1] then
        /* Appends the node to the adjacency
           array of a given node in the graph
           */
        existing_node.adjacencies ← node
end
graph.nodes ← node
```

When a new node is created, the *add_node* function is called to add it to the graph. The node is compared to every existing node already in the graph; if the node is within *R* radius of the node it is being checked against, it is added to the second node's adjacency list. This approach is simple and intuitive, yet inefficient. As mentioned previously, its time complexity is $O(n^2)$. The algorithm works as a proof of concept but it quickly becomes evident that with sufficiently large n, a more efficient approach is necessary. See Figure 2 for a visualization of this function.

5.2 Octree Implementation

An octree object has three member variables associated with it. First, the *bounds* variable, which contains the boundaries for the 3-Dimensional (3D) box that the octree resides in; these are stored as a set of coordinates. Next, it contains a *children* array. This array contains either zero or eight octants. These octants are the result of splitting the *bounds* object into eight equally sized boxes.

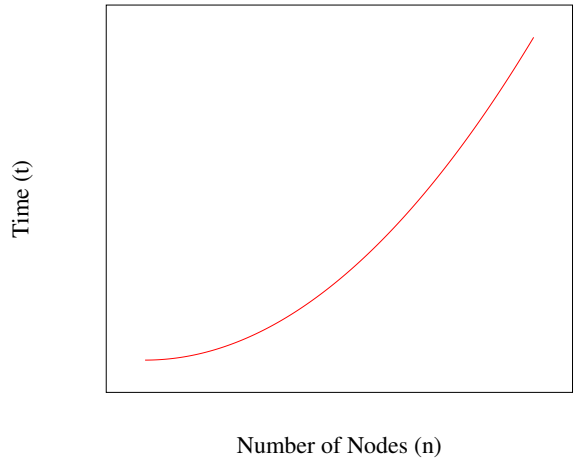

Figure 2: Plotting number of nodes n against time t where $t = n^2$.

Algorithm 2: *make_octree*(*nodes*, *bounds*) Creates an octree given a set of points in 3D space.

```
Input: An array of nodes: nodes, the current boundaries for
       the octree: bounds
Output: An octree containing either eight octree "children"
        (octants) or N ≥ leaf nodes
initialize octree
initialize boundaries
if !nodes then
    octree.children ← NULL
else
    if length(nodes) ≤ N then
        octree.nodes ← nodes
    else
        /* Splitting the space into octants  */
        boundaries ← split(bounds)
        initialize octree.children
        for i = 0; i ≤ 8; i + + do
            initialize next_nodes
            foreach node ∈ nodes do
                if node ∈ boundaries[i] then
                    next_nodes ← node
            end
            octree.children[i] ←
              make_octree(next_nodes, boundaries[i])
        end
    end
end
return octree
```

Finally, an octree object has a *nodes* array; it remains empty unless the octree object is a leaf in the greater data structure. It should be noted that, unlike the naive approach above, this algorithm is run after the subject has ended the path recording. The pseudo-code for the octree creation is illustrated in Algorithm 2.

The initial octree object is created with its bounding box characterized by the minimum and maximum (x, y, z) coordinates of the whole graph. The function *make_octree* is then called on an array

of every node and the aforementioned boundaries. Then, one of three things must happen: first, if the array of nodes is empty, the octree's *children* array is set to $NULL$ to indicate that it is empty. Second, if the array of nodes exists, and contains less than or equal to the amount of allowed nodes in a given octant space, the octree's *nodes* array is populated and the octree object becomes a "leaf".

Finally, if neither of the previous two conditions are true, the function recursively calls itself. A new set of octants is created by splitting the boundaries into eight identically sized cubes. Next, for each new octant, each node is compared to its bounds and, if it is contained within the octant's boundaries, it is placed in a new array. Once the array is populated with the appropriate nodes, the *make_octree* function is recursively called on it and its respective boundaries. The resulting octree is added to the *children* array for the current octree.

The construction of this octree creates an efficient, though spatially complex, search space which allows for the appropriate edges in the graph to be created. The pseudo-code for edge creation, as a series of octree searches, is in Algorithm 3.

Algorithm 3: $find\_adjacencies(octree, node)$ Populates a node's adjacency array with nodes within a radius R.

