A* Pathfinding Driven Emergency Evacuations of High Density Areas

Jake McKenna
Monmouth University, New Jersey, USA
s1055453@monmouth.edu

Abstract:-

Efficient emergency evacuations in high-density environments, such as university buildings and apartment complexes, are of critical importance to ensure the safety of occupants. Traditional evacuation plans can struggle to manage congestion and allocate optimal paths to all individuals, resulting in slower evacuation times and increased risk. This paper proposes a novel approach to large-scale, spontaneous evacuations in complex environments by utilizing an adapted A* pathfinding algorithm, which incorporates additional parameters to enhance its effectiveness.

The A* pathfinding algorithm is valued for its optimality and efficiency in search problems, employing a best-first search strategy to evaluate nodes based on a combination of movement cost and heuristic distance from the target. In this study, we extend the A* algorithm by incorporating traffic-related parameters to better manage congestion and optimize evacuation routes. We then evaluate the effectiveness of this modified algorithm by modeling and simulating evacuation scenarios in the Unity real-time rendering engine.

By addressing the challenges of congestion and path allocation in emergency evacuations, our research aims to contribute to the development of more efficient and adaptable evacuation strategies for high-density environments, ultimately enhancing the safety and well-being of individuals in crisis situations.

1. Background and Related Work

The gap we aim to address is to contribute a fully dynamic pathfinding system that can adapt to changing conditions and efficiently guide evacuations. Where previous literature has focused on pre-generated data via reinforcement learning, requiring extensive setup and being unable to adapt to unforeseen changes in the environment's layout, we instead propose a dynamic system that requires no pre-simulation. This paper presents a theoretical examination of this type of pathfinding and a technical breakdown of the function of this simulation.

In developing the simulation and implementing the A* algorithm, we followed a comprehensive tutorial series on YouTube by Sebastian Lague titled ‘A* Pathfinding Tutorial (Unity)' [1]. This series provided valuable guidance on setting up the grid and environment, as well as implementing the base-level A* pathfinding algorithm, which we then adapted and modified to meet the specific requirements of our evacuation simulation.

2. Methodology

2.1 Environment Setup:

The methodology adopted for this research begins with creating a 2D environment in the Unity real-time rendering engine. We set up a test environment out of simple geometric shapes and defined them as walls via the tagging system. We then generated a grid of nodes at a predefined size and resolution that overlapped the environment. Nodes store penalty data, the nodes ‘world position’ in the Unity scene, and its two dimensional position in the grid. During the first frame of runtime each node is evaluated via a raycast to check the tag of any overlapping object. We can then determine walkability if the overlapping tag indicates an obstacle. Next, static penalties for each node are calculated based on proximity to other unwalkable nodes, then are smoothed to provide a more realistic result.

2.2 A* Algorithm:

A* pathfinding uses a best-first search methodology to evaluate nearby nodes based on two parameters: the cost of moving to an adjacent node and a heuristic value based on the node's distance from the intended destination. These two costs are added together and evaluated to find the best path. We define the movement cost as the gCost, the heuristic as the hCost, and the combination of two as the fCost.

Presently we can define the A* algorithm as such:

- Each Node's Costs:
  - *g(n):* the cost of movement from the current node
  - *h(n):* the heuristic cost associated with reaching the target node (in our case Octile distance)
  - $f(n) = g(n) + h(n)$
- Pseudo Code:

```
openSet //Nodes to be evaluated
closedSet //Nodes that have been evaluated

add startNode to openSet

Loop
    currentNode = node in openSet with lowest fCost
    remove currentNode from openSet
    add currentNode to closedSet

    if currentNode is targetNode
        return

    for each neighborNode of currentNode
        if neighborNode is unreachable or in closedSet
            skip neighborNode

        get newCost to neighborNode //we recalulate the gCost to see if it is less than it was prevously

        if the newCost is < the neighborNode gCost or the neighborNode is not in openSet
            //If we have moved closer to the neighborNode or it new we recalulate its values and add it to the openSet
            recalulate the gCost and hCost of the neighborNode
            set the parent of the neighborNode to be the currentNode
            if the neighborNode is not in openSet
                add neighborNode to openSet
```

