Project Everglades
Game Development for Reinforcement Learning

Jayden Bendezu, Ethan Irwin, Jerold Lodge, Zachary Neidig, and Patrick Sherbondy

Group 21

Lockheed Martin Sponsors:

Rebecca Broadway and Tanner Lindbloom

Table of Contents

Executive Summary 1

Project Description 2

- Motivations 2
- Goals and Objectives 3
- Group Member Tasks 4
- Function 5
- Criteria and Constraints 5
- Broader Impacts 5

Legal, Ethical, and Privacy Issues 6

- Legal Issues 6
- Ethical Issues 6
- Privacy Issues 6

Specifications and Requirements 7

- Overview 7
- Gameboard Enhancement 7
- Drone Loadout Customization 8
- Increased Playability 9
- Improved Targeting Behavior 10
- Electronic Communications 10

Explicit Design Summary 11

- Drone Loadouts 11
- Three-Dimensional Game Board 11
- Improve User Experience 12
- Improved Targeting Capabilities 13
- Electronic Spectrum Communications 13

Block Diagrams 14

- Game Process Diagram 14
- Project Goals 16

3D Game Board 17

- Approach 17

Breadth-first search 18
Establishing Connections Between Nodes 27
3D Bell Curve Feature 29
3D Gameboard Renderer 31
Overall Requirements 31
Design Approach 31
Implementation 32
JSON Input 33
Important Implementation Details 35
Result 36
Using the Renderer 39
3D Wind stochasticity 40
Drone Loadout Customization 45
Storage and Loading 45
JSON Format 45
Original Drone Loadout 48
Original Assertions 49
Refactoring to Accept JSON 54
Input JSON 55
Conversion Method 55
Loadout Rule Handling 57
Prototype Interfaces 58
Developer GUI 59
Difficulties with Loadouts 68
Recon Drones 69
Improved Playability 72
Unit Creator 72
Making the Unit Creator 72
Unit Attributes 76
Interfaces 79
Everglades Desktop Application 89

Desktop Application Dependencies 91
Unit Class Refactor 94
Justification 94
Proposal 97
Implementation 97
Detection Phase 101
Construction Phase 102
Destruction Phase 103
Telemetry Data 105
Regression Testing 106
Approach 107
Unit Health Representation 109
Drone Targeting 111
Overall Requirements 111
System Implementation 111
Gameplay Implications 113
System Implications 114
Previous Implementation 115
Targeting Statistics 119
Approach 120
Statistics 121
Interpretation 124
Electronic Communication 125
Overview 125
Battling 126
Flee 130
Limitations 132
Range 132
Game Implementation 136
EMP 136
Background 136

Game Implementation 137
Apex Legends 137
Deus Ex 138
Watchdogs 138
Mario Kart 139
EMP Incorporation 139
Crucial Changes to Everglades Server 141
OpenAI Gym 141
File Paths 142
Build, Prototype, Test, and Evaluation Plan 145
Unit Testing 145
Integration Testing 145
System Testing 145
GitHub Development 146
Unit Tests 148
Unit Integration 149
Budget and Financing 150
Overview 150
Python Server Costs 150
Unreal Engine Costs 150
Version Control Costs 151
Summary of Costs 151
Gantt Chart 151
Project Milestones 154
Senior Design I Milestones 154
Senior Design II Milestones 156
Project Summary and Conclusion 158
Project Summary 158
For Future Groups 160
Observation Space Refactor 160
New Unit Attributes 160

New Targeting Functions 161
Extend Electronic Communications and Warfare 161
Extend the Unreal Visualizer 161
References 162

Executive Summary

Project Everglades is a game made to research and teach AI training using a turn-based video game where each team controls multiple groups of drone swarms. The game allows AI to train strategy toward achieving victory while graphically displaying the game for view interpretation and enjoyment. A python-based server runs the game where the AI's are trained, while the Unreal Engine is used to display a graphical representation. The game is a tool for training to be used by Lockheed Martin and Universities, created to be fun and enjoyable to allow for a more modern medium of teaching through gaming.

There are two primary objectives for this iteration of Project Everglades. The first is to expand the game board, allowing for nodes on multiple elevations and connect them realistically. This means creating nodes on the at different elevations for a 3D board. The other objective is to allow players and AI to create custom drone squadrons comprised of any drone unit type players desire, using a balanced cost system.

Though not required, Lockheed Martin would like to see increased playability, improved targeting capabilities in drones, and an implementation of electronic communications between said drones. In its current state, the foundations for the game have been set without much regard for playability and making the game fun. As such, a larger focus on bettering user experience is desired to make the game more user-friendly and engaging.

In terms of the current state of drones, they do not exhibit intelligent behavior when targeting enemies, so coordination in these attacks is desired. Communications are intended to model real-life electronic communications between drones, with the possibility for cyberattacks to jam enemy communications. Drones will be given fresher AI to allow them some self-agency in the case that such a cyberattack occurs.

We will expand the game board by into three dimensions. We will allow for custom drone squadrons by gutting the hard-coded existing squadron code, supplement it with modular code allowing for various numbers and type of units in greater squad conditions with customizable attributes and create GUIs within the Python server's files to allow developers and users to more easily customize the settings to fit their needs.

Project Description

Motivations

Jayden Bendezu

My first interaction with programming was developing video games in high school, primarily using the Unity Engine in C#. I decided to major in Computer Science at UCF so that I could delve deeper into computer science and learn things asides from developing games. I soon became split between favoring an analytical career in algorithms or a creative career in game design or software engineering. I chose this project as my number one top pick because I wanted to utilize my skills in game design for something of greater purpose and really push myself to work on a large scale project that will incorporate that analytical thinking I crave.

Jerold Lodge

I was introduced to programming at a young age and fell in love with the opportunities coding has to offer. I'm pursing my degree in Computer Science at UCF and I have gained a lot of experience and knowledge along the way. I am interested in Software Development and Artificial Intelligence, so this project gives me the opportunity to learn about game development in more of a professional setting. I'm looking forward to going above and beyond with this team and creating a great product.

Zachary Neidig

I have always had a passion for programming and software design in video games. I am very self-driven and enjoy learning new languages and programs, particularly how they allow familiar concepts and programs to be structured differently and take advantage of differences in the environments. I hope to take a lot away from this project as I use new programs and collaborate with my teammates by pooling our knowledge. My most notable previous project is a mobile game (made in Unity) whose leaderboard and forums were accessible via a webpage.

Patrick Sherbondy

Ever since high school, I knew I wanted to do something related to computers, and within my first semester at UCF I realized that software programming was the path I wanted to take. Programming has quickly become something I enjoy doing – alongside playing video games – so game development was a natural evolution in my interests. When the opportunity to make a game for Lockheed Martin presented itself, I realized that this was a project I would be passionate about – solve novel problems while developing a video game.

Ethan Irwin

Making video games has become a strong passion of mine and an outlet for my creativity. When looking at this project I saw it as an opportunity to apply my passion toward gaming to an interesting project where I could learn more about AI, the Unreal Engine, and high-level concepts from experts in multiple fields. The creativity allowed in this project appealed to me greatly, so I pushed to get this project so that I could bring my creativity to it and make something amazing, while also gaining more skills and knowledge that will benefit me in my computer science career moving forward.

Goals and Objectives

There are two primary objectives for this iteration of Project Everglades. The first is to expand the game board, allowing for nodes on multiple elevations and connect them realistically. This means creating nodes on the at different elevations for a 3D board. The other objective is to allow players and AI to create custom drone squadrons comprised of any drone unit type players desire, using a balanced cost system.

Though not required, Lockheed Martin would like to see increased playability, improved targeting capabilities in drones, and an implementation of electronic communications between said drones. In its current state, the foundations for the game have been set without much regard for playability and making the game fun. As such, a larger focus on bettering user experience is desired to make the game more user-friendly and engaging.

In terms of the current state of drones, they do not exhibit intelligent behavior when targeting enemies, so coordination in these attacks is desired. Communications are intended to model real-life electronic communications between drones, with the possibility for cyberattacks to jam enemy communications. Drones will be given fresher AI to allow them some self-agency in the case that such a cyberattack occurs.

Group Member Tasks

To clearly delineate the roles of all group members, the following table briefly states what section each group member was assigned to.

Group Member Tasks

Member	Tasks
Jayden Bendezu	Drone Squadron Customization, Drone Targeting Improvement, Electronic Communications
Jerold Lodge	3D Gameboard, User Interfaces
Zachary Neidig	Drone Squadron Customization, User Interfaces, Electronic Communications
Patrick Sherbondy	3D Gameboard Renderer, Refactor, Drone Targeting Improvement
Ethan Irwin	Loadout Customization, Unit Customization, Electronic Communications

Function

Project Everglades is a game made to research and teach AI training using a turn-based video game where each team controls multiple groups of drone swarms. The game allows AI to train strategy toward achieving victory while graphically displaying the game for view interpretation and enjoyment. A python-based server runs the game where the AIs are trained, while the Unreal Engine is used to display a graphical representation. The game is a tool for training to be used by Lockheed Martin and Universities, created to be fun and enjoyable to allow for a more modern medium of teaching through gaming.

Criteria and Constraints

We primarily only have one constraint, as we are using the existing code and design of previous project groups. Using the existing python server which runs the game, and the Unreal Engine code that visualizes it for a view, we must add onto the project without changing functionality and code that is not desired to be altered. These systems are set, making the choice of programming language, styles, and setups constrained to follow that of the previous groups who have worked on it.

Broader Impacts

Project Everglades is an evolving game meant to keep pushing drone behavior research forward. As such, the findings of both this version and upcoming versions may make contributions to how drone behavior works for uses such as military applications. Improved communication between drones, smarter AI, and better understanding of AI in relation to drones are all possible impact areas of this project. From a training perspective, this game is a tool for teaching how AI works, and as such will impact many people in the future in terms of education toward AI. With both combined, it is likely this project will impact how drone AI is used in the future

Legal, Ethical, and Privacy Issues

Legal Issues

There should be little to no legal issues that present themselves from working on Project Everglades, unless Lockheed Martin intends on releasing the software generated to the public. All the code that was added to our finished iteration of the project is generated by ourselves, without any intent to publish the finished product.

If Lockheed Martin decides to go public with the project in a future iteration, then legal issues could occur depending on the copyright status of any algorithms we used, such as stochastic wind. It is possible that we use a patented algorithm that causes the game to get flagged. Additionally, since we are using Unreal Engine, royalty fees would be incurred on the product should it be monetized.

So long as Lockheed Martin keeps the game for internal use only, there should be no legal issues that arise with the development or post-development status of Project Everglades.

Ethical Issues

Naturally, working on any project for a defense contractor brings its share of ethical concerns, no matter what the project may be. Lockheed Martin is not making a game for the sake of making a fun game for people to play, but rather they are creating a game that can be used to train artificial intelligence as an exercise in machine learning.

Working on this project can conceivably open the door to more advanced artificial intelligence capable of performing on an active battlefield against enemy combatants. The application of such knowledge could be utilized at any point, so one would have to ask themselves if they were comfortable lending their skills to a project that could eventually be utilized for such purposes.

Privacy Issues

In the current iteration of the project, there are no foreseeable privacy issues. As a self-contained single player game, we are not handling any form of user data – sensitive or otherwise – and thus we are not concerned about privacy. Our game is incapable of interacting with the Internet and it does not have the ability to store user data locally in any form. Thus, there is no need to be concerned for privacy.

Specifications and Requirements

Overview

The following is an organized list of all the requirements for this iteration of Project Everglades. There are seven total sections: gameboard enhancement, drone loadout customization, increased playability, improved targeting behavior, electronic communications, unit class refactor, and Unreal visualization. Of these, only the last four are not absolute requirements for this project to be a success, though their inclusion would be greatly appreciated by the sponsor. The last two requirements were added by us as a way to improve the overall state of the project and make certain aspects of development easier.

Each requirement section is further broken up into requirements and stretch goals. Requirements must be met to satisfy the overall requirement, while stretch goals are less important to the success of the requirement but will markedly improve the quality of the requirement overall.

Gameboard Enhancement

1. Requirements
   - 1.0. Overhaul the gameboard generation system to accommodate nodes in three dimensions.
   - 1.1. Implement a way to connect nodes on different levels in a natural way.
   - 1.2. Represent the new grid implementation in a 3D manner on the gameboard.
   - 1.3. Ensure that special resource and team start nodes still generate as normal.
2. Stretch Goals
   - 2.0. Re-implement stochastic wind to the new 3D gameboard.
   - 2.1. Create a development tool that allows for easy viewing of generated 3D gameboards.
   - 2.2. Inside of the development tool, distinguish special nodes (i.e. resource nodes) from their regular counterparts.

Drone Loadout Customization

1. Requirements
   1.1. Implement a point system that gives players a base amount of virtual currency to spend on drone loadouts.
   1.2. Assign a balanced cost value to each drone type.
   1.3. Allow players to purchase drones and add them to their squadron loadout before the start of a game.
   1.4. Ensure drone loadouts are unique between players, such that one player's loadout does not end up being the other player's loadout as well.
   1.5. Implement the ability for AI players to create their own loadouts.

2. Stretch Goals
   2.1. Create a system that supports drone customization with the attributes of health, damage, armor, control speed, and cost.
   2.2. Add additional attributes that units could have.
   2.3. Create a user interface that allows players to create their own drones and be charged accordingly.
   2.4. Create a system that allows AI players to create their own drones and be charged accordingly.

Increased Playability

1. Requirements
   1.1. Create a Desktop UI to read in inputs from the user, view both player loadouts, generate a game setup JSON file, and run Project Everglades for use by developers and non-developers alike.
   1.2. Connect the Desktop UI to UI that allows the user to create custom groups loadouts, units, and unit attributes.
   1.3. Create a UI for setting up custom group loadouts.
   1.4. Create a UI for setting up custom units with various attributes.
      - Add mechanics to create new units using a system of attributes.
      - Separate units by default, presets, and custom for balance.
   1.5. Add mechanics to define units using a system of attributes.
   1.6. Separate units by default, presets, and custom for balance
2. Stretch Goals
   2.1. Implement the drone unit creator outlined in the Drone Loadout Customization section.
      - Add in new unit attributes to expand possible unit combinations
   2.2. Refactor the Unreal portion of the game to accept telemetry data from the newest iteration of the game and display 3D gameboards.
   2.3. Create group icons in the Unreal portion to easily identify where various groups are on the gameboard.

Improved Targeting Behavior

1. Requirements
   1.0. Implement a system where drones can prioritize their targeting using pre-determined targeting functions.
   1.1. Create a targeting system foundation that allows for different types of targeting systems.
   1.2. Allow players to choose what targeting system is used and apply it to all their drones across all groups.
   1.3. Create a targeting behavior that allows for completely random targeting.
   1.4. Create a targeting behavior that has a drone prioritize attacking enemy drones with the lowest health.
   1.5. Create a targeting behavior that has a drone prioritize attacking enemy drones with the highest health.
   1.6. Run all targeting systems through a callback function, so that all targeting systems can be handled as functions that are called based on the selection.
2. Stretch Goals
   2.0. Create a targeting behavior that has a drone prioritize attacking enemy drones who have the highest health and deal the most damage.
   2.1. Allow players and AI to create different targeting systems for their customized drones. This would be contingent on getting a working drone customization system working.

Electronic Communications

1. Requirements
   1.1. Set up the unit creator to allow for future attributes that can expand electronic communications.
   1.2. Fully integrate Recon drones to work with the new systems.
   1.3. Create a Jammer unit that slows nearby enemies' movement to resemble electronic warfare.

Explicit Design Summary

Drone Loadouts

To implement this feature, we designed a User Interface. Players can fully customize their team loadout by selecting which unit they would like to have and how many they want. The purpose of this is to add more complexity to the game which will affect how the AI will adapt and learn.

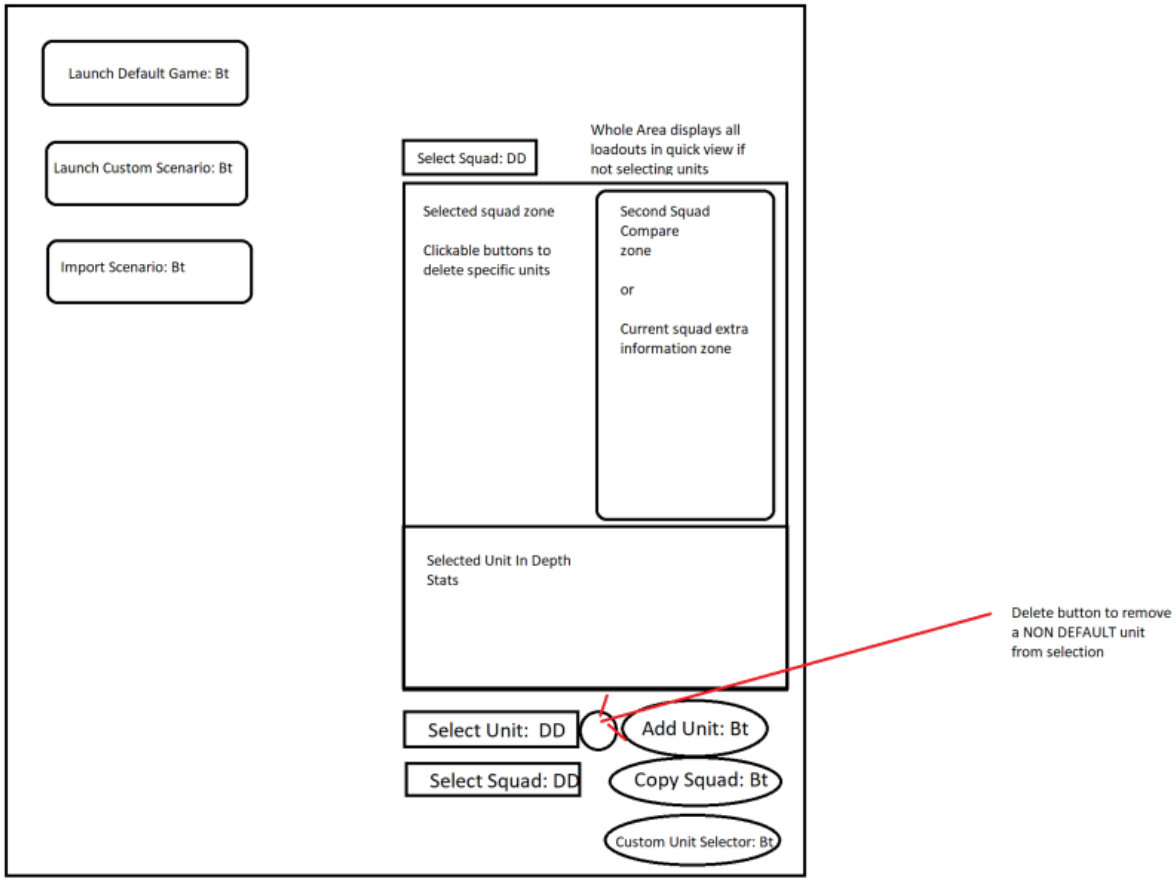


Figure 1: Loadout Prototype UI

Three-Dimensional Game Board

The previous senior design team implemented an algorithm to randomly generate a fair gameboard base on the number of nodes you want. Our plan is to build on top of this feature to generate a 3-Dimensional board. This feature can be used with the current map as drones fighting in the air. Also, there are plans to create an unreal map in space which will take full advantage of the 3-Dimensional game board.