```
Input: An octree object: octree, the current node that we
       wish to find the adjacencies for: node
if octree.nodes then
    foreach oct_node ∈ octree.nodes do
        if node ∉ oct_node.adjacencies and
          distance_between(oct_node, node) < R then
            node.adjacencies ← oct_node
    end
else
    for i = 0; i < 8; i++ do
        if node ∈ octree.children[i].bounds then
            find_adjacencies(octree.children[i], node)
    end
end
```

As Algorithm 3 demonstrates, $find\_adjacencies$ takes a fully formed octree and a node as arguments. It is called on every node in the graph once. The function checks if the octree object is a leaf, and, if it is, the selected node is compared to each node in the octree's *nodes* array. If any nodes are within the given distance R, they are added to the adjacency array for the node. Conversely, if the octree object is not a leaf in the octree, the bounds for each member of its *children* array are checked against the coordinates of the node. If the node is within the given child's bounding box, the $find\_adjacencies$ function is called on the child and the node. This is done recursively until all of the appropriate octree branches for a given node are explored.

It should be noted that the *average* case runtime for creating the octree and recreating its path is $O(nlogn)$, where n is the total number of nodes in the system. The worst-case runtime for this octree approach is the same as the above *naive implementation*, which is $O(n^2)$. The worst case for this algorithm occurs if and when a node is compared to every leaf in the octree and thus, every other node in the system.

The bottleneck for efficiency in this case is not the creation of the octree, which is in fact linear time, nor is it any given single search of the octree, which is also $O(logn)$. Rather, the bottleneck is that the search must be performed for each node in the graph. Thus, the time complexity for this algorithm is $T = n + nlogn$, which is the sum of both the runtime of the octree construction and the creation of each node's adjacency list. Idiomatically, the runtime is $O(nlogn)$. See Figure 3 for a visualization of this function.

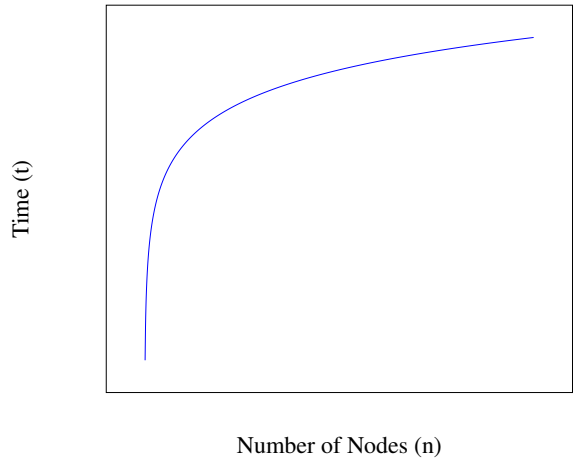

Figure 3: Plotting number of nodes n against time t where $t = nlog(n)$.

6 Egress Path Creation

This section assumes one of the previous two graph creation algorithms was run and there now exists a graph with appropriate nodes and their respective adjacencies. Algorithm 4 creates a path back to the first node of the graph for the user. See Figure 4 for an example overview of a reconstructed path.

Algorithm 4: $create\_pathway(graph)$ Creates a stack containing the nodes comprising the most efficient path back to the beginning of the graph.

```
Input: A graph with appropriate adjacencies already
       created: graph
initialize stack
initialize node ← graph.nodes[0]
while node ≠ graph.nodes[−1] do
    stack.push(node)
    node ← node.adjacencies[−1]
end
/* Gives the directions to the user in order
   */
initialize next_node
while stack do
    next_node ← stack.pop
end
```

6.1 Naive Search

Algorithm 4 navigates through the graph by "jumping" to the last neighbour in the current node's edges. Both of the previous algorithms place the latest recorded neighbour at the end of the adjacency array. The naive algorithm accomplishes this by appending new

nodes to the graph as they are created and updates the adjacency arrays accordingly by comparing each existing node to the new node. The octree algorithm emulates this behaviour by comparing a node's ordering ID to the neighbour's ID. If the node's ID is less than the incoming neighbour's ID, the node is appended to the adjacency array. Thus, a stack is the only thing necessary when recreating the shortest path with which the user can navigate back to the beginning of the graph. The time complexity for this approach is linear, $O(m)$, where m is the number of nodes in the egress path; the worst case runtime is $O(n)$, where n is the number of nodes in the graph.