Figure 1

2.2 Extended A* Algorithm:

For use in optimizing evacuation traffic we must add in parameters to A*'s cost calculations. To achieve accurate pathfinding the traffic cost must be split into two categories with three underlying parameters. The first of the two categories is calculative traffic which we represent

with *c(n)*. Calculative traffic is what we use to evaluate future traffic congestion and draw efficiently allocated paths. To find the calculative traffic we must add data from two traffic sources. The first is the effect of nearby static objects, this we will call the static penalty. The next is the added cost from each drawn path which we will call the path penalty. These two parameters combine to give us a picture of what future traffic will look like at points along our grid, and can thereby inform our pathfinding algorithm. The next category of traffic is tangible traffic which we represent with *t(n)*. Likewise this is made up of two parameters: the static penalty, and the dynamic penalty. The dynamic penalty is the real world changing conditions of physical space of the evacuation. The primary contributor to the dynamic penalty is proximity to other units, though additional sources such as changing terrain conditions exist as well. The dynamic penalty is only used in simulating the this A* implementation. It is not used in the calculation of new paths. This disconnect between the real-world speed of units and the forecasted path allows for useful recalculation of a units current path as they farther stray from the 'optimal' route. This dynamic penalty can be thought of as the unforeseen conditions that may arise as an evacuation commences. These additional traffic parameters and categories we can modify the A* algorithm to provide useful path allocation to crowds traversing complex areas.

With these modifications we can now define the pathfinding algorithm as such:

- Each Node's Costs:
    - *Static Penalty:* the movement penalty conferred by the static environment
    - *Path Penalty:* the movement penalty added by paths drawn through this node
    - *Dynamic Penalty:* the movement penalty proximally added by nearby units
    - *c(n): Static Penalty + Path Penalty*
    - *t(n): Static Penalty + Dynamic Penalty*
    - *g(n):* the cost of movement from the current node
    - *h(n):* the heuristic cost associated with reaching the target node
    - *f(n) = g(n) + h(n) +c(n)*
- Pseudo Code:

```
openSet //Nodes to be evaluated
closedSet //Nodes that have been evaluated

add startNode to openSet

Loop
    currentNode = node in openSet with lowest fCost
    remove currentNode from openSet
    add currentNode to closedSet

    if currentNode is targetNode
        return

    for each neighborNode of currentNode
        if neighborNode is unreachable or in closedSet
            skip neighborNode

        get newCost to neighborNode + cCost //we recalulate the gCost, this time accounting for traffic

        if the newCost is < the neighborNode gCost or the neighborNode is not in openSet
            //If we have moved closer to the neighborNode or it new we recalulate its values and add it to the openSet
            recalulate the gCost and hCost of the neighborNode
            set the parent of the neighborNode to be the currentNode
            if the neighborNode is not in openSet
                add neighborNode to openSet
```

Figure 2: when assigning the paths to units the path penalty is added to each node as it is committed to the path, this is done after pathfinding is completed since the cCost is globally shared and should not be updated during calculation.

2.3 Path requests:

After the grid generation is complete units within the environment begin requesting paths from the pathfinding script via the path request manager. Requests are queued and processed one at a time to allow for the grid to update the Path Penalty value before calculating a new path.

The modified A* algorithm distributes paths over the available area while avoiding walls and confined spaces. This balance between the Path and Static penalties is essential to drawing reasonable paths. We used a process of trial and error to determine an adequate balance.

2.4 Evacuation:

During an evacuation, units follow their assigned paths while continuously checking the walkability of the nodes ahead. If a path becomes unwalkable during traversal, the unit recalculates its path using the same method as the original calculation and proceeds along the new route.

Units are slowed during the evacuation as a function of the tangible traffic ($t(n)$) present on their current node. Tangible traffic is increased by the static penalty conferred by nearby obstacles and the presence of nearby units.

3. Findings

Below we have included images from the tests run within the Unity engine. It is difficult to quantitatively analyze the results of the simulations at present since the penalty values are set somewhat arbitrarily. We however can qualitatively analyze these images and draw findings.

In figure 3 we have our base environment within the simulation. We chose the layout to highlight certain characteristics of the pathfinding algorithm. Note that the rooms on the right side of the environment have many tight corridors whereas the left is far more open.