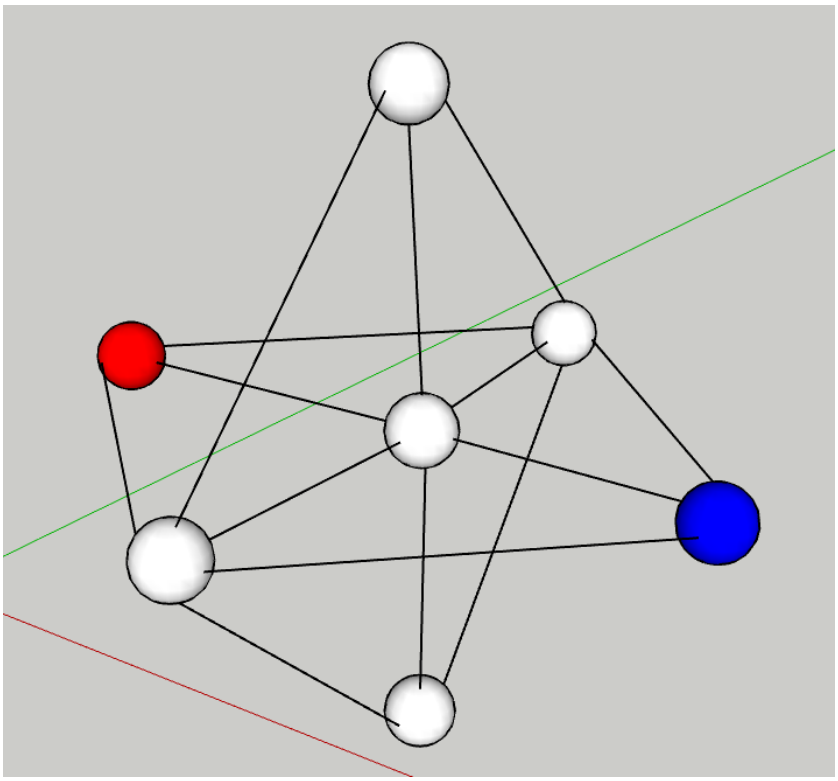


Figure 2: 3D Gameboard Sketch

Improve User Experience

To create a more streamlined experience and increase the engagement of the user, the decision was made to create several UIs contained in one desktop application such that the user will be able to launch games easier. This application would encompass creating loadouts, creating custom units, selecting various settings such as 3D or 2D maps, and launching the game with the appropriate AI agents. In additional to these UI additions, a unit creation was decided to be needed to increase the enjoyment and engagement of both recreational and research users. This creator would allow for new units to be created that are compatible with the game.

Improved Targeting Capabilities

In the previous iteration of Project Everglades, drones exhibited a very simplistic randomized targeting behavior. In this iteration of the project, one of our “nice-to-have” goals was to implement sophisticated targeting systems that operate on the attributes of the enemy units they face. This can be something as simple as a targeting system that prioritize attacks on units with lowest health first, or something as complex as targeting by specific unit types in a specific order.

As an added bonus, we decided to create a way for the artificial intelligence teams to create their own custom targeting functions. While we have created a small variety of preset targeting functions accessible via a callback function, the AI teams will be able to design their own based on any unit attribute they desire.

Electronic Spectrum Communications

This was our only “homerun” goal for the project, which initially had loftier goals. At the start of the first semester, the idea would be to bring in an expert to discuss how electronic communications could be realistically modelled. It was deemed by the sponsors to restrict the loftier goals down to what they considered more manageable and realistic for the scope of the project.

For this reason, we spoke with our sponsors to agree upon a new set of requirements. In the end, we agreed to create drones which would be able to perform electronic warfare in some capacity, with the end result being effects to the movement of enemy drones. If enemy drones are affected by electronic warfare, their speed will be decreased, making it take longer for them to navigate between nodes. In using the unit creator and attributes to achieve this goal, the foundation has been set up for more implementations of electronic warfare through attribute based abilities.

Block Diagrams

Game Process Diagram

This is the overall game process as it was given to us. Since we are continuing an existing project, all the components started in states of varying completeness. We chose not to uproot any of the functionality of the existing project and decided to continue its development in Python. As a result, this diagram remained unchanging through development.

Data is executed on the Python server in its entirety, and outputs telemetry data to be used in the game portion of the project. Unreal renders the telemetry data to graphically represent the decisions made by the AI. Additionally, if there is a player instead of 2 AI, Unreal presents the player a UI, accepts input from them, and visualizes their decisions.

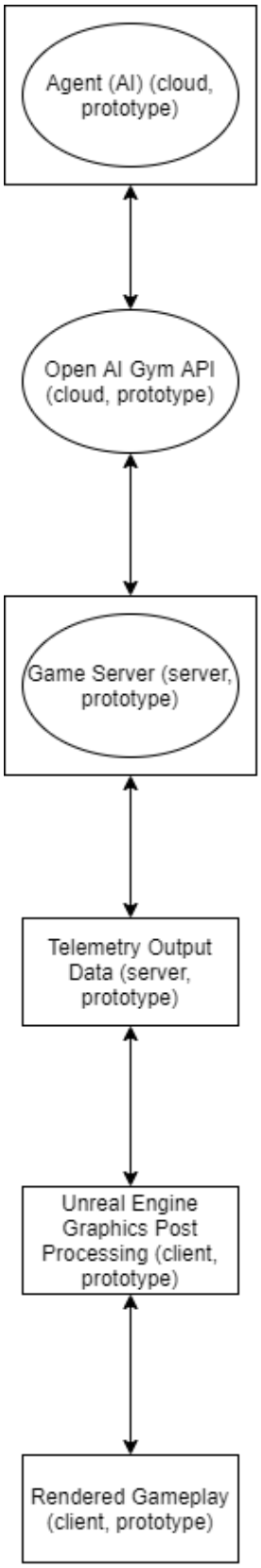


Figure 3: Game Process Diagram

Project Goals

These are the overall project goals, with early member assignments to each requirement.

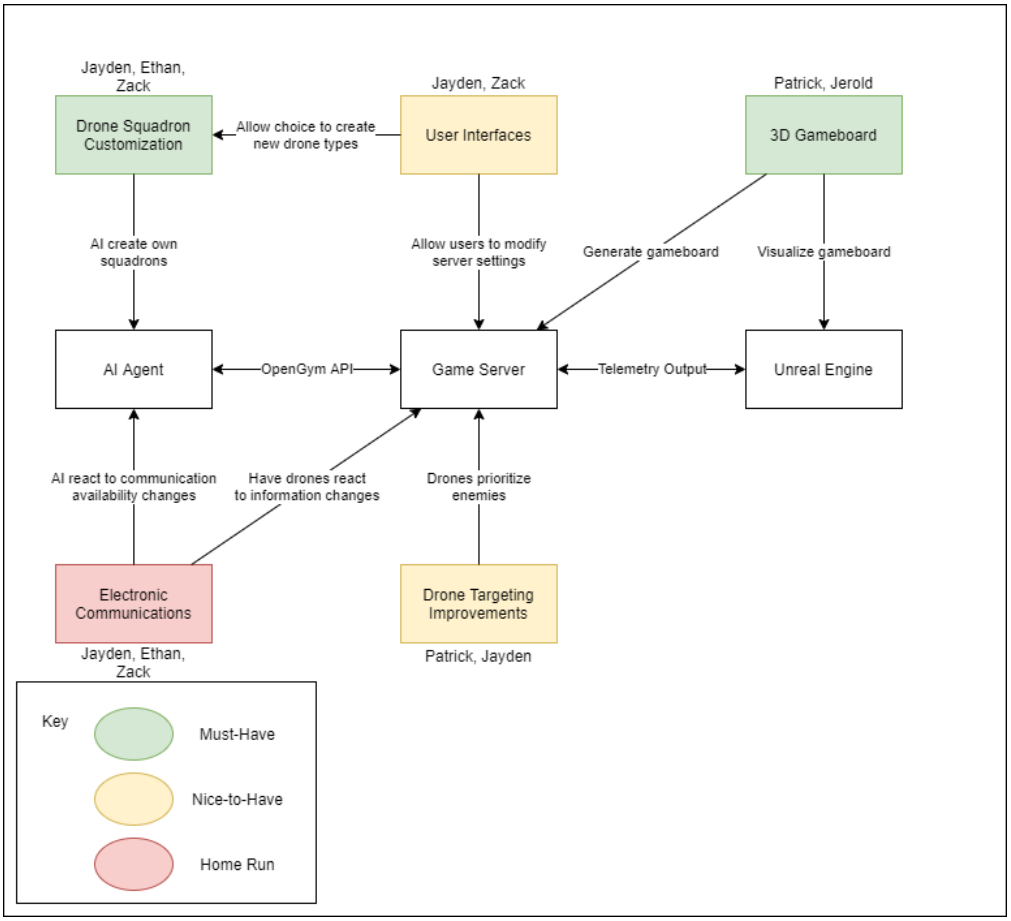


Figure 4: Everglades Block Diagram

3D Game Board

Approach

Our initial approach in implementing a 3-dimensional gameboard was to use last year's 2D procedurally generated map and stack the maps on top of each other. This approach is simple because the foundation is already built so the only implementation needed is to connect the nodes between each layer. However, this method does not offer much variability in different maps because the two layers would be identical.

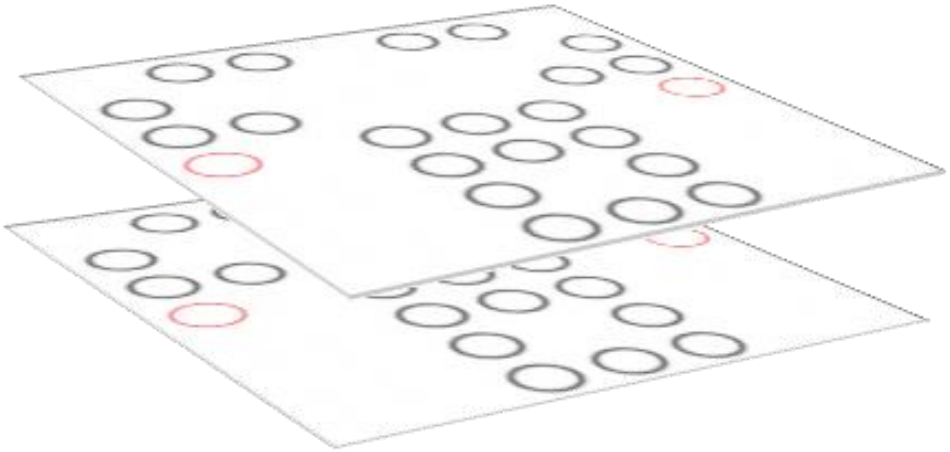


Figure 5: Preliminary Algorithm Design

Our alternative approach was to use the same algorithm for generating a 2D board and modifying it to work in a 3-dimensional space. The algorithm uses breadth-first search to build the map.

Breadth-first search

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root and explores all the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

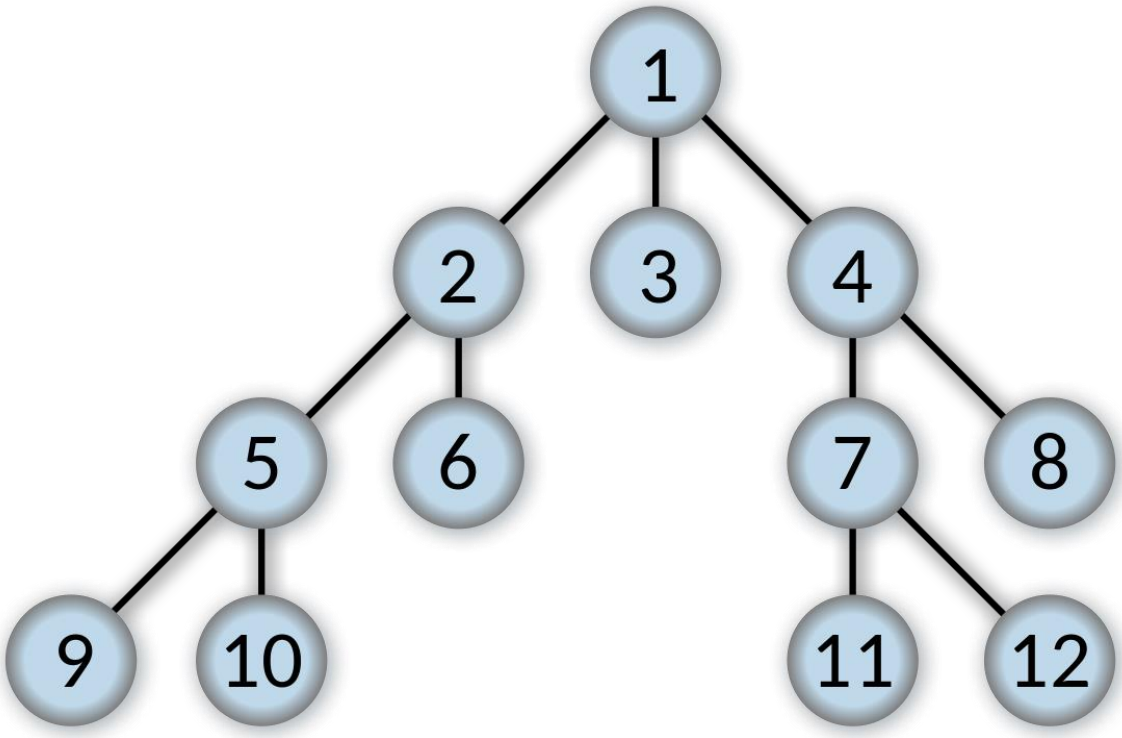


Figure 6: Breadth-First Search Image [1]

There are two main database structures used for this algorithm, a 3D array, and a queue. The 3D array is used to represent the 3-dimensial space of the game board, and the queue is used to keep track of which direction the breadth-first search travels to. The 3D array can be of varying length, but each 2D plane must have the same dimensions. All the values of the array are initialized to 0.

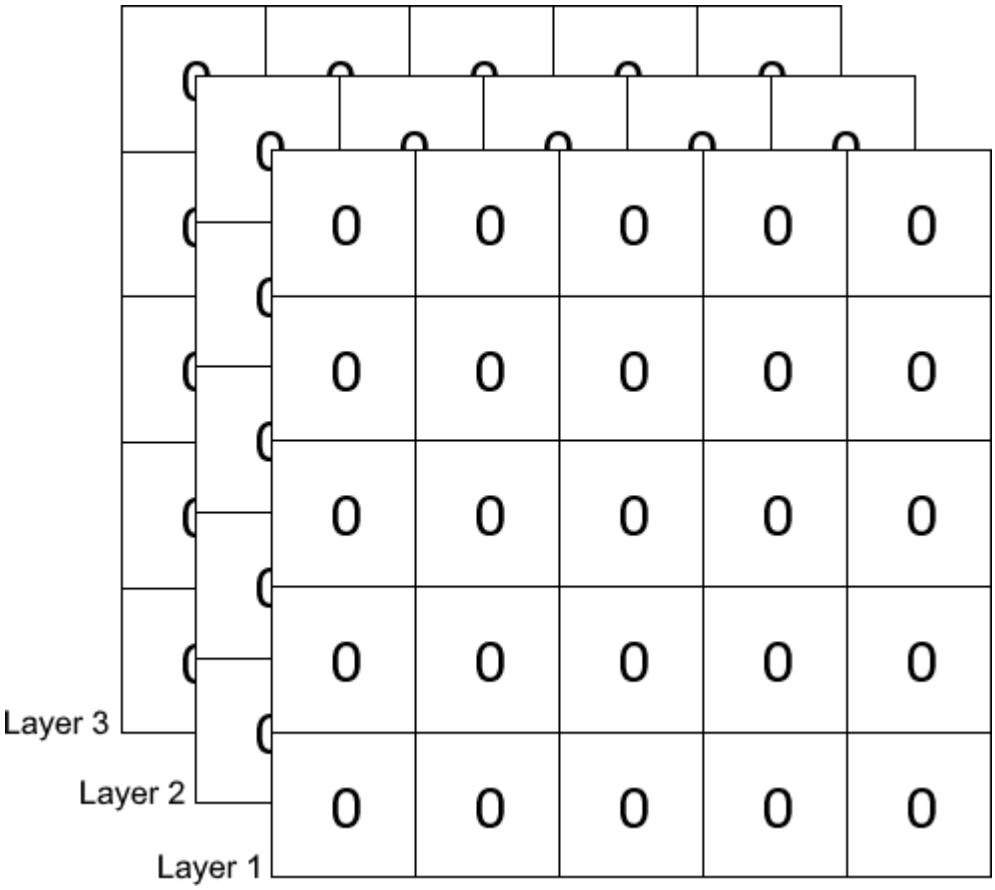


Figure 7: Initial Game Board 3D Array

Once the data structures have been initialized, the algorithm finds the center coordinate of the 2D plane in layer 1 and changes it to a 1. With the format (z, y, x), where z is the layer, y is the row, and x is the column. (0, 0, 0) is at the top left of the plane in layer 1 and (0, 4, 4) is at the bottom right, so the point (0, 2, 2) is added to the queue. This point represents the base of either the red or blue team.