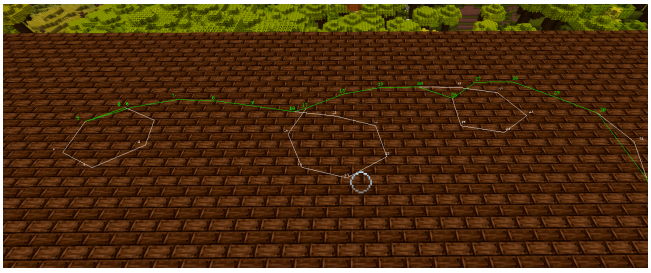

Figure 4: An aerial view of a reconstructed path.

Algorithm 5: A* Search

Input: A graph with all adjacencies drawn: *graph*
Output: A path containing the nodes from the user to the closest goal node: *path*

```
initialize priority_queue
priority_queue.enqueue(start_node, 0)
while !priority_queue.is_empty do
    u ← priority_queue.deqeueue
    if u == graph.end then
        final_node ← u
        break
    else
        foreach adjacency ∈ u.adjacencies do
            if adjacency.g + distance(adjacency, u) <
              adjacency.g then
                adjacency.previous ← u
                adjacency.g ←
                  u.g + distance(adjacency, u))
                f ← adjacency.g + adjacency.h
                priority_queue.enqueue(adjacency, f)
            if adjacency.visited then
                adjacency.reExpansions + +
            else
                adjacency.visited ← true
            end
        end
    end
end
return path(start_node, final_node)
```

6.2 A* Search

The issue with the previous implementation is that it does not take into account a future feature for this system: multiple users. Even worse, there are some instances of path creation in which jumping to the highest neighbouring node will not actually create the most efficient pathway back for a single user. An edge case which would cause this undesirable behaviour, for example, is if the user continuously navigates in "loops" or interconnected circles. Algorithm 5 is a basic implementation of A*, which replaces the naive approach above. While the naive approach is linear in its time complexity, the average time complexity for A* is $O(b^d)$, where b is the branching factor and and d is the depth of the solution [15].

The heuristic function utilized by Algorithm 5 is the Euclidean distance value from any given node to the final node. This allows for prioritization of nodes, leading the user in a straighter line towards the goal. This heuristic, given only one goal node, is admissible [14], meaning it does not overestimate the cost of a given node to the goal node [14].

7 Giving Directions in *Minecraft*

This section assumes the egress path has already been constructed. As the *Modding Minecraft* section states, the game was created and released in the early 2010 [19]. The programmers behind the original Java-based game made some unusual implementation decisions, particularly in regards to its coordinate system. In essence, it is a right-handed system with unconventional axes. The three issues which we circumvent later in this paper, in order to provide accurate navigation, are:

1. The y-axis is the measure of how high or low the player is relative to the ground, as opposed to the conventional z-axis for this task;
2. The angles between points in the XZ-plane are given clockwise, instead of the conventional counter-clockwise that is generally utilized for trigonometry;
3. The positive Z axis, which is South, is 0 radians in *Minecraft*. Consequently, North, which is negative Z, is π radians.

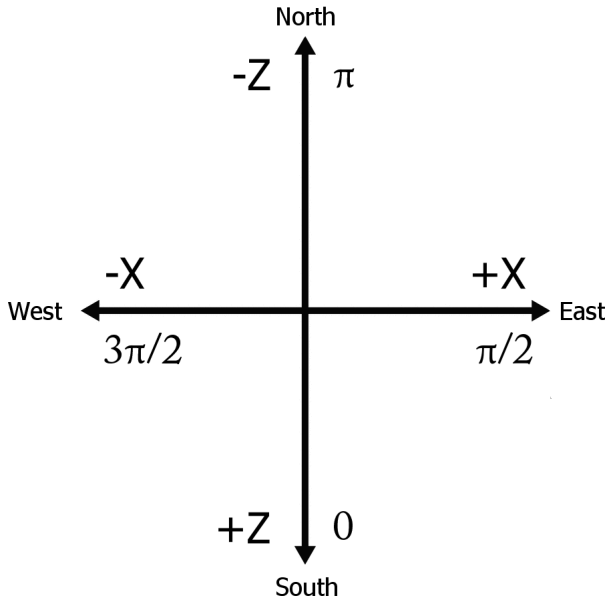

Figure 5: A diagram of the *Minecraft* coordinate system.