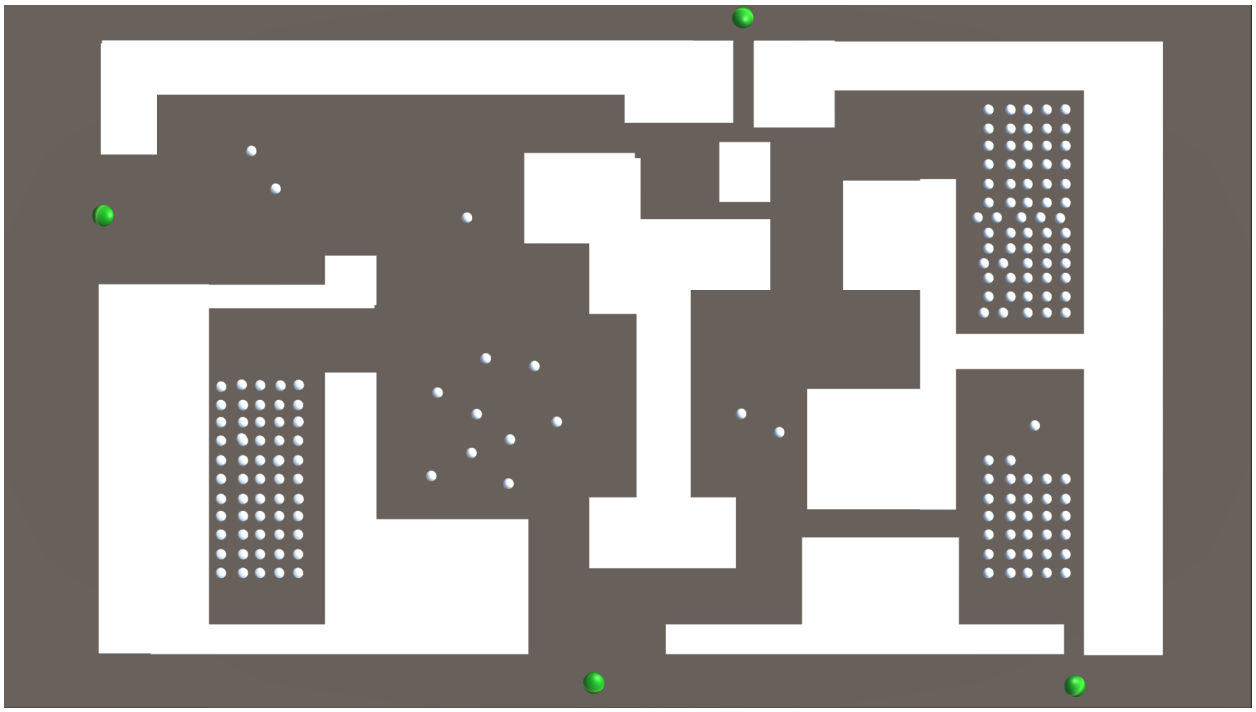

Figure 3: The green spheres represent targets, each white sphere is a pathfinding unit.

In figure 4 we can see the implementation of the grid system. Each node underneath an obstacle has been marked red to represent its status as unwalkable. Units will not pathfind over unwalkable nodes.