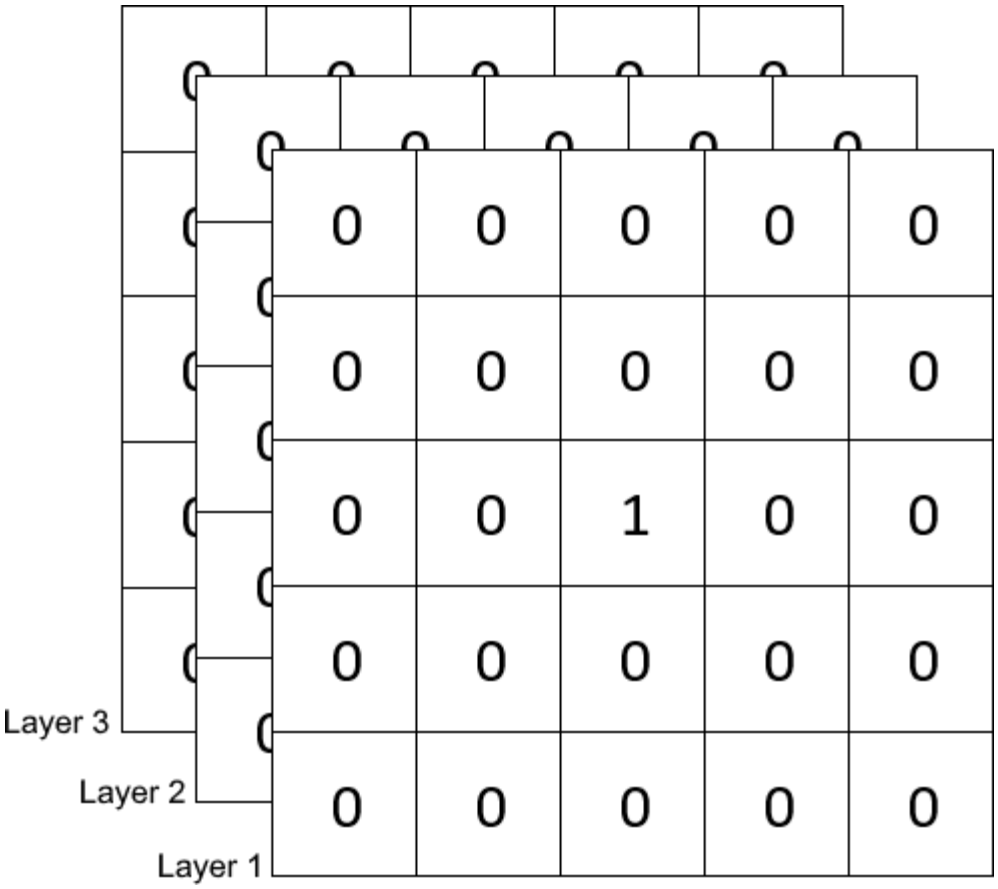


Figure 8: Initial Node Added

After the starting point as been placed into the queue, a while loop starts until the queue is empty. The algorithm then checks all the possible spaces around it, including the plane in the next layer.

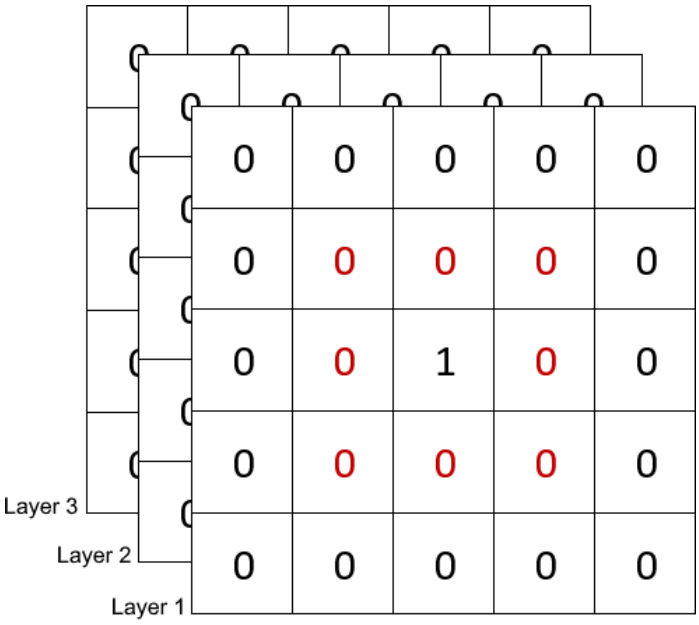


Figure 9: Check Surrounding Nodes In Current Layer

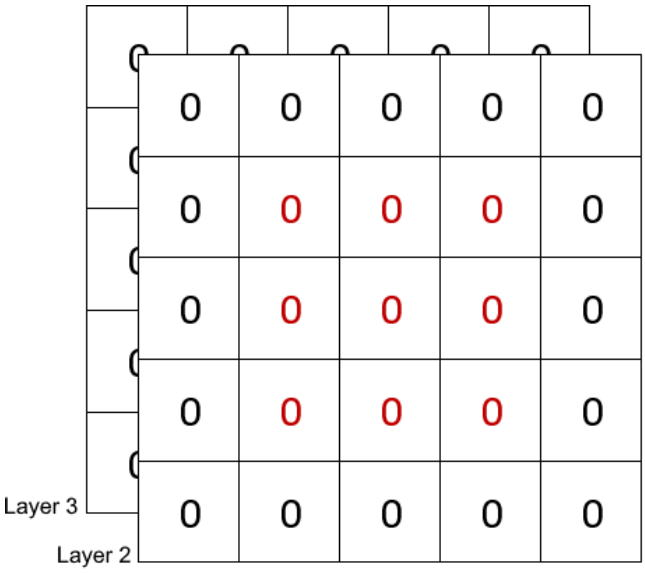


Figure 10: Check Surrounding Nodes In Next Layer

These surrounding coordinates are iterated through and a random number between 0 and 1 is generated for each space. A user defined weight is used to calculate the chance one of these coordinates will turn into a node. The formula is $\frac{weight}{PossibleConnections}$.

In this case let's set the $weight = 5$, so the chance of creating a new node is $\frac{5}{17} = 29.4\%$ where 17 is the number of surrounding spaces as denoted by the red values in the previous image. Therefore, if the random number that's generated is less than or equal to the calculated chance value, $randVal \leq \frac{weight}{PossibleConnections}$, then the 0 is changed to a 1, otherwise the 0 becomes a -1. If a node is successfully created, then that new point is added to the queue. Below is an example of what the 3D array and queue would look like after running the algorithm on the starting point.

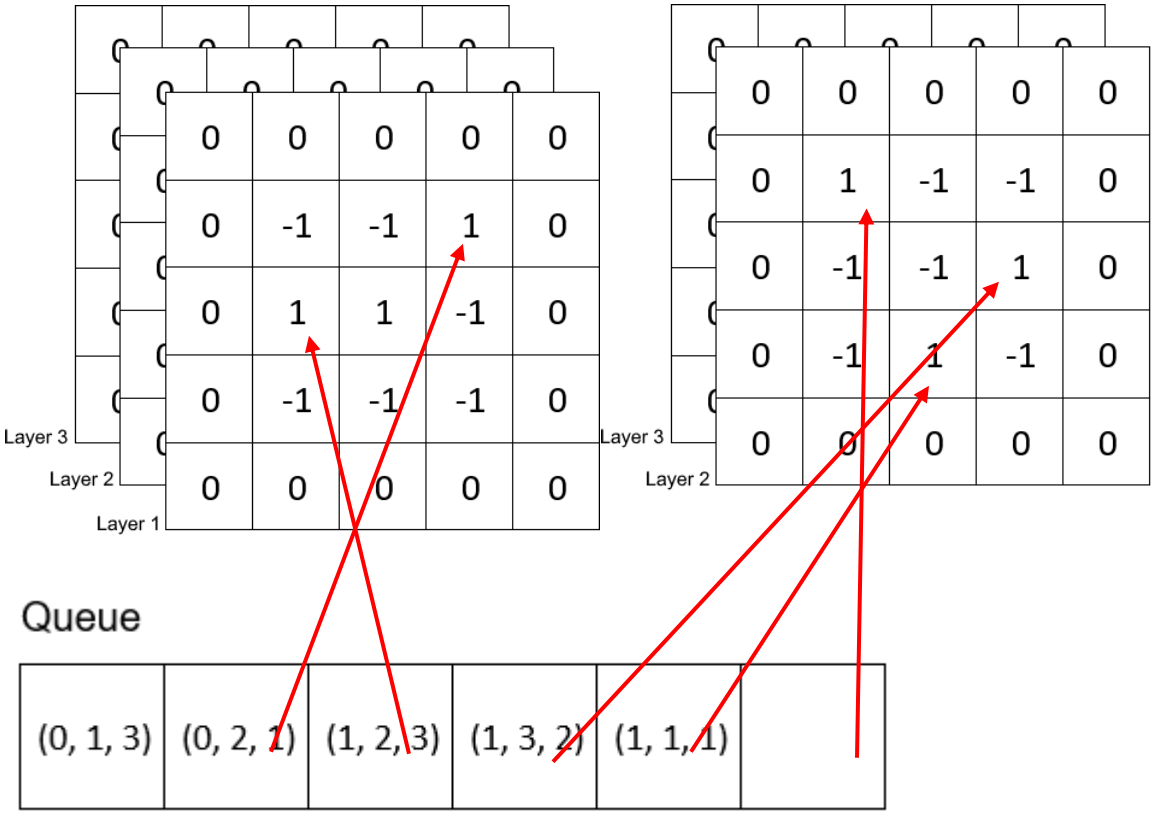


Figure 11: Queue Visual

After completing the algorithm on the starting point, the while loop takes the next point in the queue, (0, 1, 3) and runs the algorithm again.

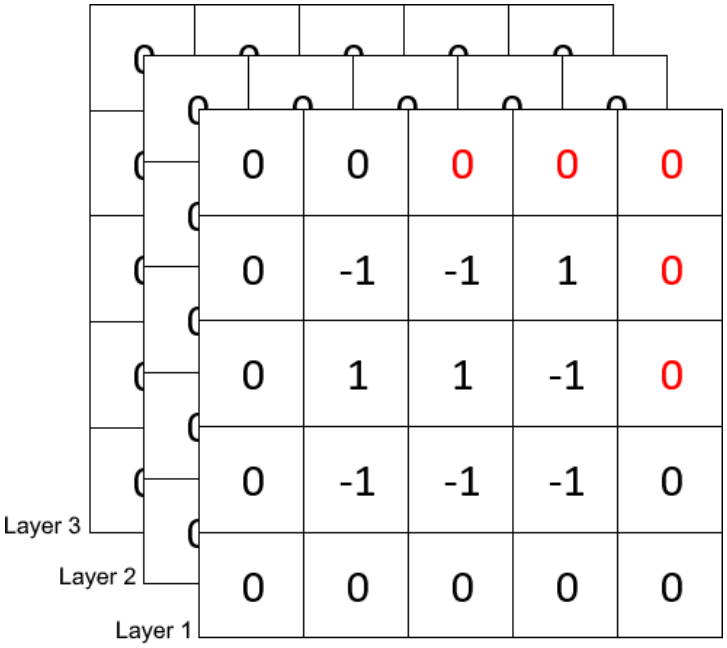


Figure 12: Check Surrounding in Current Layer

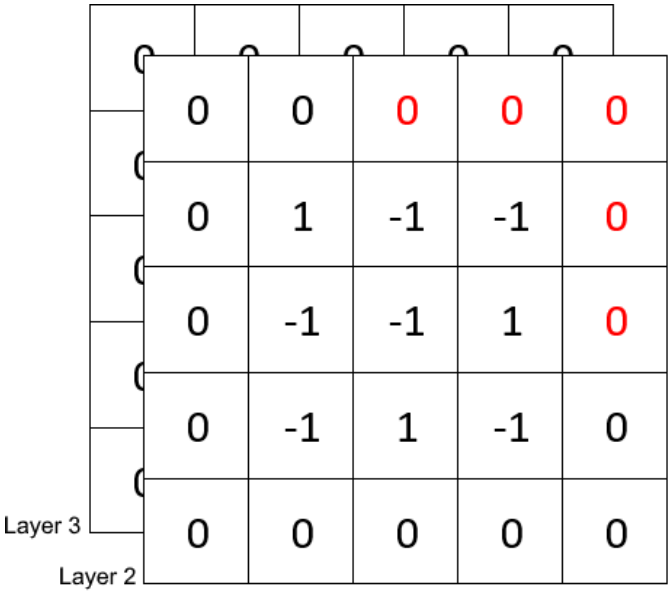


Figure 13: Check Surrounding in Next Layer

The algorithm will run until it generates half of the layers, $\frac{z}{2}$. So, for this example lets expand the layers to 5. In this case, the algorithm will generate the first 2 layers and stop at layer 3.

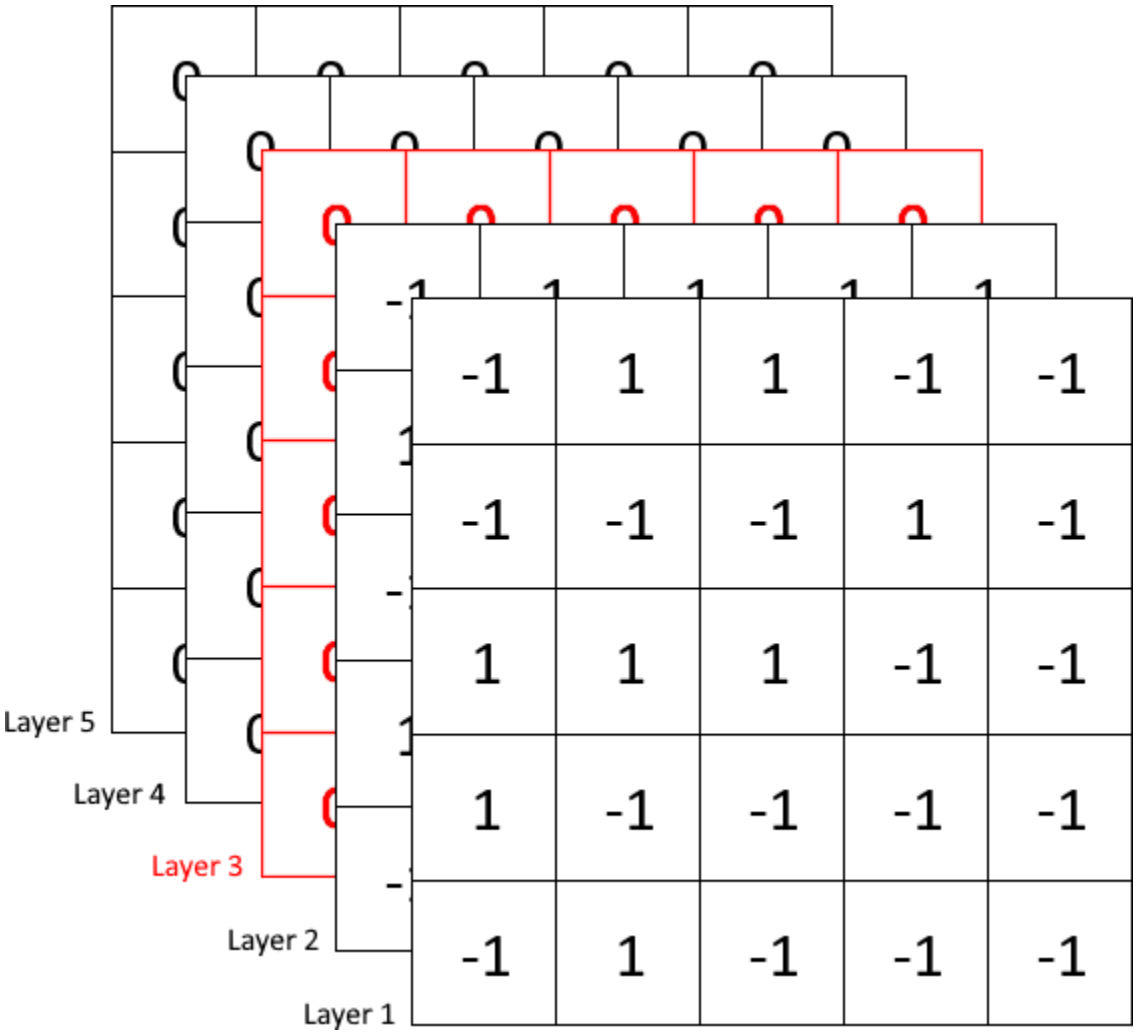


Figure 14: Stop at Middle Layer

The BFS algorithm is complete once half the layers are generated, and the queue is empty. The next step in creating the game board is to ensure that both sides of the board are equal. The generated half is mirrored to the second half of the board.

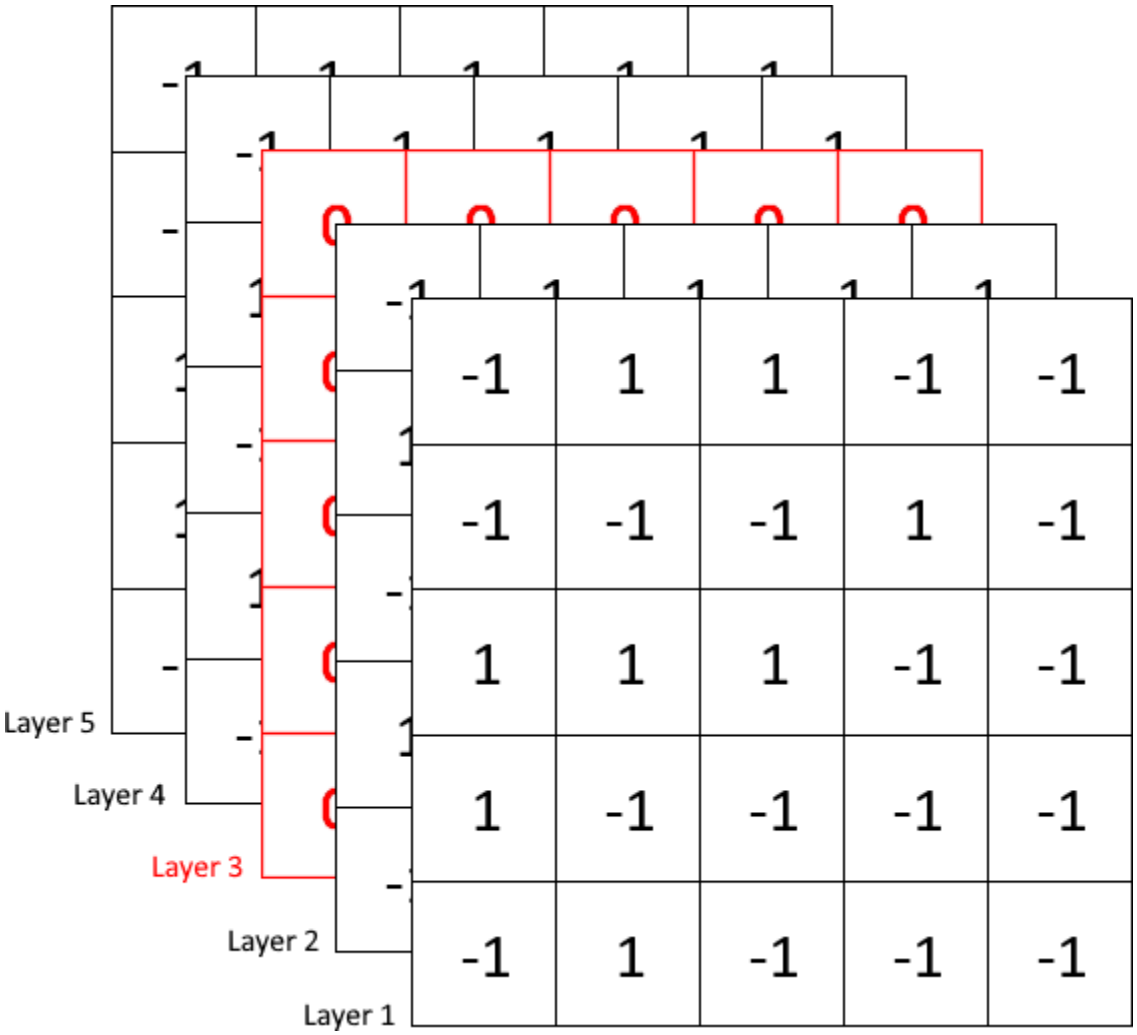


Figure 15: Mirror Game Board

In the case that there are an odd number of layers, the middle layer will be empty. To address this issue, simply iterate through the 2D array linearly and generate random nodes based on the weight and surrounding connections. After the middle layer has been generated you now have a complete 3D array that represents a 3-dimensional gameboard.