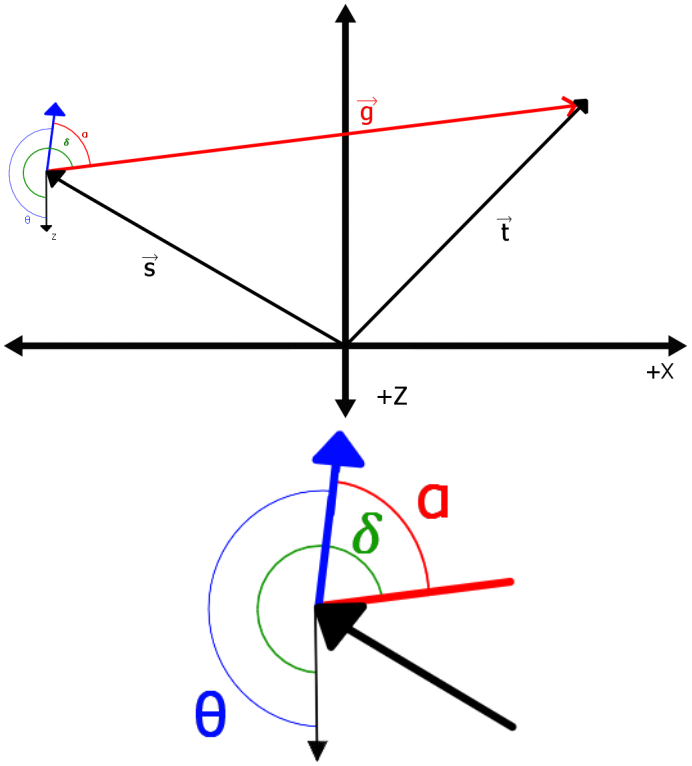

Figure 6: An example of the resulting system of vectors when a player's yaw is facing a different direction than the next desired node. θ and the blue arrow are the player's yaw. $\vec{t}$ is the node vector, $\vec{s}$ is the player's position vector.

Once the shortest path has been created for the player, we need to give him directions to the next immediate node in that path. To do this, we calculate the angle of the next node relative to the player's yaw. We shall call this angle α. First, we take the vector $\vec{t}$, which is the vector pointing to the desired node, and the player's position vector, $\vec{s}$, and calculate $\vec{g}$, which we will refer to as our *direction vector*:

$$\vec{g} = \vec{t} - \vec{s}$$

Once we have our direction vector, we calculate β, which is the angle of $\vec{g}$ relative to the native coordinates to *Minecraft*. For this, we use 2-argument arctangent:

$$\beta = atan2(-g_z, g_x) + \pi$$

Once β is calculated, we have to adjust the angle so that it is relative to the flipped coordinate system that we are now working in. We will call this angle δ:

$$\delta = 3\pi/2 - \beta$$

The final step to find the next desired direction is to calculate the difference between the resulting δ and θ, where θ is the player's yaw. This is α:

$$\alpha = \delta - \theta$$

For the sake of convenience, in order to ensure that the resulting angle is easy to use when giving directions from the suit in the next node, we may add 2π to normalize the value of α. Please refer to Figure 6 for a visualization of an example in this system. Thus, the algorithm used to calculate the next desired directional instruction, relative to the player, is outlined in Algorithm 6.

Algorithm 6: Returns the angle of the node relative to the player's position and yaw.

Input: A player's coordinates and yaw: *player*, the desired node: *node*

$g_x \leftarrow node.x - player.x$
$g_z \leftarrow node.z - player.z$
$\beta \leftarrow atan(-g_z, g_x) + \pi$
$\delta \leftarrow (3\pi)/2 - \beta$
$\alpha \leftarrow \delta - player.yaw$
if $\alpha < 0$ **then**
  $\alpha \leftarrow \alpha + 2\pi$
return α

A public repository including these algorithms may be found in [20]. The functionalities which are specific to the ARAIG suit, as discussed below, have been removed for the sake of licensing.

8 Integration with ARAIG

The "As Real As It Gets" (ARAIG) suit, as outlined in IFTech's specifications in [21], has numerous vibratory and stimulus sensors. In order to provide a distinct set of visual cues on the simulated suit, we utilize the vibratory sensors. This way, the player can quickly translate the visual instructions from the simulated suit to following directions inside the *Minecraft* environment. To ensure the system was quick to learn for new users, the program only outputs four directions (which are given in relation to α as calculated in the previous section):

1. **Forward:** $\pi - 1/2 < \alpha < \pi + 1/2$. The user's abdomen and pectorals are stimulated, indicating to them that they should move forward.

2. **Left:** $\pi/2 - 1/2 < \alpha < \pi - 1/2$. The user's left shoulder is stimulated, indicating to them that they should turn left.