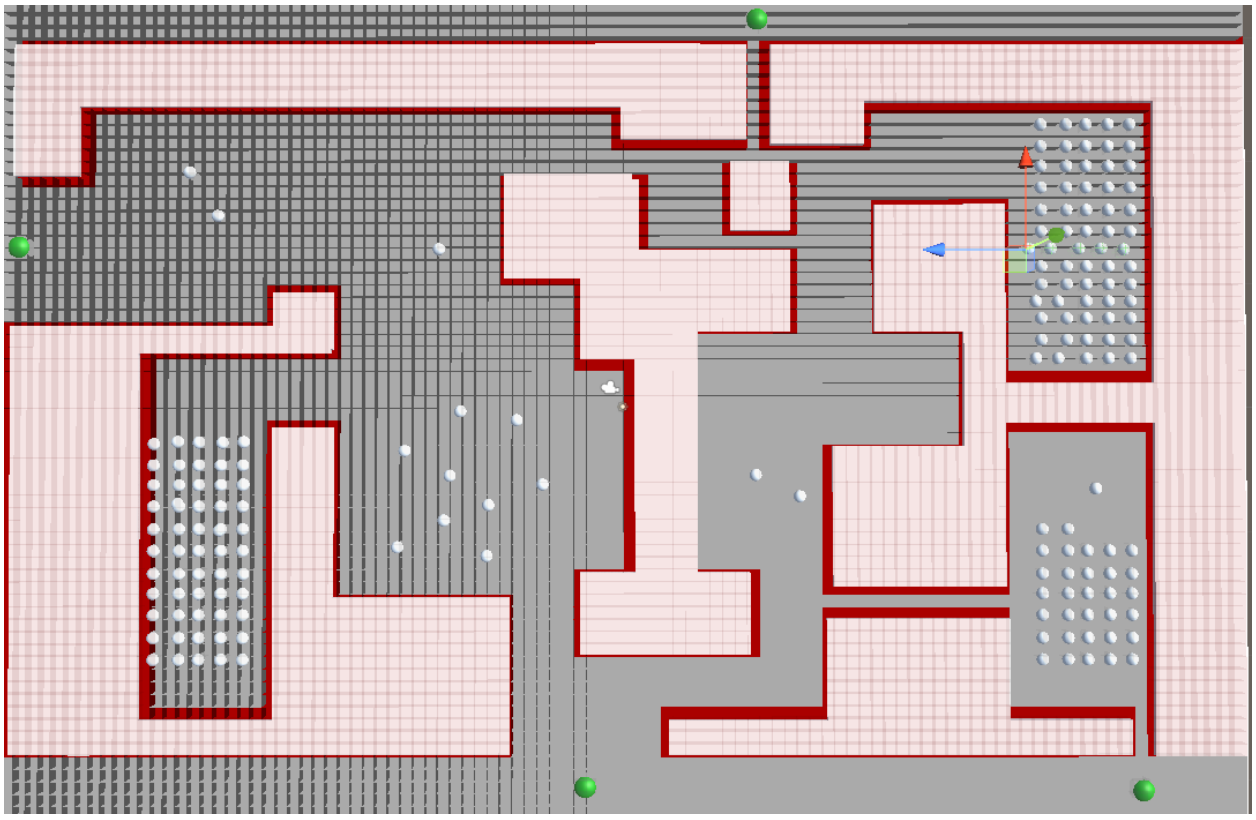

Figure 4.

In Figure 5 the static penalties have been visualized. The nodes with higher penalty values have been darkened to indicate their increased pathfinding cost. Note the scale of the visualization is relative to the maximum static penalty that was set before the grid was drawn, and does not represent its absolute value relative to all other pathfinding costs. The static penalty map has been blurred at this stage, resulting in the gradient of values seen.