Similar to the algorithm generating a 2-dimensional map, nodes are randomly assigned either a Fortress or Watchtower based on set weight values and a random number from 0 to 1. For instance, the Fortress weight can be set to 0.2 and the Watchtower weight can be set to 0.8. When generating a random number from 0 to 1, a number less than 0.2 will spawn a Fortress and a number greater than 0.8 will spawn a Watchtower, and the remaining 60% will set the node to neither. These weights can be changed within the code to accompany the user preference for map generation. The values in the data structure will change from a 1 to either a 2 or 3 based on what node type is generated.

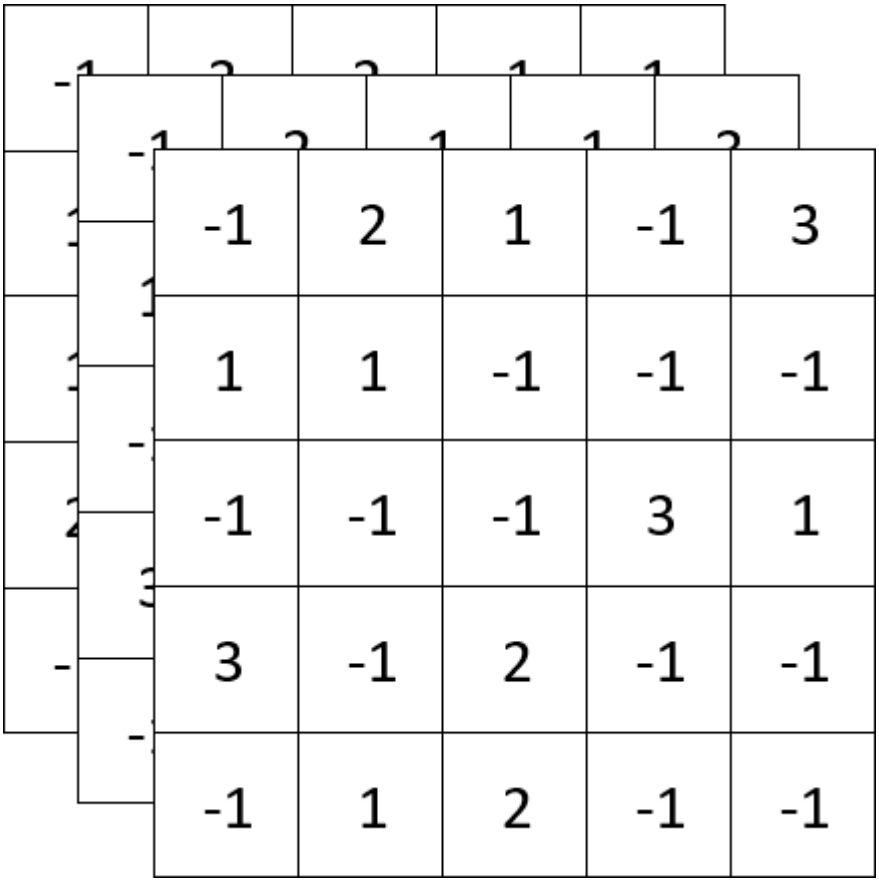


Figure 16: Different Node Types

Establishing Connections Between Nodes

To determine which nodes are connected to each other you would need to iterate through each index of the array and check if the surrounding spaces are nodes. All the connected node data will be stored in a json file so that the python server will be able to read the game board. This json file matches the format for the 2D map to avoid unnecessary implementations in the python server. Additional data for the length, width, and depth must be included since the size of the 3D space is variable.


```
{
    "__type": "Map:#Everglades_MapJSONDef",
    "MapName": "3DRandom",
    "Xsize": 7,
    "Ysize": 7,
    "Zsize": 10,
    "nodes": [
        {
            "Connections": [
                {
                    "ConnectedID": 31,
                    "Distance": 3
                },
                {
                    "ConnectedID": 74,
                    "Distance": 3
                },
                {
                    "ConnectedID": 82,
                    "Distance": 3
                },
                {
                    "ConnectedID": 81,
                    "Distance": 3
                },
                {
                    "ConnectedID": 80,
                    "Distance": 3
                }
            ],
            "ID": 25,
            "Radius": 1,
            "StructureDefense": 1,
            "TeamStart": 0,
            "ControlPoints": 500,
            "Resource": []
        },
```


Figure 17: Map JSON Sample

Each node has its own ID so that it can be distinguished by the server when looking for connection nodes. The ID is calculated with this formula:

$$(y * xLen) + x + (xLen * yLen * z) + 1$$

This will output into the following format.

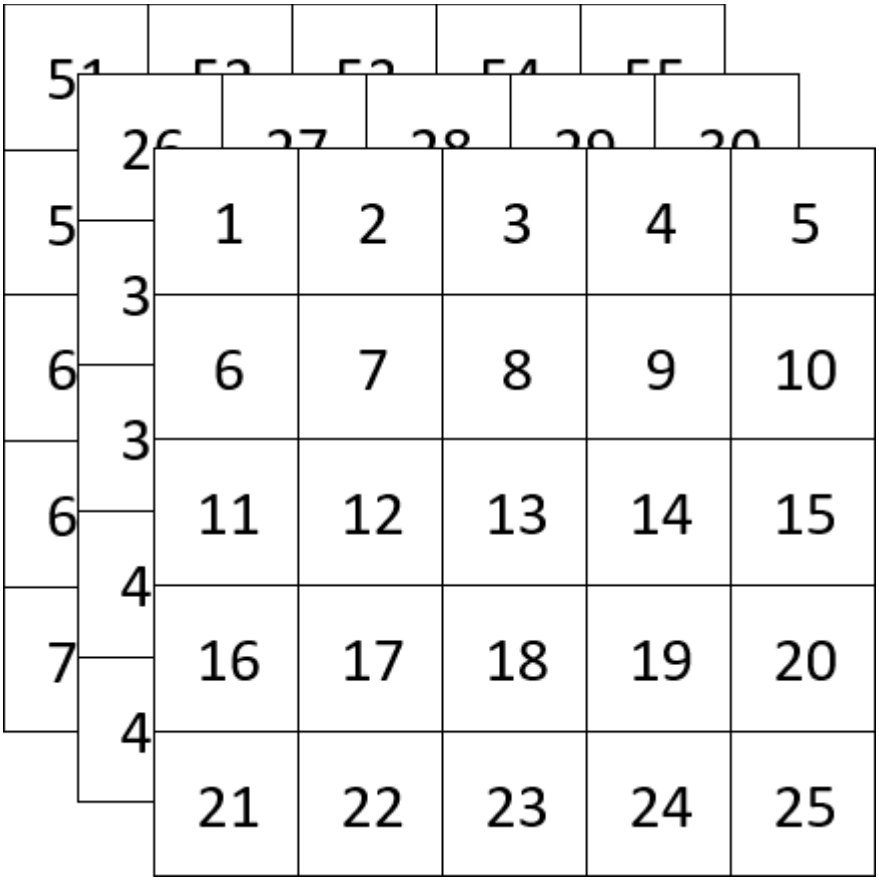


Figure 18: Node IDs

The ControlPoints determines how many points are awarded for controlling that node. The Resource array holds the type of node, either “DEFENSE” for a fortress or “OBSERVE” for a watchtower. TeamStart determines if it is the red base or blue base.

3D Bell Curve Feature

A bell curve node distribution algorithm was implemented into the breadth-first-search algorithm to give the map a more uniform distribution. The node creation weight is changed depending on what layer the algorithm is currently on. The weight will be the lowest when at the two ends of the map and the weight will be the greatest in the middle of the map. The formula used to determine these weights is an upside-down parabola with custom translations.

$$y = -\left(\frac{x}{\frac{len+1}{2}} - 1\right)^2 + 1$$

Here is what the function would look like when $len = 5$. Extra padding is implemented into the function to assure that the first and last layer do not generate a value of 0.

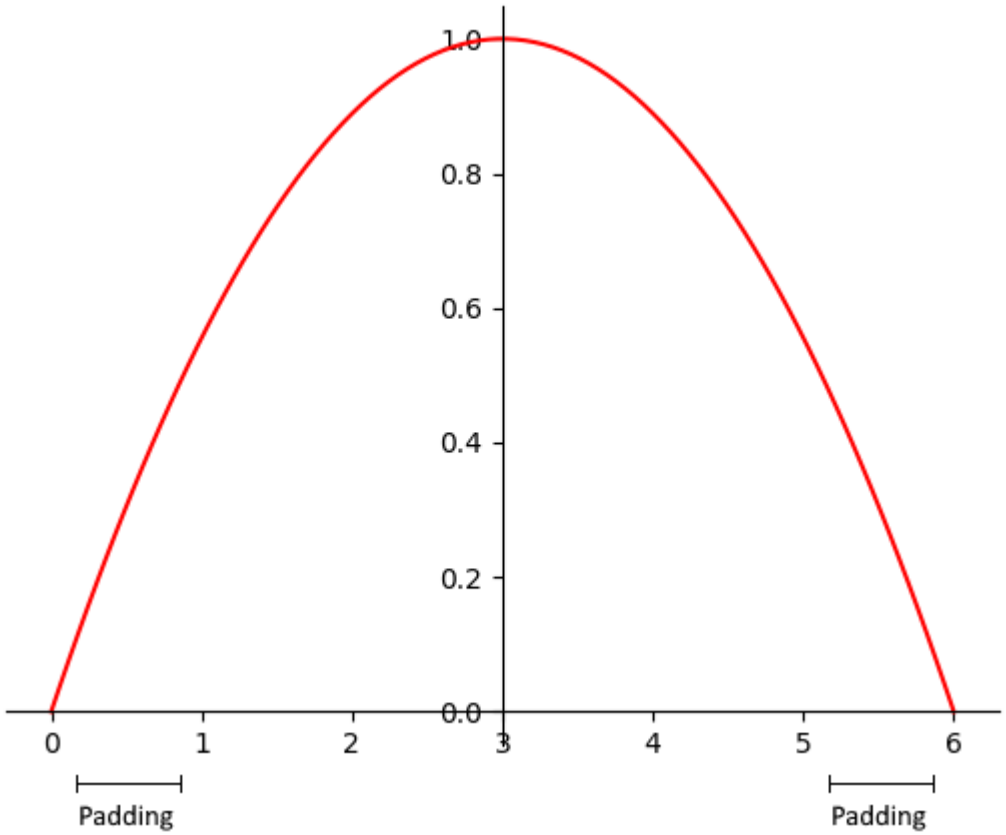


Figure 19: Bell Curve Function Graph

The function is implemented into Python so it can be called during each iteration.

```
def bellCurveVal(x, len):
    return - ( pow((x / ((len + 1) / 2)) - 1, 2)) + 1
```

Figure 20: Bellcurve function

In order for this feature to work, you need to also pass in the length for all three dimensions, x, y, and z. Then apply the bell curve weight to each of the dimensions. If the z length is the only parameter that is passed, then the distribution will create unwanted behaviors. This is because the y and x axis needs to be applied the variable bell curve weight as it strays from the center.

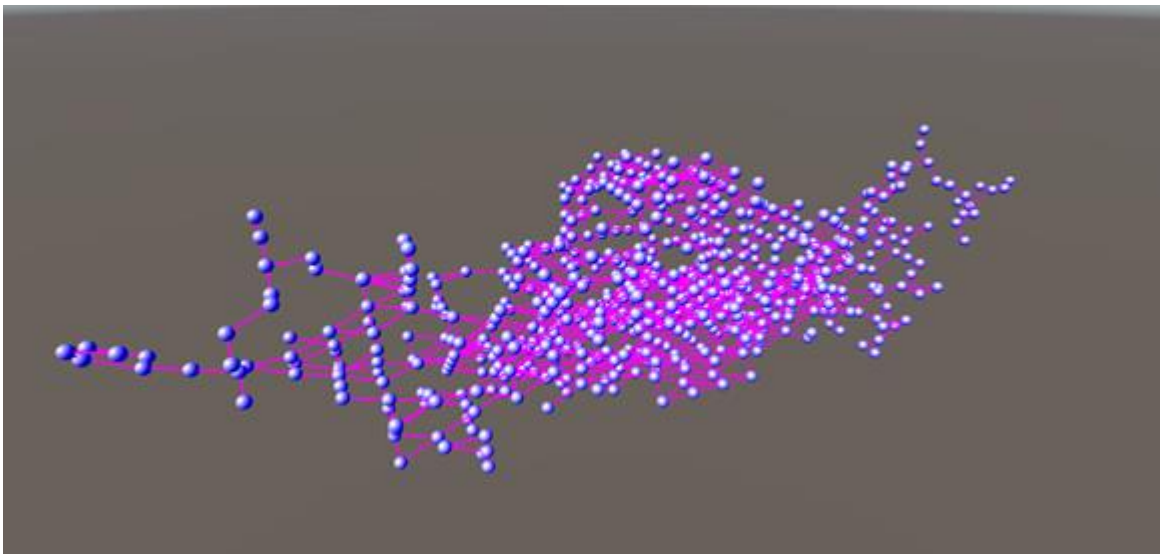


Figure 21: Bell Curve Map

The image above is a crude example of the bell curve feature implemented into the map, without any optimizations or adjustments to the weights. As you can see the two ends of the map are less dense than the middle. This funnels the interactions between the two teams into the middle of the map.

3D Gameboard Renderer

Overall Requirements

As the 3D gameboard was being developed, it became quickly apparent that simple text in an output terminal would not be sufficient enough to ensure that the map generation algorithms were working correctly. Thus, we decided that it would be necessary to create a renderer that would visualize the JSONs generated by the algorithms.

The overall requirement for this component is very simple: create a program that can display the 3D gameboard and its connections accurately based on supplied JSON data. Additionally, the user should be able to interact with the gameboard by moving the camera to view it at different angles.

Design Approach

The initial thought process was to use the Unreal Engine to render the 3D gameboard, as there was a pre-existing Unreal component that rendered 2D gameboards successfully and processed the telemetry data supplied to it. Thus, development began with creating a standalone map renderer using Unreal.

The troubles with this approach were apparent from the beginning, however. The visualizer could either be programmed using Unreal's blueprint system or utilizing C++ files. The issue with both of these approaches is that the existing JSONs were difficult to process in Unreal, and there is a dearth of information online pertaining to processing JSONs in Unreal in C++.

What information did exist on the Internet about processing JSONs in Unreal suggested using the blueprint system. At the time, the JSONs were being generated in a 3D array to represent the gameboard in a straightforward manner, but the blueprint system was not flexible enough to support three-dimensional arrays.

With these hurdles in the way, it became increasingly apparent that Unreal would be too cumbersome to use for the desired program. It is wholly possible that, with enough time, patience, and experience, an Unreal gameboard visualizer would be achievable. However, the need for a gameboard visualizer was so great that spending the time to learn Unreal and figure out the issues presented was not a viable option.

Thus, it was at this point that the switch was made to Unity. While using this engine would mean that we would not gain any possible code that could later be applied to the Unreal component of Everglades, it did mean that a visualizer could be made within a short timeframe. The experience we had with Unity was far greater than our collective experience with Unreal, making the creation of a visualizer in this engine much more feasible.

Unlike Unreal, the Unity engine primarily utilizes the C# language, though it can also support Javascript. Additionally, we knew from personal experience that Unity has a very supportive community with a great deal of support, which provided us with assurance if we had encountered a problem that was outside the scope of our Unity expertise.

Unity can very easily handle a lot of processes using nothing but C# files. Using this approach, one could simply render each node individually along with any line connections and create a way to move the camera around easily. For all of these reasons, Unity ended up being the chosen platform for creating the standalone gameboard renderer.

Implementation

Naturally, the current implementation of the gameboard renderer takes the Unity approach. While it is conceivable that the Unreal Engine approach would be possible with enough configuration, research, and trial and error, as stated previously, there was not enough time to get all of this underway. We had set goals to finish the basic requirements as soon as possible, and the need for a working gameboard renderer was higher than a need to make it in the Unreal Engine.

To provide a brief overview of the gameboard renderer, it follows a few very simple steps: it first reads in the JSON file, deserializing its information into a custom Gameboard object.

It will then fill the dimensions of the gameboard designated by the JSON with instantiated node objects. Every space within the dimensions is filled with a node object even if the generated map does not place a node at that position. After a node is instantiated, its active status is immediately set to false, meaning it will not appear visually within the renderer.

Next, the nodes specified in the JSON are looped through, and wherever a node is designated, the corresponding node that was instantiated in the previous step is set to active, making it appear within the renderer. During this step, if the node has a special resource – like a watchtower or stronghold – or is designated as a team start position, the node is colored to visually show that it has a special attribute.

Once a given node is evaluated in full, its various connections are evaluated and it draws lines between the source and destination node to show that there is a connection.

Now that the brief overview has been given, the programming that powers the renderer can be viewed briefly as well, in hopes of gaining a more complete understanding of the current implementation.