3. **Right:** $\pi + 1/2 < \alpha < 3\pi/2 + 1/2$. The user's right shoulder is stimulated, indicating to them that they need to turn right.

4. **Turn around:** If α is not within the previous three ranges, the user is not facing the correct direction. Thus, the user's back is stimulated, prompting them to turn around.

Once the program is running, the user is given a set of initial instructions. The directions relayed to the user are updated every 500 milliseconds given the user's yaw and location. Taking the conditions outlined in Figure 6 as an example, the user's suit output would appear on the screen as shown in Figure 1.

Figure 7: An example of how the ARAIG suit simulation software appears given the conditions in Figure 6.

The ARAIG simulation software integration has been omitted in the supplementary GitHub repository. If readers wish to use this software, they are encouraged to reach out to IFTech at [22]. Once readers receive the appropriate permissions to use the ARAIG visualization software and SDK, they are free to contact Cassandra Laffan or Robert Kozin for access to the full version of this mod and its functionalities.

9 User Testing

A small series of tests were designed to examine whether or not this system is intuitive and quick to learn for new users. The sample size for this study is limited by the non-disclosure agreement (NDA) which protects the ARAIG visualization tool. Consequently, users tested in this study are only those with access to the researchers' machines. As a result, we survey four users of varying backgrounds, all of whom have access to one of the computers with the visualization software available.

Figure 8: Screenshots of the burn house as constructed in *Minecraft*.

The user tests take place in a "burn house", which is a structure built to the specifications as outlined in [23]. Burn houses are standardized buildings in which firefighters may practice navigating structures and fighting fires in a physically simulated environment here in North America. This is a logical testing environment as the system is being designed with first responders in mind. The main goal of this "pilot study" is not necessarily to evaluate how quickly a user may exit a building given different circumstances, but to observe how the average user interacts with the system. We also gather feedback on possible system improvements in hopes of making it more intuitive to new users.

There are four categories of navigation tests, all of which have the same set up and goal: the user is tasked with navigating to the top of the tower, then retracing their pathway down. The navigation upward is not timed as it generally took 60 seconds±1 second; what was timed was the user navigating back to where they started. The users are told it is acceptable to both stray from the path if they were lost or quit for the same reason. The categories for testing are as follows:

1. **Control Run:** The control run times the users navigating to their starting points at the bottom of the tower with high visibility and no navigational assistance.
2. **Low Visibility:** This run is much like the control run, with no navigational assistance. However, distance of visibility for the users is greatly decreased, as per Figure 8.
3. **High Visibility with Path Recreation:** This category of testing allows the users their full field of vision. Their goal of retracing their path is assisted with output on the simulated ARAIG suit on a neighbouring screen. The users are informed that they could opt not to acknowledge the suit's output if they find it to be confusing or a hindrance to their task.
4. **Low Visibility with Path Recreation:** This set of tests has the users navigate up to the top of the tower and back down with low visibility. They are given the output of the simulated ARAIG suit on a neighbouring monitor to assist them in this task. Much like in the previous category, they are informed that if they felt the suit is acting as a hindrance to their task, they can opt to ignore its output.

Figure 9: An example of a low-visibility environment created by our mod.

Users are also asked to give any and all feedback they believe is pertinent to the experiment.

10 Results

Users were given time to practice controlling the player character in the game before running the tests. The results for each run are

shown in Figure 10. Generally, the users were more efficient when not following the directions given by the suit in high visibility situations. However, whether or not the use of the suit made it easier for the user to find their way back to the start in low vision environments seemed entirely dependent on the user's experience with *Minecraft* in the past and the path they took. This will be further discussed below.

	User A	User B	User C	User D
Control Run with High Visibility	0:51:12	0:23:03	0:47:29	0:19:03
	0:53:2	0:55:07	0:22:16	0:49:11
	0:52:5	0:45:13	59:33:00	0:39:33
Control Run With Low Visibility	1:35:14	1:02:29	0:17:35	0:26:27
	1:30:24	0:39:1	1:12:07	1:10:23
	1:25:29	59:18:00	1:36:09	1:35:16
High Visibility with Path Recreation	0:61:3	54:26:00	1:17:42	0:22:16
	0:56:4	1:46:48	2:57:00	1:05:12
	0:55:6	0:21:17	2:17:59	0:58:45
Low Visibility with Path Recreation	1:15:03	1:42:14	1:30:04	0:46:06
	1:20:32	1:26:45	1:51:01	1:23:36
	1:12:12	1:40:28	36:09:00	1:28:29

Figure 10: The results of our short pilot study. Measurements are given in Minutes:Seconds:Milliseconds.