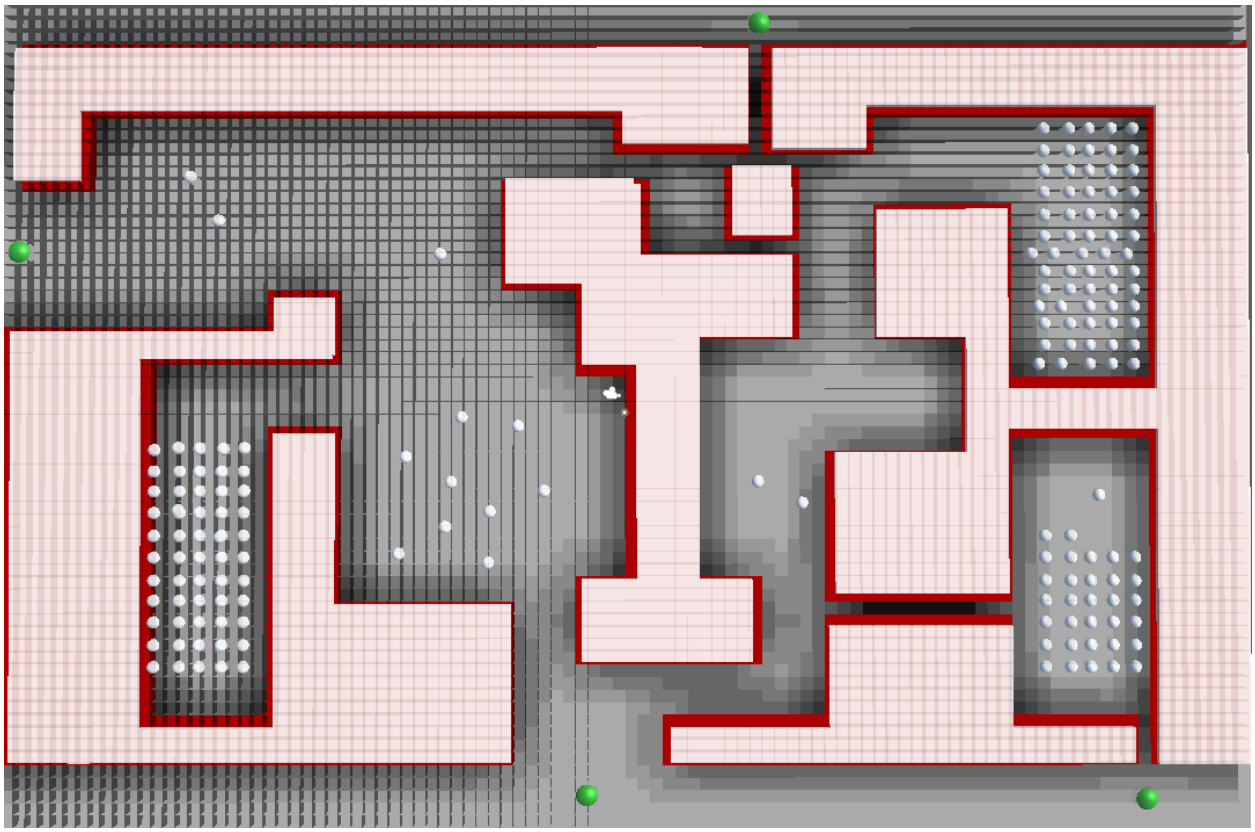

Figure 5

In Figure 6 we see the base A* algorithm without any of the additional penalties added in. Note the units often draw overlapping paths and stay close to obstacles. The units are adept at finding the closest target at this stage, but do not efficiently make use of the larger areas on the left side of the environment. .

Figure 6

In Figure 7 we add in the static penalty to the calculation of paths. At this stage the paths cluster together to avoid the additional penalties added by nearby obstacles. This phenomenon is particularly visible in the bottom left room where all the paths have constricted to one route through the doorway.