JSON Input

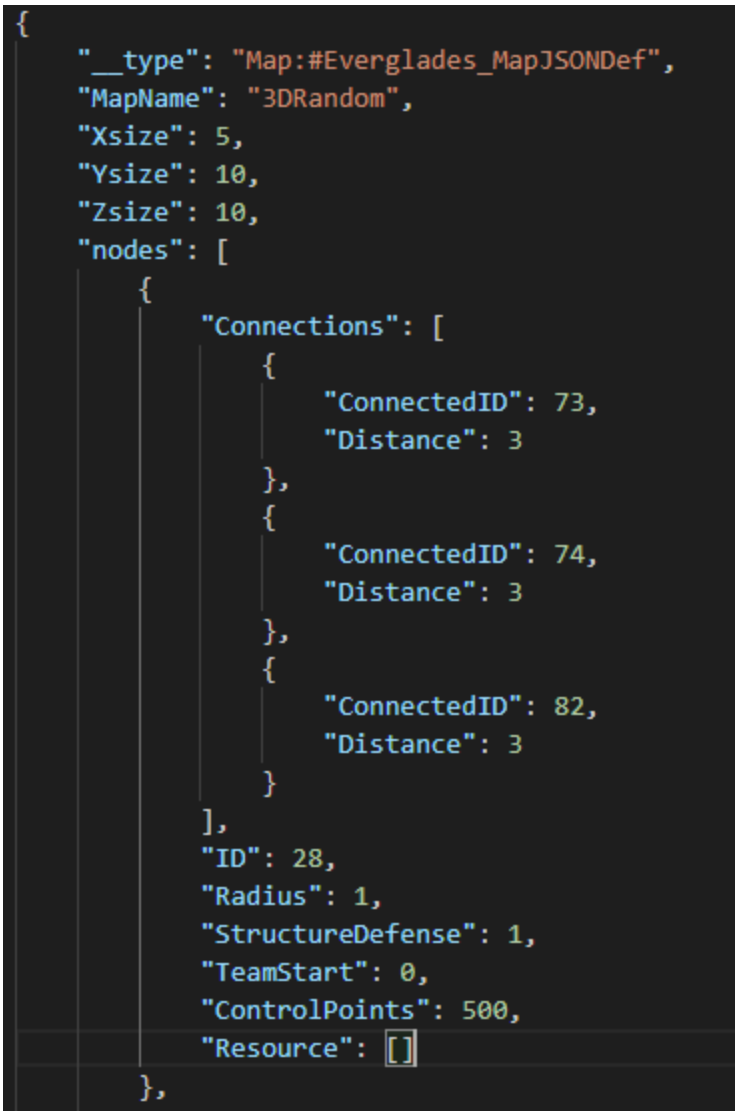
Firstly, the JSON should be evaluated to give a better understanding of what the renderer takes in as input. One should note that – in the built version of the renderer – JSON gameboard files should be placed within the following directory:

CubeRenderer/CubeRenderer_Data/StreamingAssets

The renderer will read in the first alphanumeric file it finds within this directory. So, a file named “1.json” will be chosen and rendered instead of a file named “map.json” since it appears first alphanumerically.

Now, the JSON format will be dissected to give a better understanding and appreciation for how the gameboard renderer operates. The first node of a generated 3D gameboard is shown in the image below, along with some of the extraneous values that are required while the gameboard is being generated. The first two fields, *__type* and *MapName*, are values used internally by the Python server files, and thus they do not need to be explained within the context of the map renderer. The *Xsize, Ysize,* and *Zsize* values correspond to the dimensions of the generated gameboard.

Finally, there is the *nodes* array, which keeps track of the various nodes associated with this gameboard. Each node has a *connections* array, which is a list of nodes the given node is directly connected to, along with the distance of the connection. The *ID* is a unique identifier given to each node. The *structureDefense* is a value used in damage calculation which varies between nodes. The *TeamStart* is a value indicative of whether a team – 0 or 1 – has its home base at that node. If it is -1, then the node is not the home base of a team. The *controlPoints* dictates how many points a player will get for capturing this node. Finally, the *resources* array contains the resources – like watchtowers or strongholds – that generate at that node.



```
{
    "__type": "Map:#Everglades_MapJSONDef",
    "MapName": "3DRandom",
    "Xsize": 5,
    "Ysize": 10,
    "Zsize": 10,
    "nodes": [
        {
            "Connections": [
                {
                    "ConnectedID": 73,
                    "Distance": 3
                },
                {
                    "ConnectedID": 74,
                    "Distance": 3
                },
                {
                    "ConnectedID": 82,
                    "Distance": 3
                }
            ],
            "ID": 28,
            "Radius": 1,
            "StructureDefense": 1,
            "TeamStart": 0,
            "ControlPoints": 500,
            "Resource": []
        },
```


Figure 22: Sample from a map JSON.

There are three separate classes that allow for the JSON to be parsed. The *Gameboard* class handles the high-level variables like the dimensions and nodes contained within he gameboard. The *Node* class contains all variables that pertain to each individual node, with a node's connections being handled by the third class – *Connection*. These classes must be present within the renderer's code in order for the JSON parsing library – Newtonsoft – to work. The library deserializes the JSON and automatically assigns the values is finds to the variables of the same name.

Important Implementation Details

The code utilized by the gameboard renderer is relatively straightforward, as the renderer is meant to do its job in a sure-fire way without trying to pull off any fancy tricks. All of the renderer's capabilities is found squarely within the *Renderer*.cs file, which follows the steps outlined previously within this section, though this section will delve a bit deeper into the inner mechanisms.

The actual processing of the file is contained within Unity's *Awake*() function, which is called before the first frame update and before Unity's *Start()* function. Since there are no other scripts or classes trying to compete for a turn, there is no chance of a race condition occurring.

In the first set of loops instantiating all possible nodes within the gameboard and setting their active status to false, the coordinates they are given are entirely arbitrary. While they are based firmly on the x, y, and z values generated by looping through the three loops, the modifiers they are given of 1.5 for x and y and 3 for z were chosen manually by sight. These values allow for decent spacing between nodes on the x-y and y-z planes, allowing the nodes to be distinct enough from their neighbors.

In the second loop which sets the deactivated nodes to true if they are present within the gameboard, the only thing of note is the *MeshRenderer* that is applied to the instantiated sphere which represents the node. This mesh allows the renderer to change the color of the node to help depict the team start positions or the presence of a resource. Strongholds are colored gray, watchtowers are colored yellow, Player 0's start is colored red, and Player 1's start is colored blue.

At the very end of the second loop, lines are drawn to represent the connections between the starting node and adjacent nodes it's connected to. This results in a call to the *drawLine()* function, which utilizes Unity's built-in *LineRenderer* component to draw simple, effective lines.

The gameboard variable stores a 3D array comprised of all the nodes in the gameboard. As stated previously, this gameboard is made on integers, with a negative one indicating the lack of a node at a position, and other nonnegative integers – typically one – represent a present node. The x, y, and z variables store the respective dimensions of the gameboard.

```
// Creates the physical instance of the line in the renderer.
1 reference
void createLine()
{
    line = new GameObject("Line" + currLines).AddComponent<LineRenderer>();
    Material material = defaultMaterial;
    material.color = Color.white;
    line.material = material;
    line.positionCount = 2;
    line.startWidth = 0.15f;
    line.endWidth = 0.15f;
    line.useWorldSpace = false;
    line.numCapVertices = 50;
    currLines++;
}
```

Figure 23: The createLine() function in the Renderer.

After instantiating a new line and naming it based on the current amount of lines present within the gameboard, the function will then color it to make it a more visually-appealing color. Without coloring the material, the lines show up as an ugly, bright purple color which can make viewing the overall gameboard.

Result

The result of this program is, naturally, a gameboard renderer. It successfully renders a simplistic model of the gameboard comprised of spherical nodes and lines to represent the connections. What follows are a few sample gameboards which were rendered by the program.

The first set of gameboards shown are taken from the gameboard when it was still in early development. At this point in time, an older JSON format was used which represented the nodes and connections within a 3D array. While Unity was able to handle this format effectively, the new method outlined in the previous sections – whereby each node contains a list of nodes it is connected to – proves to be far easier to implement and maintain.

During this stage of development, it should be noted that the nodes are not colored to reflect any resources that are present on them, nor are they colored to indicate whether they are the start position for any team. Additionally, the lines are left to be their default color without a material applied, which at a far away distance can make the gameboard a visual amalgam of clashing colors.

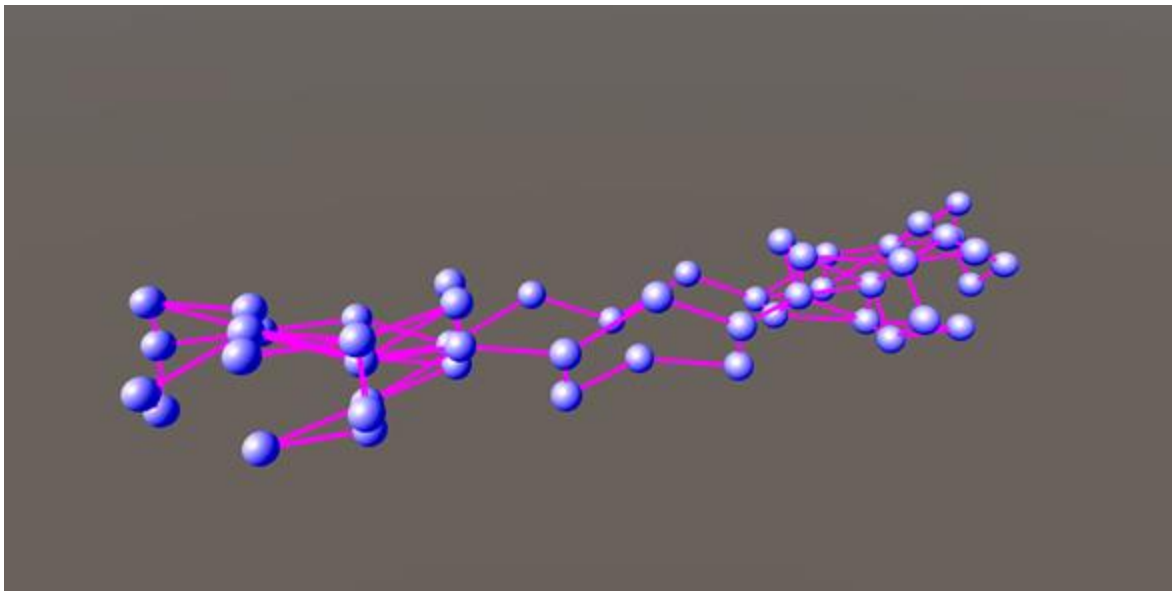


Figure 24: A very small, low density gameboard

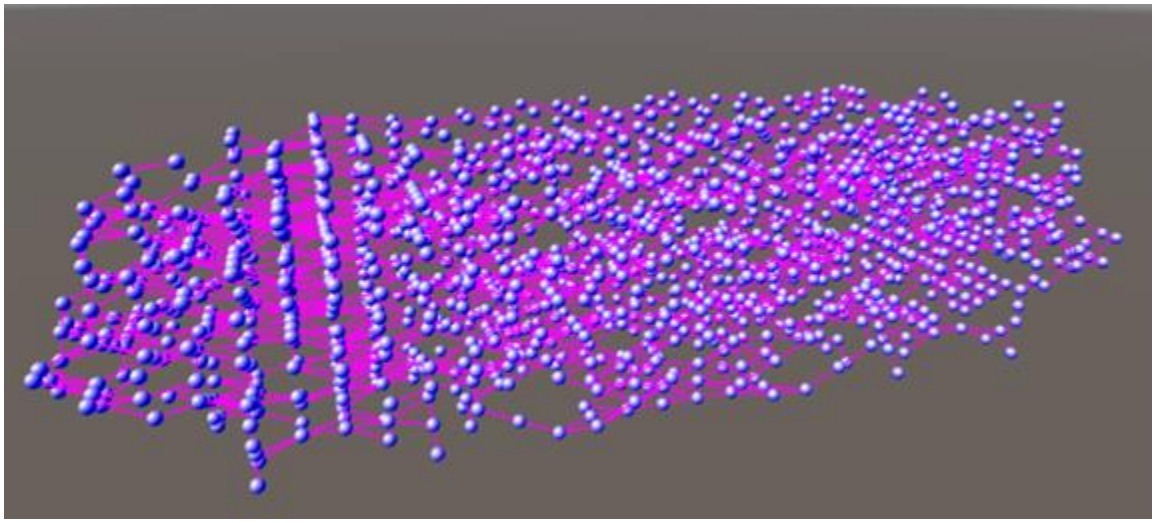


Figure 25: A very large, high density gameboard.

The following two images are of the current gameboards generated by the renderer. Note that they are much more colorful to denote the differing resources or team start positions of the nodes, with the lines having a material applied in an attempt to remove the ill effects of the untextured lines.

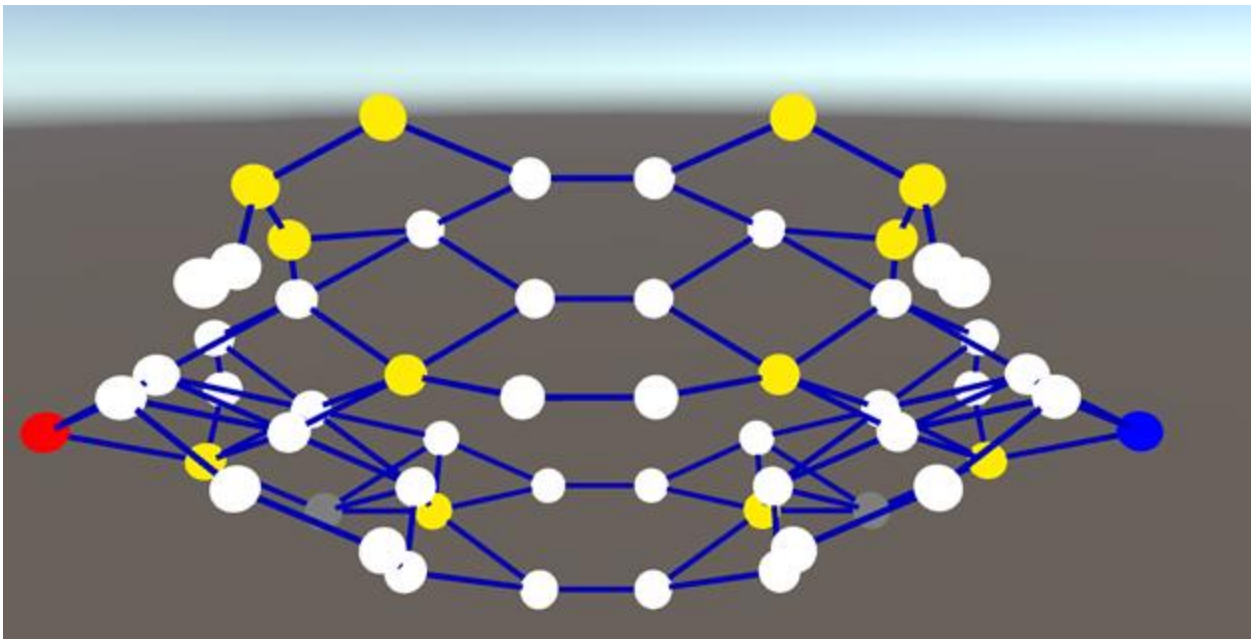


Figure 26: A simple map using the updated renderer.

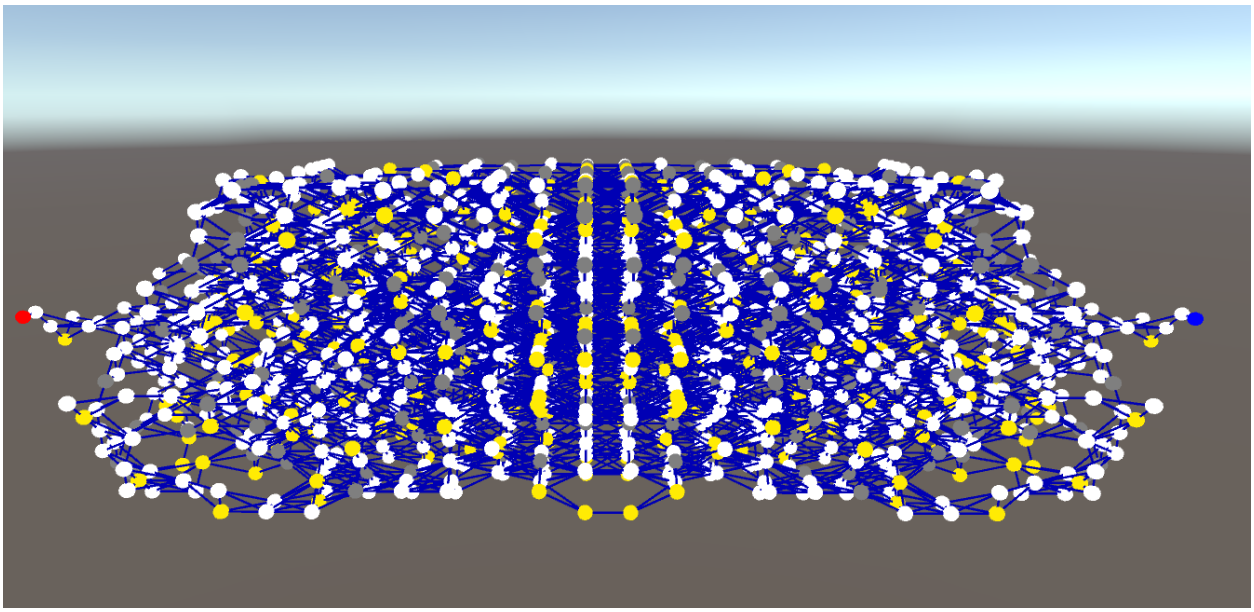


Figure 27: A massive map with the updated renderer.

The creation of this gameboard renderer has helped us develop the 3D gameboard requirement much faster than had we not created it. All one had to do was drop in a properly formatted JSON and run the program to see what the results of their gameboard generator were. In doing so, the results of the created algorithms were readily viewable, allowing us to determine if the algorithm was creating 3D gameboards in a desired fashion.

Using the Renderer

The renderer is a standalone program entirely separated from the various Python files directly attached to Project Everglades. To display a generated map, these steps must be followed:

1. Download the folder titled “CubeRenderer” and open it.
2. Navigate to *CubeRenderer/CubeRenderer_Data/StreamingAssets*
3. Place any JSON files that follow the given gameboard format within this directory.
    a. Name the file you want rendered such that it appears first in the file order alphanumerically.
4. Navigate back to the *CubeRenderer* folder and run *CubeRenderer.exe*
    a. If nothing appears, this means your JSON file is formatted incorrectly. Since this is a lightweight rendering program, its only purpose is to render the gameboards, and as such will not detect or notify of format errors.
5. If a gameboard appears, you can use the WASD keys and mouse to navigate the scene. Press the ‘ESC’ key at any time to gain control of the cursor again, allowing you to exit out of the program if desired.

This is all the information needed to effectively run and use the renderer. Currently, it has only been built, tested, and utilized on Windows devices, and has not been properly developed for Mac or Linux.

As stated previously, this renderer was created with only one real intention: visualize the 3D maps generated by the gameboard generation algorithms. It is not terribly GPU efficient – and will often struggle on larger maps – and will not alert the user of formatting errors. For these reasons, there is definite room for improvement, but since this program was developed solely as a tool to support only a single facet of this project’s development, spending much more time on development would not have been a good use of resources.

3D Wind stochasticity

With the implementation of the new 3D map generator. Additional adjustments had to be made to the wind stochasticity for the two features to be compatible. The previous wind algorithm uses coordinate points in two dimensions for the generation of simple perlin noise, so a new function has been created to handle a map of three dimension. This new function takes the length of each dimension and iterates through while sending the points into a simple perlin noise method that takes three points.

```
noise = snoise3((x + offset)/freq, (y+offset)/freq, (z+offset)/freq, octaves=1,
persistence=1)
```

This will generate random values that will be used for generating the wind vectors. To generate a three-dimensional vector, we use the relationship between spherical and cartesian coordinates. The formulas used are below, where ρ is the magnitude, and both ϕ and θ are in radians.