User feedback was as follows:

- Users A, B and D all remarked that having the suit in the low vision environment made navigating somewhat easier if they did not initially take erratic paths.
- Users B, C and D all said that dividing their attention between *Minecraft* and the ARAIG simulation software made navigating efficiently very difficult.
- Users B and C insisted that completing the task would be much easier while wearing the suit.
- User B suggested that instead of stimulating the user on the back to prompt him to turn around, to instead have it indicate that the user should go forward. This would emulate a "pushing" motion.

When we asked users if they would find the system useful when wearing the physical ARAIG suit, all of them responded that yes, it would be more helpful.

11 Discussion

On average, the time taken to navigate the high visibility portion is shorter unassisted versus navigating with the assistance of the simulated ARAIG suit. This is supported by the feedback from most users: splitting their attention between two monitors may be distracting and much more difficult than simply guessing their return path. The shortest pathways, as reflected in Table 1, are either the user taking advantage of the fact there were a few optimal routes to the top of the building from the ground, or them falling down multiple flights of stairs.

Users A and D both have previous experience playing *Minecraft*. They found the ARAIG suit's contributions to their navigation back to their starting point to be beneficial. Users B and C, on the other hand, expressed that the simulated suit detracted from their ability to navigate, as they were already focusing heavily on how to navigate in the low light environment. Shorter pathways up, which were generally just a simple race up the stairs, are reflected in the low visibility runs.

12 Conclusion

In this project, we continue the work we first presented in [2], where we propose a system for assisting first responders in navigating out of low visibility environments utilizing the ARAIG haptic suit. In order to circumvent various obstacles due to the pandemic and supply chain interruptions, we opt to simulate the ARAIG functionality and path recreation in the digital game *Minecraft*. To do so, a mod for the game integrating the ARAIG visualization software is written. This mod tracks a user's movement through the *Minecraft* worldspace. Once the player's points in space are recorded, two different implementations of creating a graph from these points are proposed: the naive approach, which directly compares every node to every other node, and an octree approach, which divides the worldspace into octants, allowing for efficient edge creation. Then, two search algorithms are implemented and compared for finding the most efficient egress path in the resulting graph. The first implementation is another naive approach, which simply jumps from a node to the last recorded node in its edge array. The second implementation is a basic A* algorithm, which, while having a higher time complexity in a worst-case scenario, does not fail in special circumstances.

Finally, we have four users test our software and give us constructive feedback based upon their experiences playing the game in conjunction with the visualization tool for the ARAIG suit. Users generally agree dividing their attention between two programs is difficult and the task would be much easier to complete if wearing the physical ARAIG suit. This feedback is useful for when our research evolves to include firefighters, as we do not want firefighters to feel as if they are disconnected from, or do not understand, the physical input from the suit.

12.1 Future Work

Succeeding this leg of our research are various steps we plan to implement. First and foremost, as we note above, we are looking to implement multi-user functionalities. In doing so, further usage of A* is necessary and the naive implementation for finding the egress path simply will not work. There are numerous variants and alternatives to A* [14], including iterative deepening A* (IDA*)

and recursive best first search (RBFS). The next immediate step is thus finishing the *Minecraft* mod, implementing various search algorithms and comparing them on large datasets.

Before we begin testing in real-world environments, we intend on bridging the gap between these two concepts. As our *Results* section mentions, users lamented having to split their attention between two screens while navigating with the simulated ARAIG suit. Further testing in the *Minecraft* environment includes having a user wear the physical suit while navigating the game world. This would allow the user to focus his full visual attention on *Minecraft* while receiving instructions from the ARAIG garment.

Other steps in this research include integrating the algorithms with software which interfaces with the physical world, such as Google's AR Core API [24]. Using our algorithms in more controlled, less noisy real-world environments will allow us to refine these algorithms before we begin integrating them with noisier data such as LiDAR, sonar or a 3D camera data. Finally, we can eventually begin work on integrating these algorithms fully with the physical ARAIG suit and supplementary sensors, as first introduced in [2]. We will then test this technology in a physical burn house as presented in [23].

Conflict of Interest The authors declare no conflicts of interest.