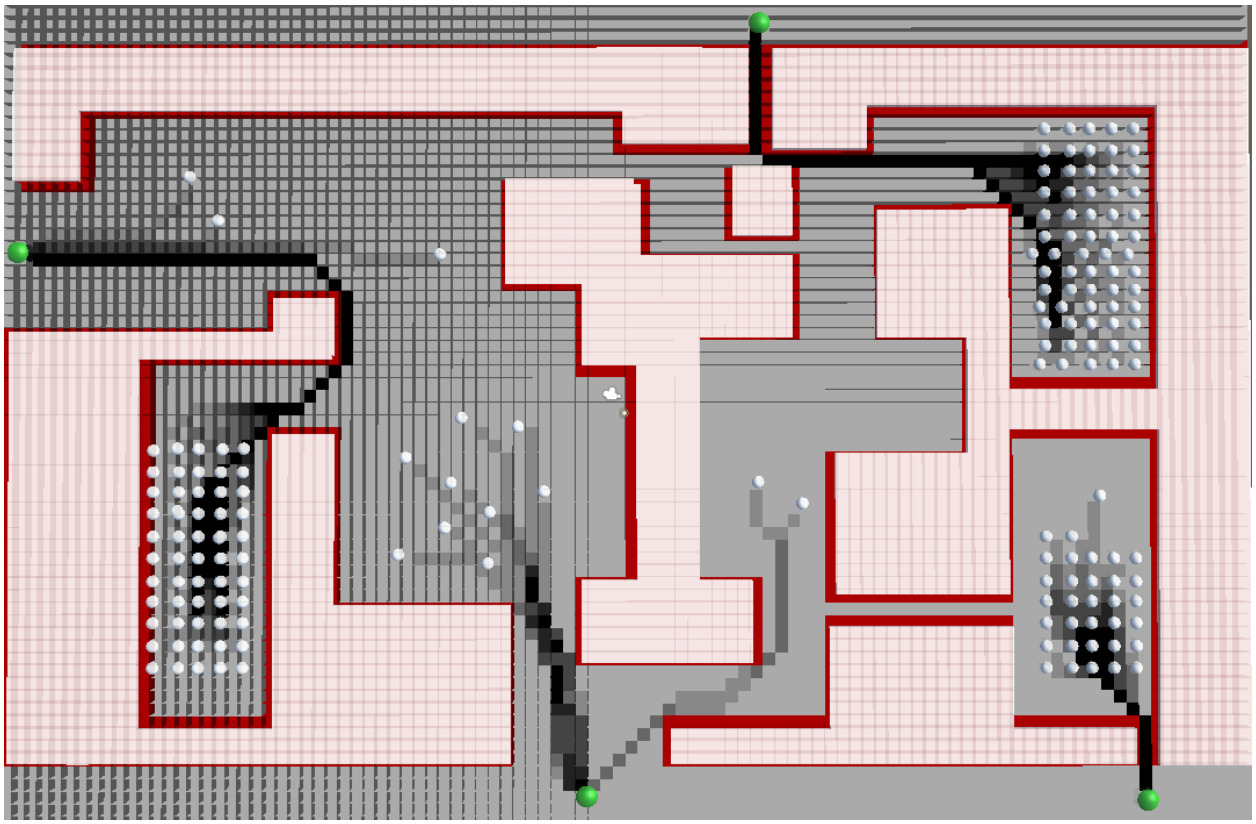

Figure 7

In Figure 8 only the path penalty is being added the base A* algorithm. Here we see the paths start to spread out in open areas to avoid overlapping. Generally this is desirable behavior to avoid traffic clustering, but as seen in the top right of the environment, units can unduly spread out when a large number of paths have been drawn through an area. This value is particularly difficult to balance. High values can lead to extremely inefficient paths as seen in Figure 9. The temporal nature of path traffic is evident in these figures. Realistically, units far from a target need not consider the paths of others that will navigate through an area and depart long before they arrive. Associating a time value with each instance of path traffic was considered, but dismissed due to the staggering amount of additional data that would need to be written to each node. Additionally, drawing paths from the closest units first would result in more accurate paths overall, but units farther from targets would begin with unnatural paths that would need to be recalculated once the traffic ahead had cleared, thus impeding their evacuation. This is a current limitation of our methodology.