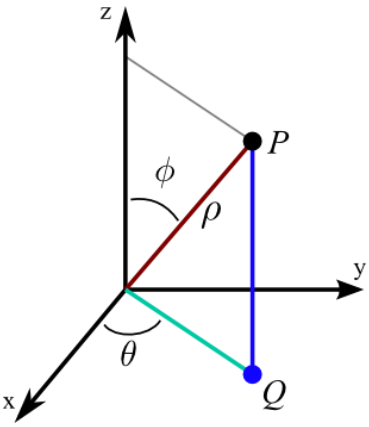


Figure 28: Spherical Coordinates [2]

$$x = \rho sin\phi cos\theta$$
$$y = \rho sin\phi sin\theta$$
$$z = \rho cos\phi$$

Phi and theta are generated from the random value by the following code, where $\rho \geq 0$, $0 \leq \theta \leq 2\pi$, and $0 \leq \rho \leq \pi$

```
theta = noise * 2 * math.pi
phi = noise * math.pi
```

From here the cartesian coordinates are generated using the spherical coordinate formulas

```
x_dir = magnitude * math.sin(phi) * math.cos(theta)
y_dir = magnitude * math.sin(phi) * math.sin(theta)
z_dir = magnitude * math.cos(phi)
```

Here are some example models of the wind maps that can be generated using this algorithm.

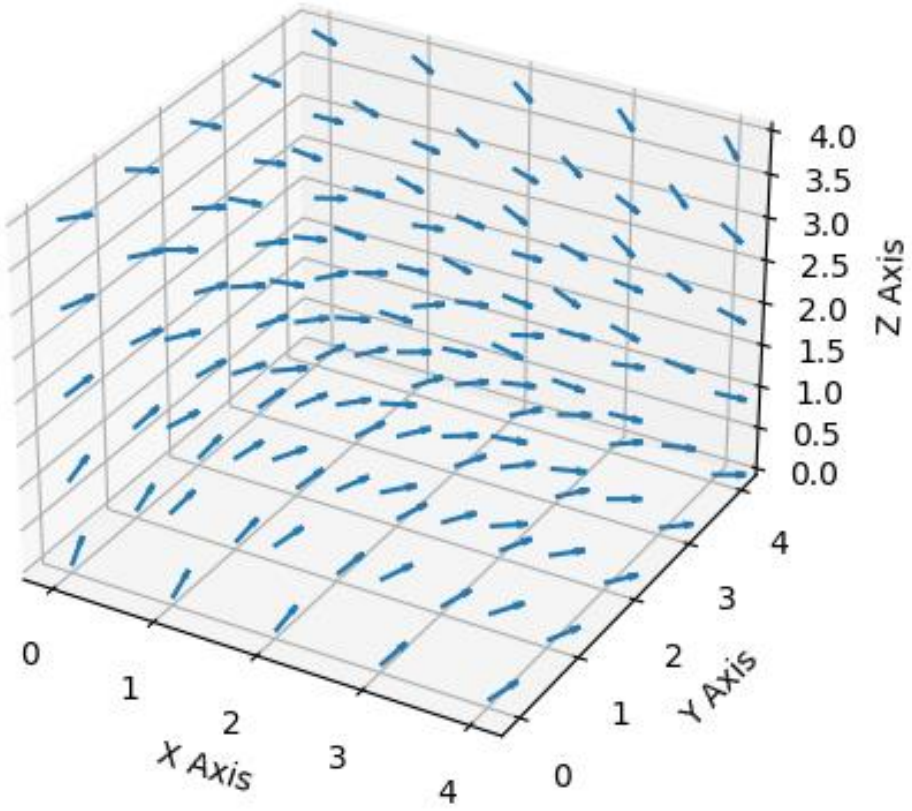


Figure 29: Random Wind Model

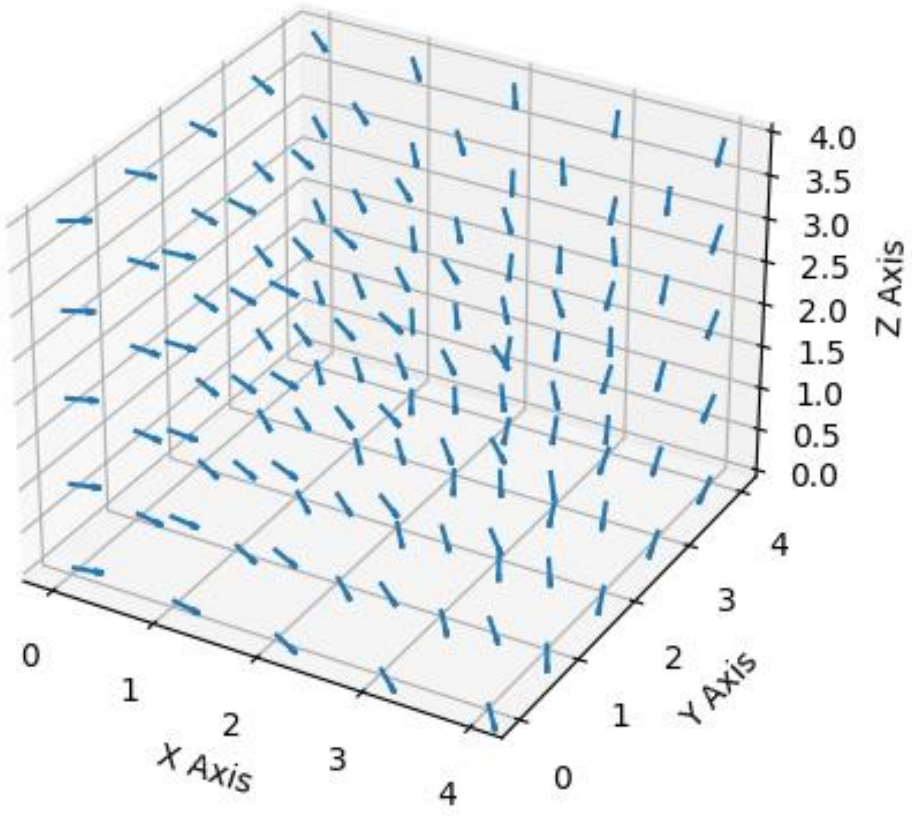


Figure 30: Random Wind Model

However, to keep the maps for Everglades consistently fair, the wind models are mirrored so that both ends of the map have the same wind model.

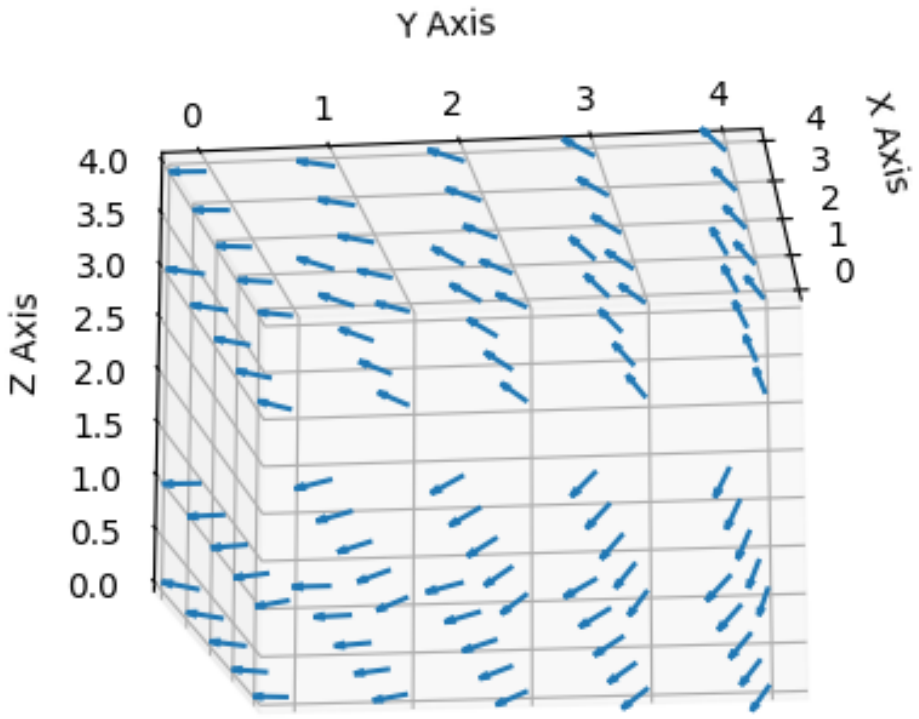


Figure 31: Mirrored Wind Model

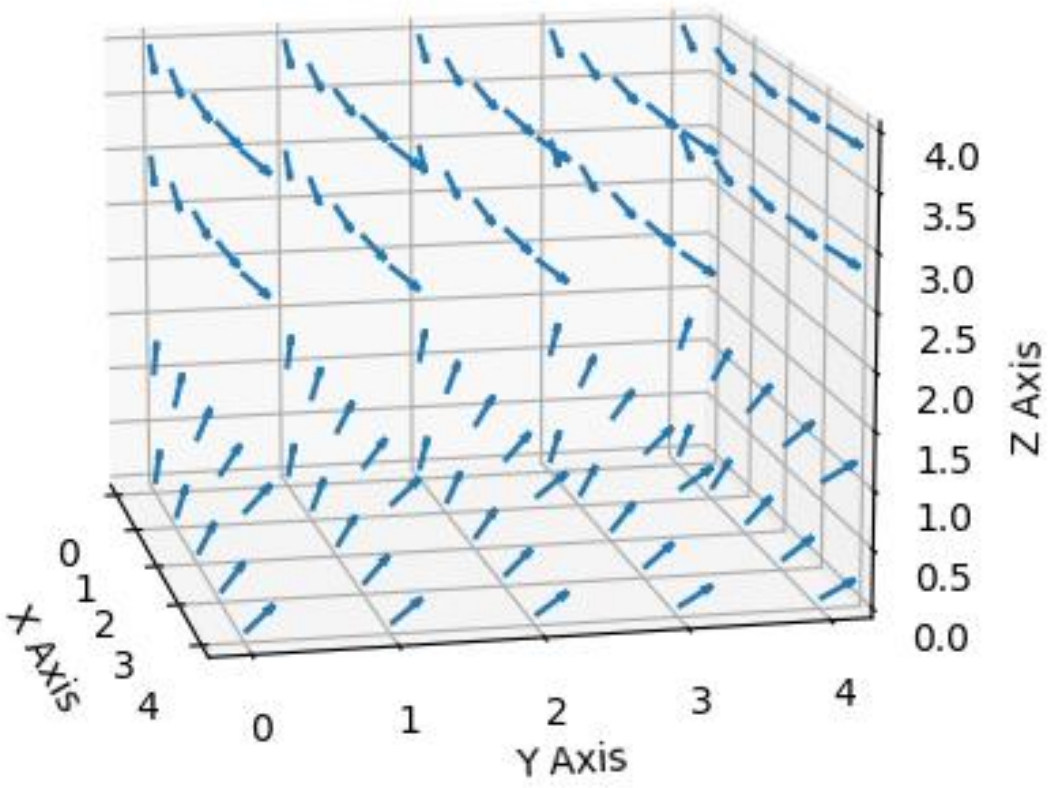


Figure 32: Mirrored Wind Model

Drone Loadout Customization

For the loadout requirement, the desired functionality to be added to the server-side aspect of Project Everglades, was to allow the player, whether AI or human, to pick and choose which drone types would be in each squad.

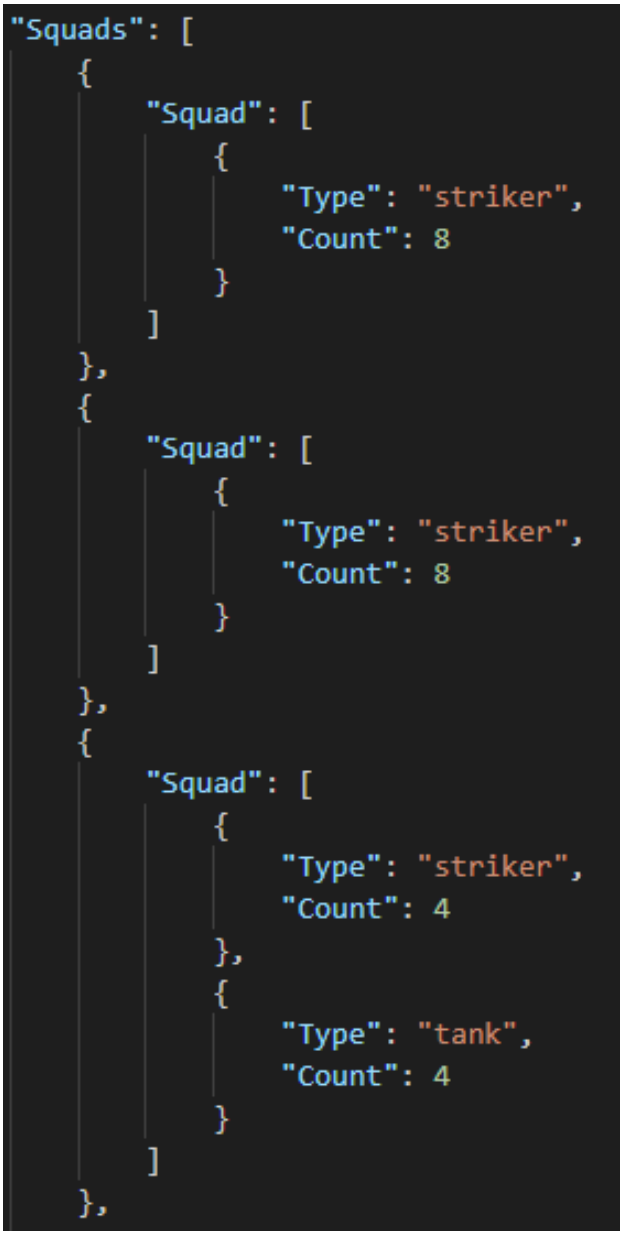
Storage and Loading

The decision was made to solve this problem by adding a way to create a loadout and save it to a json file, and then have the game load in the json file for each player on game start. This method was chosen for a variety of reasons:

- JSON is a user-friendly solution to viewing and editing the data, which helps both programmers and designers for the game, and for the AI using the game. As there are groups working on creating AI for this project, having this format allows those groups to both understand the loadouts better, and gives them an easy way to edit data.
- By saving loadouts, a default loadout can be created, presets for loadouts that are considered ‘balanced’, ‘default’ or ‘custom by user’ can be created. This allows for the implementation of standardizations such as if used for a competition, ‘default loadouts’ or ‘default and balanced’ could be specified as rules toward the match to help confine the scope or maintain desired balances.
- This method is intended to be easier for a future team to implement on the Unreal gaming engine end, which this project does not contain in scope. The reasoning is that on the Unreal side, a loadout can be created before game by any human players, or by anyone choosing the AI’s loadout for them. Then after saving the loadout, on game start the loadout will be ready for the Server side to use.
- It was decided to use json loading and saving for multiple other aspects, and the project already contained some json loading, therefore it was determined that multiple aspects of code could be reused for efficiency and organization.

JSON Format

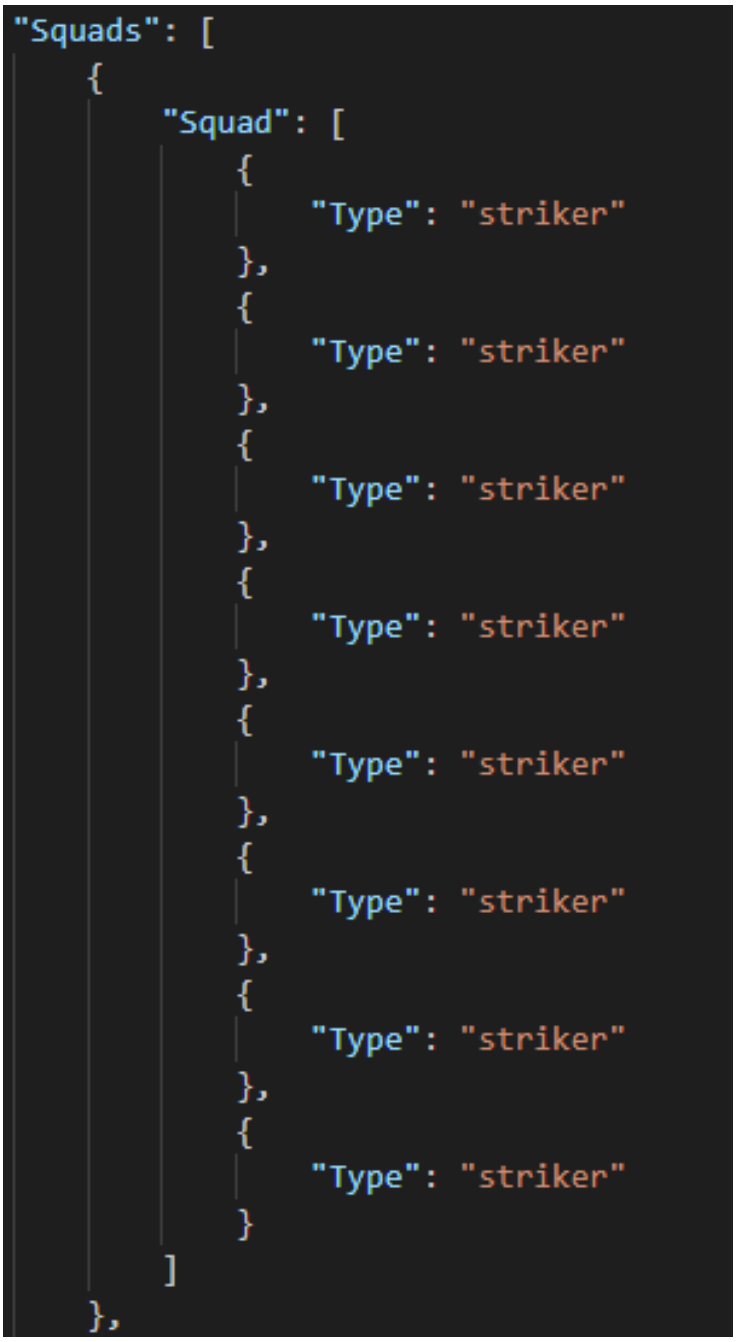
A new python file was added to the project, CreateJsonData.py, to contain the functions for saving and loading JSON data. With the format shown below, the first three squads are shown. The structure matches how the game recognizes and stores unit types and allows for a readable format for viewing.



```
"Squads": [
    {
        "Squad": [
            {
                "Type": "striker",
                "Count": 8
            }
        ]
    },
    {
        "Squad": [
            {
                "Type": "striker",
                "Count": 8
            }
        ]
    },
    {
        "Squad": [
            {
                "Type": "striker",
                "Count": 4
            },
            {
                "Type": "tank",
                "Count": 4
            }
        ]
    },
```

Figure 33: Json file representation of the first three squads of a loadout

This format was chosen over the format below due to being more readable and due to the game logic grouping together unit types on initialization.



```
"Squads": [
    {
        "Squad": [
            {
                "Type": "striker"
            },
            {
                "Type": "striker"
            },
            {
                "Type": "striker"
            },
            {
                "Type": "striker"
            },
            {
                "Type": "striker"
            },
            {
                "Type": "striker"
            },
            {
                "Type": "striker"
            },
            {
                "Type": "striker"
            },
            {
                "Type": "striker"
            }
        ]
    },
```


Figure 34: A JSON representation of one single squad

Original Drone Loadout

The current implementation of the drone loadouts in the project is utilized as a dictionary that contains values that are lists of tuples. Each tuple that exists inside of the list represents the quantity of a specific unit type.

```
unit_config = {
0: [('controller',1), ('striker', 5)],
1: ['controller',3), ('striker', 3), ('tank', 3)],
2: [('tank',5)],
3: [('controller', 2), ('tank', 4)],
4: [('striker', 10)],
5: [('controller', 4), ('striker', 2)],
6: [('striker', 4)],
7: [('controller', 1), ('striker', 2), ('tank', 3)],
8: [('controller', 3)],
9: [('controller', 2), ('striker', 4)],
10: [('striker', 9)],
11: [('controller', 20), ('striker', 8), ('tank', 2)]
}
```

Figure 35: Previous hardcoded implementation

The drone loadout was originally stored inside of the variable “unit_config” which is a dictionary. The keys of the “unit_config” represent an index for each group of drones that exists for a given player.