Acknowledgements The researchers acknowledge the funding provided by the Toronto Metropolitan University in conjunction with the Natural Sciences and Engineering Research Council (NSERC).

A special thanks to Jinnah S. Ali-Clarke and Aaron Gill-Braun for providing invaluable support and insight for our research. Another special thanks to Erin MacLellan for her invaluable editing contributions to this paper.

References

[1] S. R., "Covid-19's impact felt by researchers," 2021.

[2] C. F. Laffan, J. E. Coleshill, B. Stanfield, M. Stanfield, A. Ferworn, "Using the ARAIG haptic suit to assist in navigating firefighters out of hazardous environments," 11th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON), 2020, doi:10.1109/iemcon51383.2020.9284922.

[3] C. F. Laffan, R. V. Kozin, J. E. Coleshill, A. Ferworn, B. Stanfield, M. Stanfield, "ARAIG And Minecraft: A COVID-19 Workaround," in 2021 IEEE Symposium on Computers and Communications (ISCC), 1–7, 2021, doi:10.1109/ISCC53001.2021.9631428.

[4] W. J. Ripple, C. Wolf, T. M. Newsome, P. Barnard, W. R. Moomaw, "World Scientists' Warning of a Climate Emergency," BioScience, **70**(1), 8–12, 2019, doi:10.1093/biosci/biz088.

[5] J. P. Tasker, "Elizabeth May says climate change, extreme events like Fort McMurray fire linked — CBC News," 2016.

[6] S. Larson, "Massive fire north of Prince Albert, Sask., is threatening farms and acreages — CBC News," 2021.

[7] W. Mora, Preventing firefighter disorientation: Enclosed structure tactics for the Fire Service, PennWell Corporation, Fire engineering Books & Videos, 2016.

[8] "Fabric," 2015.

[9] "Fabric Mixin Framework," 2015.

[10] M. Johnson, K. Hofmann, T. Hutton, D. Bignell, "The Malmo Platform for Artificial Intelligence Experimentation," in Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI'16, 4246–4247, AAAI Press, 2016, doi:10.5555/3061053.3061259.

[11] S. Adams, I. Arel, J. Bach, R. Coop, R. Furlan, B. Goertzel, J. S. Hall, A. Samsonovich, M. Scheutz, M. Schlesinger, S. C. Shapiro, J. Sowa, "Mapping the Landscape of Human-Level Artificial General Intelligence," AI Magazine, **33**(1), 25–42, 2012, doi:10.1609/aimag.v33i1.2322.

[12] D. Brewer, N. R. Sturtevant, "Benchmarks for Pathfinding in 3D Voxel Space," in SOCS, 2018.

[13] F. W. Fichtner, A. A. Diakité, S. Zlatanova, R. Voûte, "Semantic enrichment of octree structured point clouds for multi-story 3d pathfinding," Transactions in GIS, **22**(1), 233–248, 2018, doi:10.1111/tgis.12308.

[14] M. J. Atallah, M. Blanton, 22.4, CRC Press, 2010.

[15] S. J. Russell, P. Norvig, Artificial Intelligence: A modern approach, Pearson, 4th edition, 2022.

[16] "Protocol FAQ, URL: https://wiki.vg/Protocol FAQ." .

[17] S. Team, "Spigot, URL: https://www.spigotmc.org/." .

[18] "MinecraftForge Documentation, URL: https://mcforge.readthedocs.io/en/latest/." .

[19] Notch, "Minecraft 0.0.11A for public consumption," 2009 URL: https://web.archive.org/web/20150716115516/http://notch.tumblr.com/post/109000107/minecraft-0-0-11a-for-public-consumption.

[20] C. F. Laffan, R. Kozin, "Octree Path Finding Algorithm," 2021. URL: https://github.com/cassLaffan/Minecraft Pathfinding.

[21] "ARAIG As Real As It Gets, URL: https://iftech-technologies.com/downloadthe- sdk/." .

[22] B. Stanfield, M. Stanfield, "Download the ARAIG SDK, URL: https : / / www . firefacilities.com/fire-training-towers/tower-models/thecaptain/." .

[23] "Fire Department Training Building - Multiple Fire Fighter Trainees," 2020. URL: https : / / www . firefacilities.com/fire-training-towers/tower-models/thecaptain/.

[24] "Build new augmented reality experiences that seamlessly blend the digital and physical worlds, URL: https://developers.google.com/ar." .