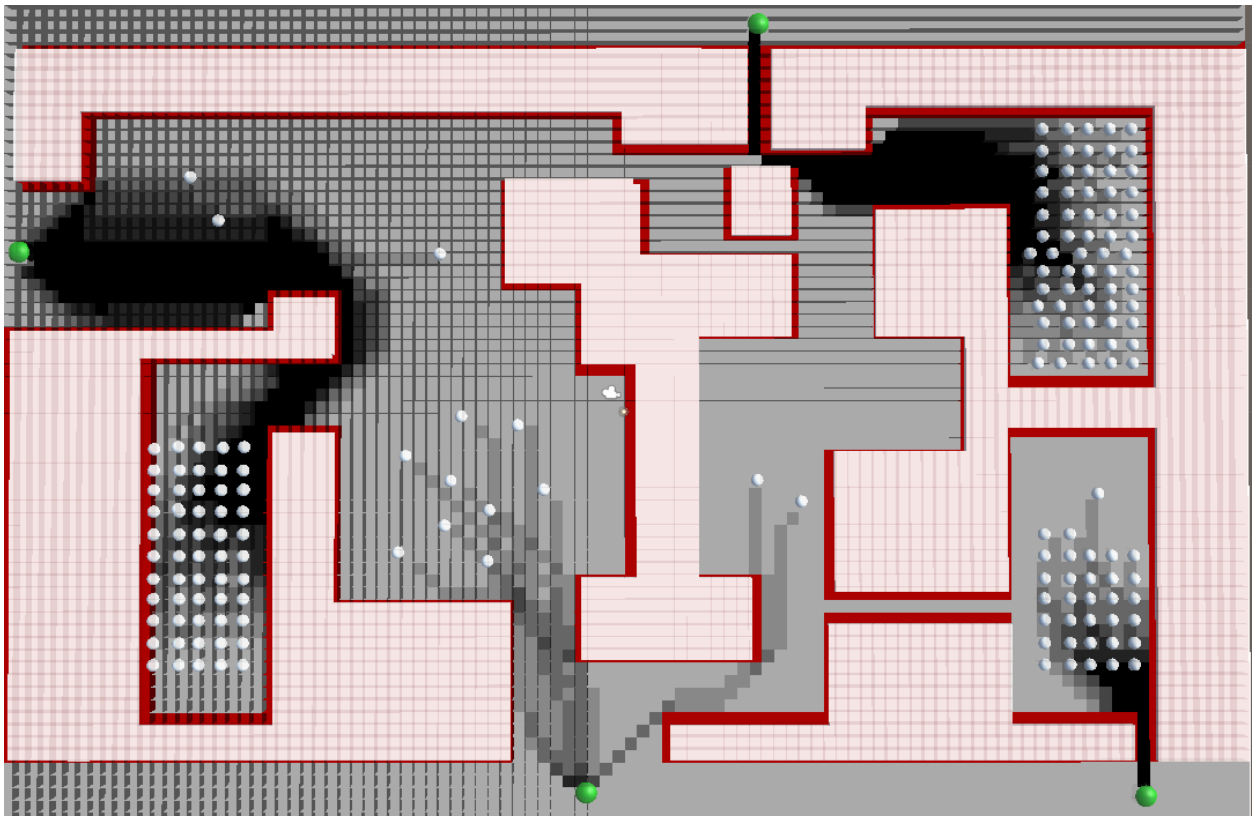

Figure 8

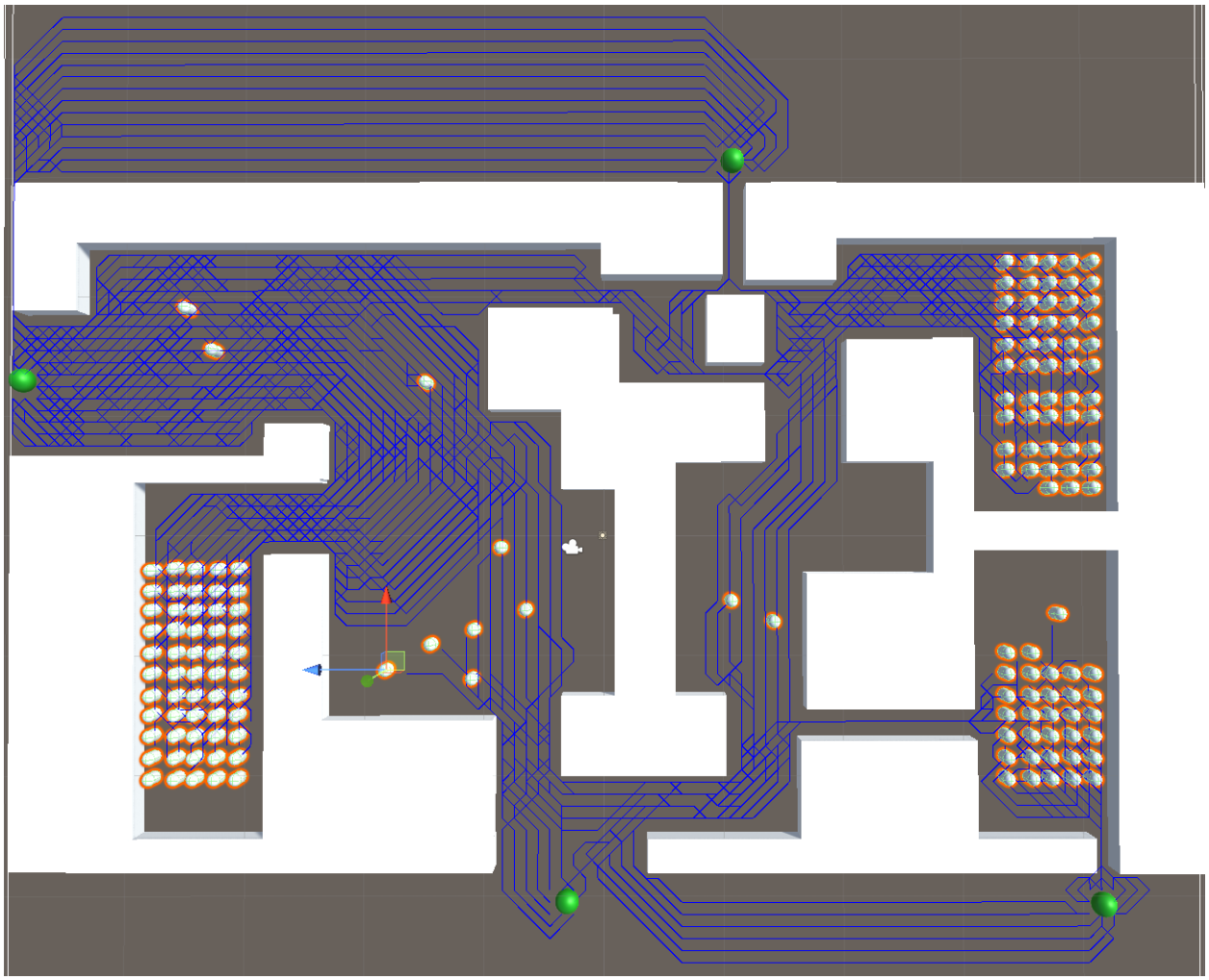

Figure 9 Note some of the more extreme paths that pass by the far left target and traverse all the way around to the top target.

In Figure 10 we see a balanced version of the algorithm fully implemented. Here we used a path penalty that was approximately five times greater than the static penalty, but it should be noted that the values will vary significantly for different environments, grid resolutions, and unit counts. Note that with this balance between the path and static penalties that some units from the bottom left room break off from their group to move toward the bottom center exit, which has relatively low traffic. Additionally the units from the top right room are not so over penalized that they navigate all the way down to either of the bottom exits. This balance of path and static penalties is the ideal behavior of the updated algorithm.

Figure 10 Note the static penalties are not visualized here.