The drone loadout dictionary has an extra requirement that is not explicitly stated within its contents. The loadout for the drones must have a total number of 12 groups of drones. It is not acceptable to have fewer or more groups of drones within a loadout.

The first element in the tuple represents a unit type of the drones that are present in the particular group of drones. The first element in the tuple has a value of a string and explicitly determines which type of drone that is existent in the context of that particular current group of drones.

The second element that is existent in the tuple has a value with that of an integer. The second element of the tuple dictates the total amount of drones of the same unit type that exist in the context of that particular group of drones.

Original Assertions

The previous team had fortunately implemented a method of checking to make sure that each player's loadout of drones follows the right format to be properly implemented. The main function that is utilized are assertions which help to debug problems that could arise from an improper drone loadout. The following code is how the python server checks if the drone loadout is in the correct readable format.

```
    for pair in player_dat[player]['unit_config'][gid]:
        assert(type(pair) is tuple),
         'Group array must contain tuples'

        assert(type(pair[0]) is str),
         'Unit type must be a string'

        assert(type(pair[1]) is int),
         'Unit count must be an integer'

        in_type, in_count = pair
        in_type = in_type.lower()
```

Figure 36: Overall code that validates drone loadout

This function for checking the contents of a drone loadout must first go through every group that is owned by that of the player and ensures that it is formatted correctly. To access an instance of a group, the player's data must be dereferenced using the id of which player to look at, in which the unit configuration must be referenced inside of that particular player. To get a specific group list of tuples, the group id must be accessed as well. The following excerpt of code that accomplishes the search for each group definition is as follows.

```
    for pair in player_dat[player]['unit_config'][gid]:
```

Figure 37: Code that iterates drone loadout

The first assessment is for the purpose of ensuring that each element in the group list exists as a tuple. If the information stored for the group does not have the format of a

tuple, it will raise an Assertion Error. The Assertion Error will be relayed with a message that the list of groups of drones must be comprised of tuples.

```
    assert(type(pair) is tuple),
     'Group array must contain tuples'
```

Figure 38: Code that checks that element is a tuple

The next assertion checks that the first element of the tuple inside of the list of groups has a type with that of a string. If the value of the first element inside of the tuple is not represented as a string, an Assertion Error is raised. The Assertion Error will be relayed with a message that states that the unit type of the tuple must be represented as a string.

```
    assert(type(pair[0]) is str),
     'Unit type must be a string'
```

Figure 39: Code that checks that first tuple element is a string

The next assertion for the tuple is to check for the second element of the tuple. This check is to ensure that the second element of the tuple has a value of an integer. If the second element inside of the tuple does not have a type of an integer, then it causes an Assertion Error. The message relayed by the Assertion Error will state that the unit count element of the tuple must be an integer.

```
    assert(type(pair[1]) is int),
     'Unit count must be an integer'
```

Figure 40: Code that checks that second tuple element is an integer

Once the contents of the tuple and the tuple itself are assessed by whether they represent the right data types, more intricate assessments are performed. To obtain the actual values of the elements inside the tuple, variables are put in place to make handling their content more manageable. The following line of code does just this for the tuples' first and second element.

```
    in_type, in_count = pair
    in_type = in_type.lower()
```

Figure 41: Code that copy and format tuple values

The variable pair is a reference to the current tuple that is being assessed in the loop throughout the group array of tuples. The variable “in_type” now has a data type of a string and has the same content as the first element in the tuple. The variable later gets converted to become a string of lowercase letters. This is to help with later asserting validity. The variable “in_count” is now assigned with a data type of an integer and contains the value of the second element of the tuple.

The first assessment is to determine whether the first element of the tuple actually references a drone type that exists in the context of the game. The string can have a value of any type of drone that the python server can access data for to better represent. If a string that does not explicitly have a value that is existent in this context, it will cause an error. The way that the python server checks for a valid unit type is as follows.

```
    assert(in_type in self.unit_names),
     'Group type not in unit type config file'
```

Figure 42: Code that checks the group type is a valid unit type

If the first element in the tuple cannot be used to reference a type of drone that exists in the context of the game, then an Assertion Error is triggered. The Assertion Error will relay a message that will state that the type of drone that is queried by the string representing a unit type, does not exist in the config file that contains the definitions of each unit type.

The list itself can not contain multiple tuples with the same first element. This is due to the fact that the first element of the tuple is an indicator for the purpose of determining which unit type has a quantity that is assigned to by the second element in the tuple.

With that in mind, if there were to be two instances of a tuple with the same first element in the same group list, then the second occurrence would overwrite the original first tuple that indicates the quantity of that unit type.

The reference used for checking the validity of the first element in the tuple is called “unit_names”. This variable has a type of a dictionary with keys consisting of each unit type that exists in the python server. The assertion checks that the unit type that is trying to be referenced by the first element in the tuple exists as a key rather than a value.

```
        self.unit_names = {}
        for in_type in self.unit_dat['units']:
```

Figure 43: Code that creates dictionary of unit names

The dictionary containing the unit names is initialized as an empty dictionary and then assigned by going through a loop of a JSON file that contains drone unit types. The file is referenced as “unit_dat” and contains a key with the string value “units”. Each unit type is extracted and initialized from the class method EvgUnitDefinition(), which contains 6 variables.

```
    unit_type = EvgUnitDefinition(
        name = in_type['Name'],
        health = in_type['Health'],
        damage = in_type['Damage'],
        speed = in_type['Speed'],
        control = in_type['Control'],
        cost = in_type['Cost']
    )
```

Figure 44: Code that creates unit type

The first variable is the drone unit type’s name and is a string data type. The second variable is the drone unit type’s health and is a string data type. The third variable is the drone unit type’s damage and is an integer data type. The fourth variable contains the value of the drone unit type’s speed and is an integer data type. The fifth variable contains the value of the control and is an integer data type. The last variable contains the value of the drone unit type’s cost and is an integer data type.

After development, the unit type was expanded to incorporate more fields such as, squad speed bonus, squad damage bonus, and an electromagnetic warfare mimicking ability called jamming. The reason for this is due to our new unit types that have unique abilities. The overall updated format looks like the following image.

```
unit_type = EvgUnitDefinition(
    name = in_type['Name'],
    health = in_type['Health'],
    damage = in_type['Damage'],
    speed = in_type['Speed'],
    speedbonus_controlled_ally = in_type['SpeedBonus_Controlled_Ally'],
    speedbonus_controlled_enemy = in_type['SpeedBonus_Controlled_Enemy'],
    jamming = in_type['Jamming'],
    commander_damage = in_type['Commander_Damage'],
    commander_speed = in_type['Commander_Speed'],
    recon = in_type['Recon'],
    control = in_type['Control'],
    cost = in_type['Cost']
)
```

Figure 45: Updated unit type fields

The bonuses labeled ‘speedbonus_controlled_...’ are for gaining bonuses while in allied or enemy territory. The jamming attribute is what constitutes an EMP attack to enemy drones. The effect of which decreases movement speed of enemy drones but can be changed by future groups to do more. The commander damage and speed are attributes that provide a change in damage or speed to an entire squad as a commander drone. The last change, recon is designated for the recon drone type which maintains the functionality it had at the beginning of the project, as the recon unit was a unit type from a previous year’s project. The full explanation on these new attributes is located in the chapter discussing new unit attributes.

Refactoring to Accept JSON

Our team decided that it would be in our best interest to reformat our JSON files that contain the drone loadouts when called to be inputted into the original code. This is because we believed that we should refrain from undoing as much of the work that already exists prior to our involvement.

This will ensure that the project will still work as there will be no changes to the core components of what makes the project run. The process of reformatting the list of strings was rather smooth thanks to the built-in libraries in python. The high-level process of conversion acts as shown in the following diagram.

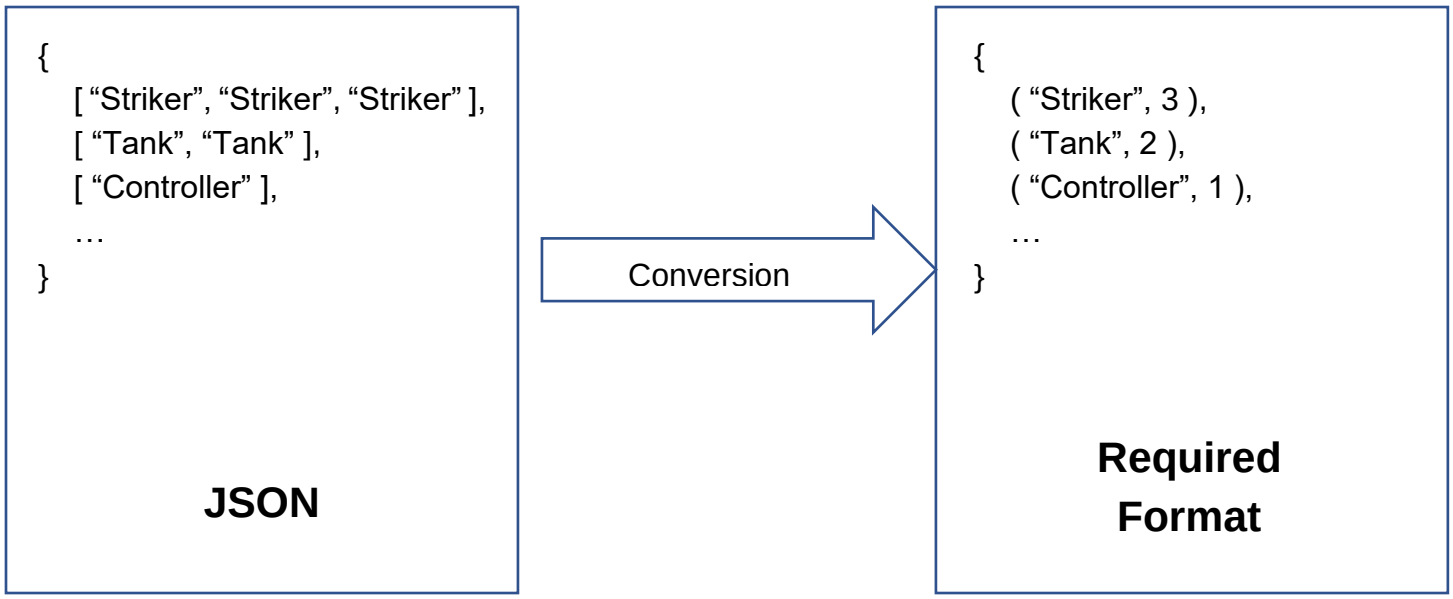


Figure 46: JSON conversion method

Here the JSON contains three groups, each group is composed of drones of the same unit type. The first group is a group that is primarily made up of strikers, with a list that has a length of 3. The second group is a group that is primarily made up of tanks, with a list that has a length of 2. The third group is a group that is primarily made up of a single controller, with a list that has a length of 1.

The required format is formatted differently than the JSON, however the required information that makes up the required format can still be acquired from the JSON file. When accessing each list in the JSON, the drone type can be accessed through accessing the content of the list. The number of drones that are existent in the specific group can also be accessed through the length of the list containing the list of strings.

The list itself represents the list of groups of drones that are existent for a given player. The elements of the list have a value of a tuple that contains two elements inside of it.

Input JSON

For the JSON created by the unit creator, there are lists that constitute a group. Each list contains strings in each element and represents what type of drone currently resides in the group. The length of the list determines the quantity of drones that are currently residing in that group. There is currently a use of four types of drones, a tank, a striker, a recon, and a controller. Ergo, a sample input of a loadout that utilizes all types can be represented as shown in the following excerpt of code.

```
{
  ["striker", "striker", "striker", "striker", "striker"],
  ["tank", "tank", "tank"],
  ["recon", "recon"],
  ["controller", "controller", "controller"]
}
```

Figure 47: Sample input loadout

Conversion Method

Before the actual conversion of the input JSON, the data stored must be iterated over. This is done using the following loop code in python:

```
for i in range(len(loadout)):
```

Figure 48: Code snippet of iterating stored data

The variable “loadout” contains the value of the lists of strings that compose the JSON. Each individual list contains the group composition code that holds the information as a tuple. This is the information that is integral for the python server to understand and process for the game.

If the python server were to read the tuple formation that is existent in the context of the JSON, it would lead to an error. The python server is unable to read the drone group information as a list of strings depicting each individual drone that is existent in the context of the current group.

To properly convert the components of the JSON to a more readable manner for the python server, a short and complex method was used. The method that was implemented for converting the input JSON that holds the drone loadout was written as followed.

```
    group_units = loadout[i]
    unit_configs[i] = [(x,group_units.count(x)) for x in set(group_units)]
```

Figure 49: Code snippet of JSON conversion

Each group of drones is cast as a set so that each drone type can be referenced only once. This is due to the fact that a set data type cannot hold more than one of the same element values. The set of drone unit types is iterated over for each element that it contains, and thus, each unique string inside the set can be placed in the first tuple of the dictionary. The number of occurrences that can be found by using the python method “count” to get the number of occurrences of that particular drone type inside of the original list of strings that represent a drone type.

Loadout Rule Handling

Included with the CreateJsonData.py are functions for error checking, CheckIfValidSquad() and CheckIfValidLoadout() are two functions included that take in information on a squad or loadout respectively and return a boolean value on the validity of the sent information. It was determined to create these functions for three specific uses.

- When loading in a loadout for use in game, these functions can be called to determine if the loadout is valid, and if not, a default loadout is used. These functions can also be utilized by GUIs created so that when invalid loadouts or squads are created, the GUI can recognize them.
- These functions are planned to be integrated with pulling data from a game rules config to allow rules to change in the future, such as determining to change the squad sizes, total drone amounts, or any other loadout rules. As a secondary goal of project Everglades is to create a fun game such that players can become engaged, increasing the range of available options to game setups is a way to boost playability. Therefore, these functions were determined to be a viable way to introduce a place where those rules would be gathered and checked.
- As there are groups of people creating AI for this game, if they wish to include to their AI the ability to decide their own loadout, these functions should be accessible to them. These functions provide a way to check validity for that AI's loadout, Whether it be through the AI creators using a GUI to create a static loadout for their AI, or for their AI to have these functions available to call when trying to generate its own winning loadout composition.

The current working requirements toward a valid loadout:

- 8 Units per squad (12 on last squad)
- 100 Units total. No more, no less.
- All units must be either defaults, presets or balanced custom units, based on the game mode selected
- JSON: Formatted correctly. Ensured if using provided functions
- 12 Squads in total.

Prototype Interfaces

To both increase the speed and accuracy of testing the loadout system implemented for saving and loading, interfaces were designed and created.

While the original prototype concept was created as a utility to creating loadouts for testing, it was also made to consider what would be needed for a human player of the game to create a loadout before playing the game on the Unreal side of gameplay. While not intended to be created with a large amount of aesthetic appeal, the idea was to create something that could be theoretically implemented into the Unreal side that is functional, and later could be replaced by a more aesthetically pleasing and themed menu should the expansion of the tool be desired.

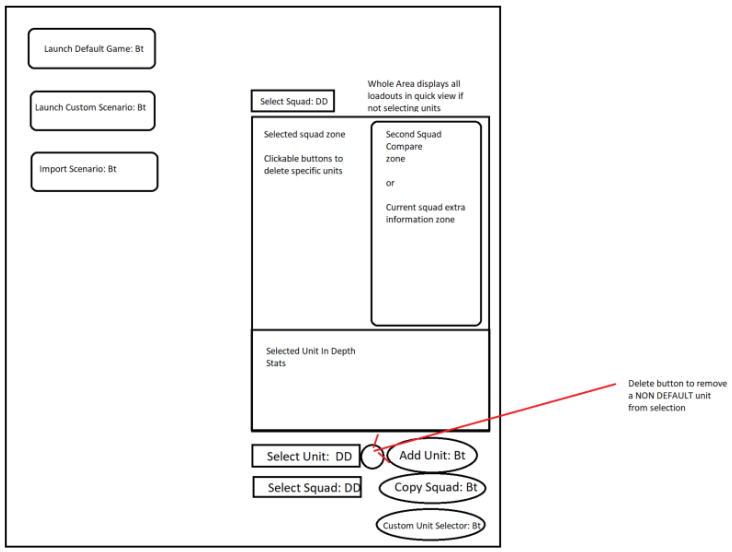


Figure 50: Prototype UI

An early prototype design, a pop-up menu that triggers when starting a new game via player to AI mode. The game does not start till a launch option is selected. This UI is to interface with the Python server on behalf of the player. The UI consists of three main option buttons; One which launches the game with default loadouts for both teams, one which launches a custom scenario to be played, and one which allows importing of custom scenarios. The importing of scenarios is a feature that is desired but not required. If the game is launched as default, then the game should have normal gameplay without custom loadouts.

To decide loadouts, this menu will let the user swap between groups they control and mix and match drones in each group to create a customized loadout experience. The player functionality will have ease of access commands like copying entire squads to

other squads and displaying current squad information for better decision making. The ability to export out these loadouts is desired to be added for increased usability.

As a stretch feature to expand upon playability, it was determined that when custom units would be added, there will be options to open an additional window to use a point buy system to create units. The workings of the unit creation system are lain out in its own section. With this stretch feature, multiple custom units will be added as premade extra options and become available for the user to choose from. Upon launching the game, there will be an option as to whether the units will be default, expanded, or custom. Where expanded contains the premade ones, and custom allows for both premade and custom created units. Both teams will share the choice made to maintain balance.

All of these features were fully integrated into the UI such that loadouts can be created, and new units created, before starting a game. This allows for easy setup of new scenarios for testing.