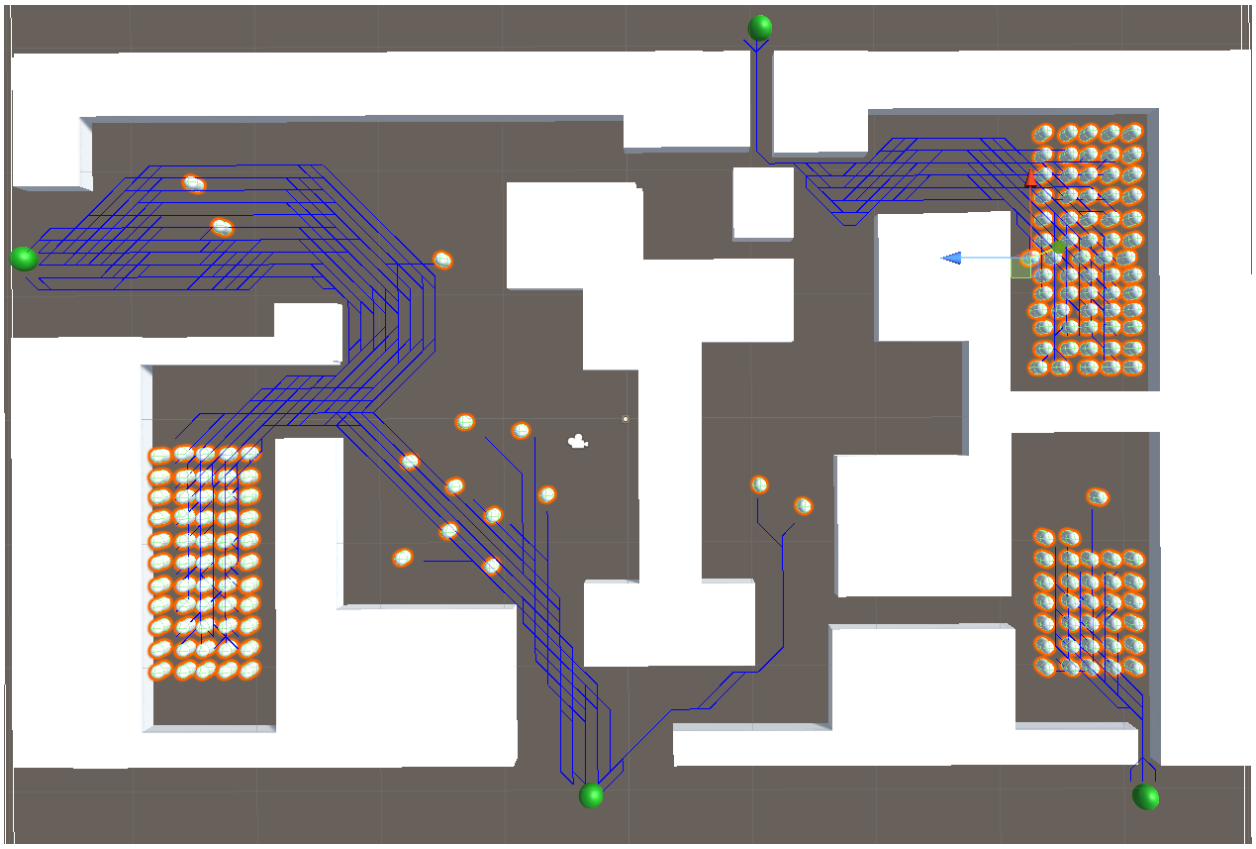

Figure 11. Note, this is the same simulation as figure 10, but with the paths visualized for clarity.

3. 1 Discussion and Future Work:

There are a number of ways to improve the current simulation such as randomization of the environment, creating a 3D navigable area for multiple floors, and adding randomization to unit speed. Though the most limiting factor found was the values associated with each traffic penalty. It was challenging to find a balance between the path and static penalties that yielded reasonable paths. Addressing this issue by reworking how the penalty values are allocated would greatly improve the accuracy of each simulation, and possibly allow for the collection of relevant data. Additionally, adding a time based dimension to path traffic would greatly increase the accuracy of the initial paths generated.

4. Conclusion

Emergency evacuations of high-density areas present a plethora of opportunities for optimization via the use of pathfinding algorithms. Implementation of these systems can help model evacuations effectively, predict and avoid traffic, and increase the overall efficiency of evacuation routes. This modified A* pathfinding algorithm offers a promising start to the modeling and optimization of emergency evacuations, addressing a gap in the literature for a fully dynamic pathfinding system that can adapt to changing conditions and efficiently guide evacuations.

Citations:

[1] Sebastian Lague. (2014). A* Pathfinding Tutorial (Unity). Retrieved from [https://youtube.com/playlist?list=PLFt_AvWsXl0cq5Umv3pMC9SPnKjfp9eGW]