Developer GUI

Purpose for Creation

To facilitate the essential testing functions, with plans to expand and improve, an interim GUI was developed. The GUI allowed for quick and easy testing of the loading and saving systems and can be used to create custom loadouts without dealing with the JSON file directly.

Functionality

The GUI needed to create a 2-D array, where the first index represents the squad number of the unit, and the second index represents the number of the unit within the squad. The interface also had to be a function squad editor, adding unit names to the squads' lists.

Development

The team proposed the following GUI to outline the general look and functional of the interface.

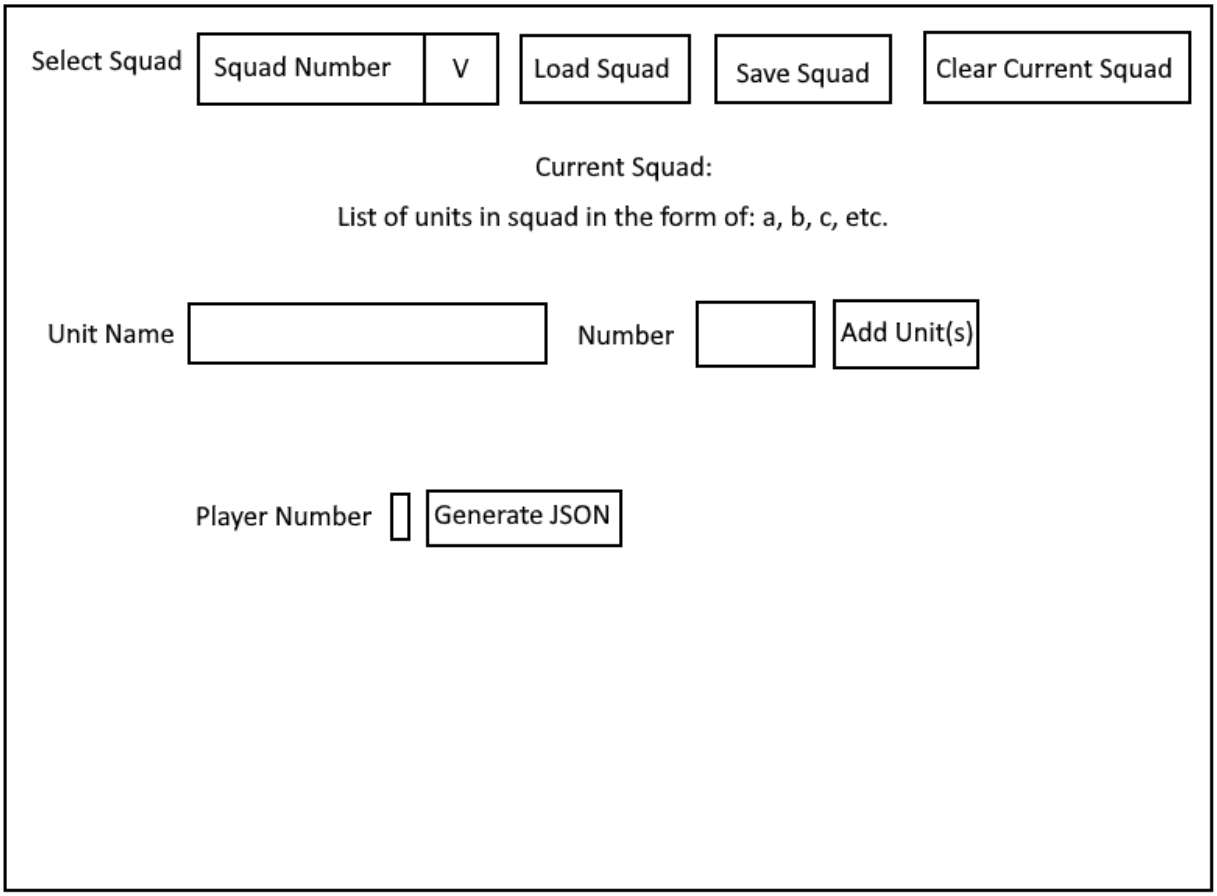


Figure 51: Initial Plan for the Squad Editor

The exporting function from this interface needed to call a Python function in other parts of the server's files. For ease of development given these circumstances, Python was chosen as the method of developing the interfaces. By extending the other Python file whose function needed to be called, or even just by extending the function that needed to be called, the solution to calling disparate code was as simple as adding one line to the otherwise independent code segment.

To keep the necessary time for research, installation, and learning low, the Python code was developed in an IDE of the same version as the Python server. The visualization package used was Tkinter, one that is both accessible and familiar to the team.

In Tkinter, objects are placed on a grid. Within the grid, frames are used to further relate items, with the window being the largest frame. Within frames, any column is as wide as the widest cell it contains, and any row is as tall as the tallest cell it contains. If objects themselves change width or height while the interface is up, the column or row expands to accommodate.

To keep a window that lines up objects in non-uniform or non-square ways, frames must be used. For example, without creating a frame within the window, it is impossible

to line up two objects over one wider object, as shown on the next page. The row and column numbers correspond to those you would give the object in Python, and the color the master they belong to (black text indicates the window, blue text indicates the newly created data frame).

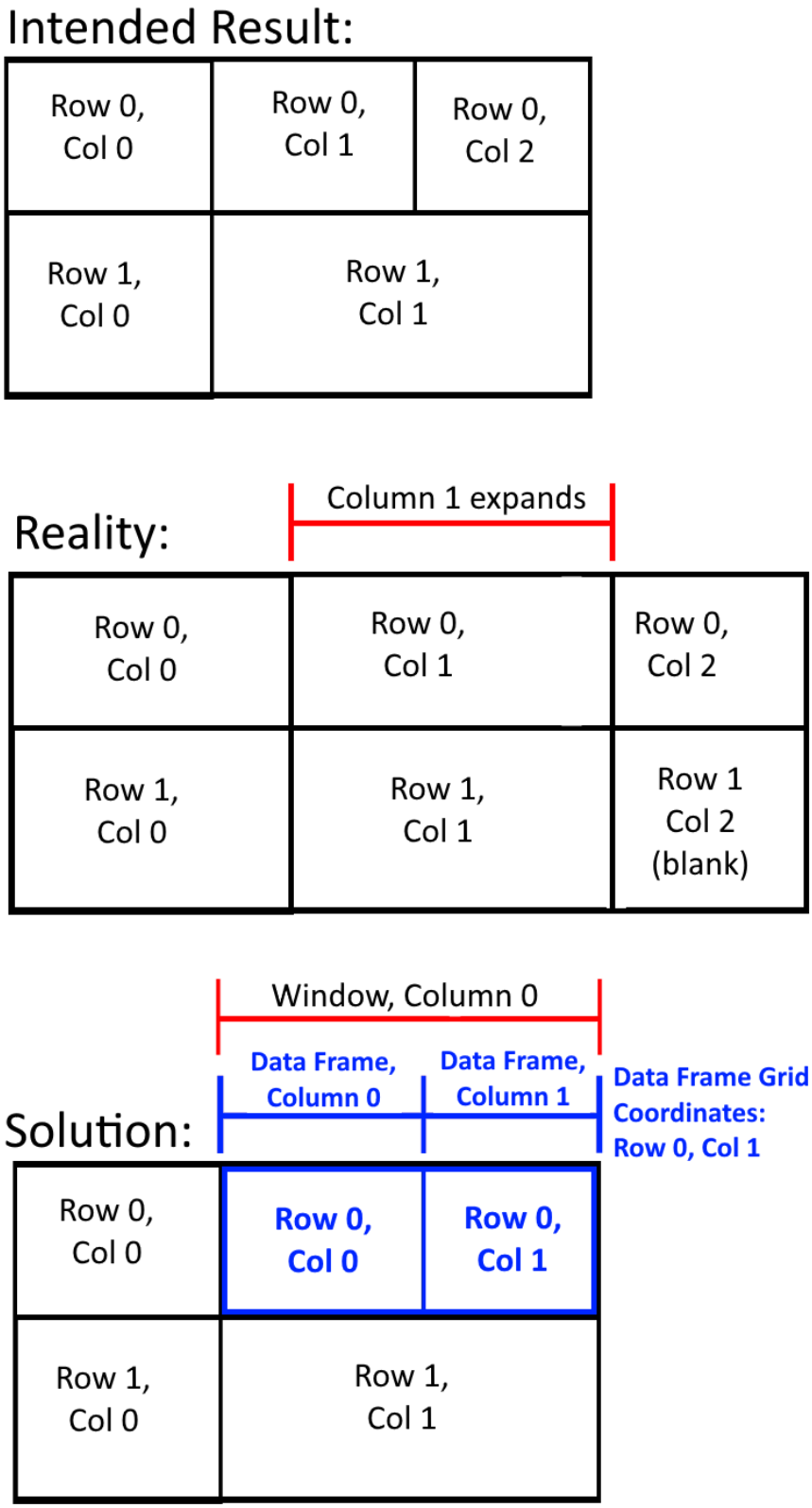


Figure 52: Using Data Frames to Circumvent Row or Column Sizing

Shown below is the initial plan for the interface, with grids drawn representing the data frames necessary to maintain the appearance initially proposed for it.

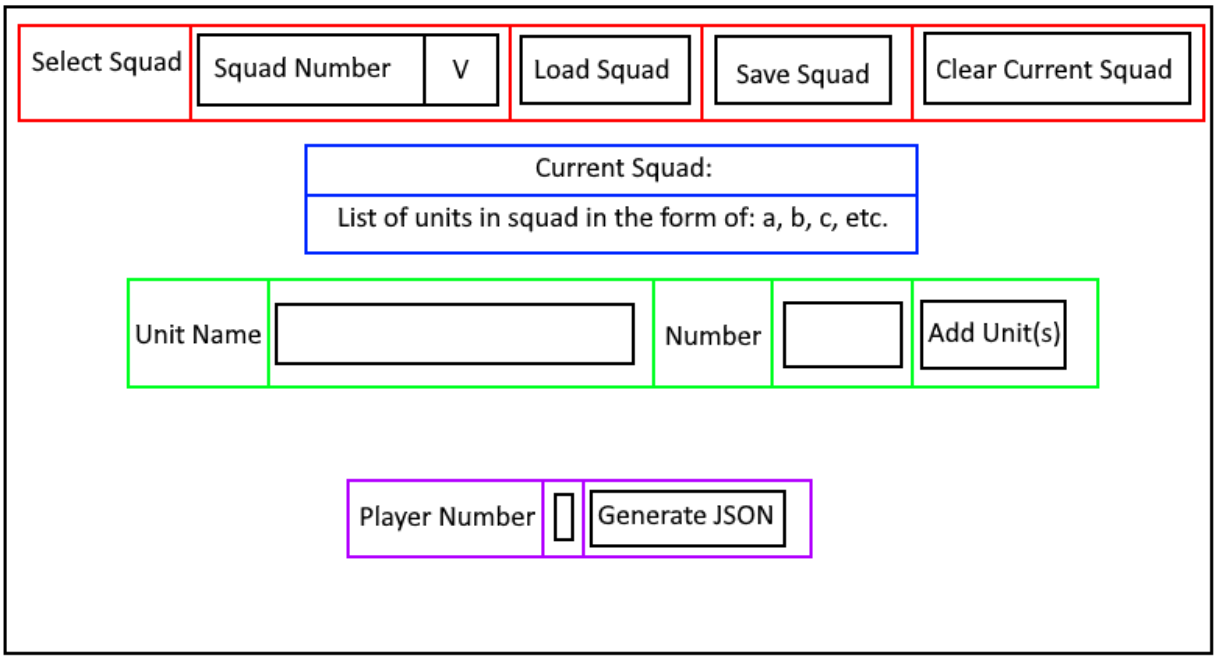


Figure 53: Plan for Squad Editor UI with Data Frame Grids Added

Initial Design

Pictured below is the version of the squad editor interface that resulted from the early prototype.

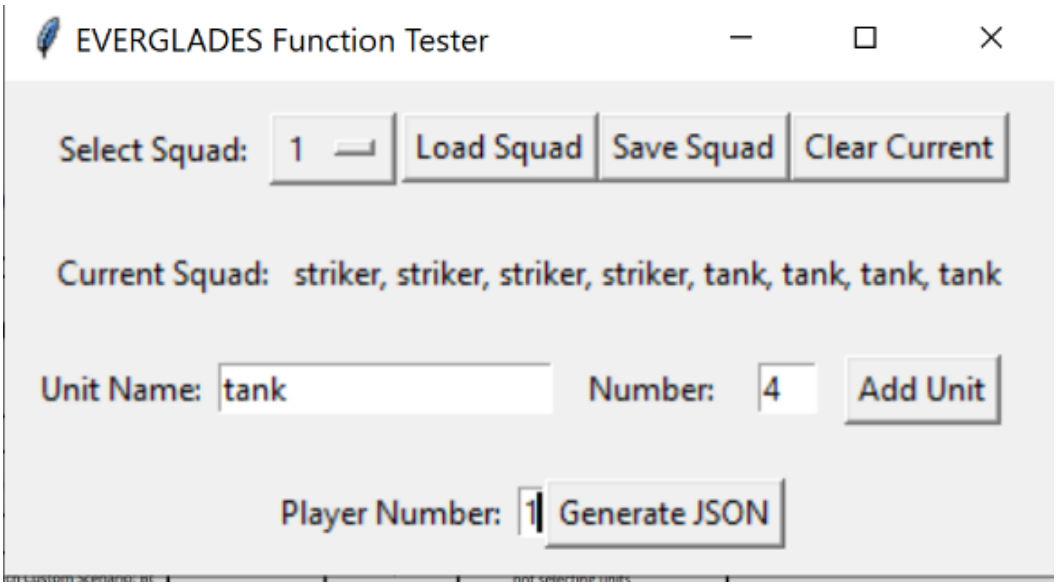


Figure 54 The First Version of the Squad Editor

The code stores a temporary list of units that it displays in the line of text next to the label “Current Squad:”. The buttons along the top of the list and the entry fields along the bottom list modify this temporary list, as follows:

- Loading a squad overwrites the temporary list of units with the list of units stored for the squad listed on the dropdown menu
- Saving a squad overwrites the list of units stored for the squad listed on the dropdown menu with the temporary list of units
- Clearing the current squad empties the temporary list of units
- Adding units appends units to the end of the temporary list of units
    - The units appended have the name entered in the entry field labeled “Unit Name”
    - The number of units appended is equal to the number entered in the field labeled “Number”

The button labeled “Generate JSON” uses the list of units saved for each squad to generate a 2-D array of length 12, where each primary index stores a squad’s list of units. After creating the 2-D array, it and the current value of the field labeled “Player Number” are passed to a function in a separate Python file that generates the JSON containing the squads for the player of the associated number.

Though the current implementation is functional, as development continues this interface will be refined to be more error-proof and more professional. This list of proposed changes before development concludes includes:

- Add a load button that allows the interface to take in the units in each squad presently

- Ensuring that the entry field for the unit's name is not empty
- Ensuring that the entry field for the number of units is an integer
- Updating the dropdown and UI to allow for a variable number of squads
    - Have the load button take in the number of squads currently
    - Add a button to create a new squad
    - Add a button to delete the currently selected squad
    - If a change is made to the list of squads, have the dropdown recalculate the list of options it has

Iterative Design

As the project progressed and we received both internal and external feedback, functions and visuals of the interface were iterated upon to make it more accessible and powerful to the end user.

The features that users often cited as being difficult to accomplish included:

- Removing a single unit type from a squad, but keeping the remaining unit types in the squad present
- Decreasing the number of a type of unit in the squad
- Changing the number of squads

This resulted in the second major design for the squad creator, show below.

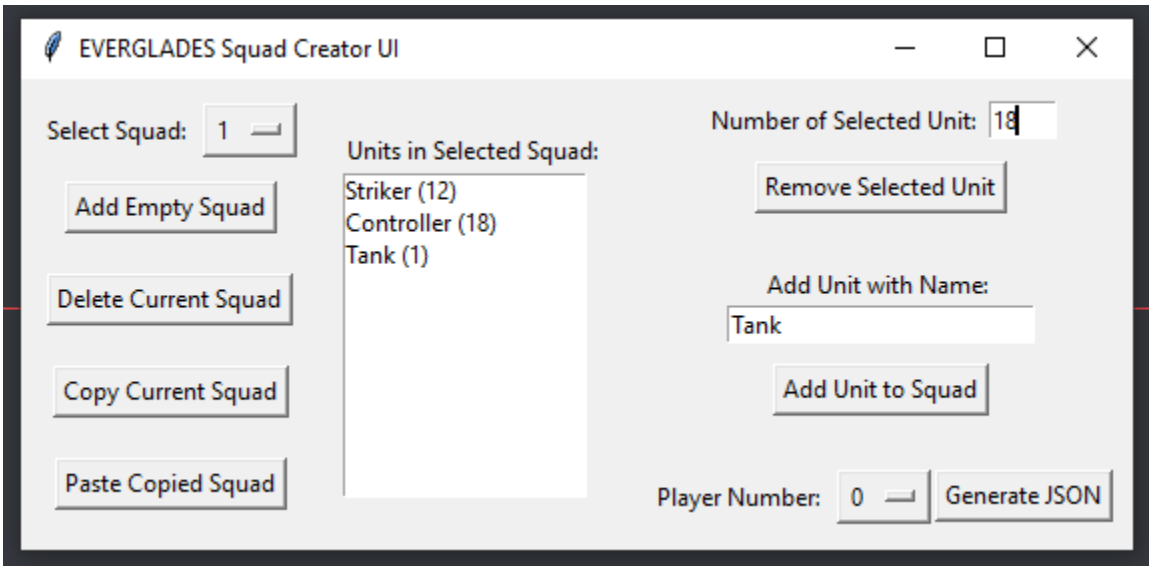


Figure 55: The second version of the squad creation UI

The TKinter widget used previously for listing squads was the basic Text element, manually separating the units by commas and parsing the values back when reading it to

write to the server's files. This version of the interface replaced the Text widget with the Listbox widget, which stores each line-seperated string as its own index in an array. This made reading and writing to and from the squads not only easier and less error-prone programming-wise but also helped to visually separate the different types of drones and their counts from one another.

In terms of functionally, the Listbox is an interactive element, and functions can be called when items in it are clicked. This allowed for the implementation of two important features that improved usability:

- The number of each unit type in the squad could be changed much more easily. By instead storing each squad as two arrays, one for each unit type and one for each type's amount, a concrete value for the amount of each unit could be loaded into a UI element, modified by the user there, then saved to the array. To this end, an entry form was created, which is set to the value of the currently selected unit's amount. Any change to the entry form is immediately saved to the array.
- Individual unit types could now be removed from the squad. This is effectively done by popping the associated indices in the aforementioned arrays and then updating the UI to account for the change.

Additionally, functionality was added to change the amount of squads from the initial 12. This was a planned feature of the server at the time, though it was not expected of us to implement.

Final Design

The final design for the squad creation UI was designed to quell any remaining issues with the interface. It is pictured below.

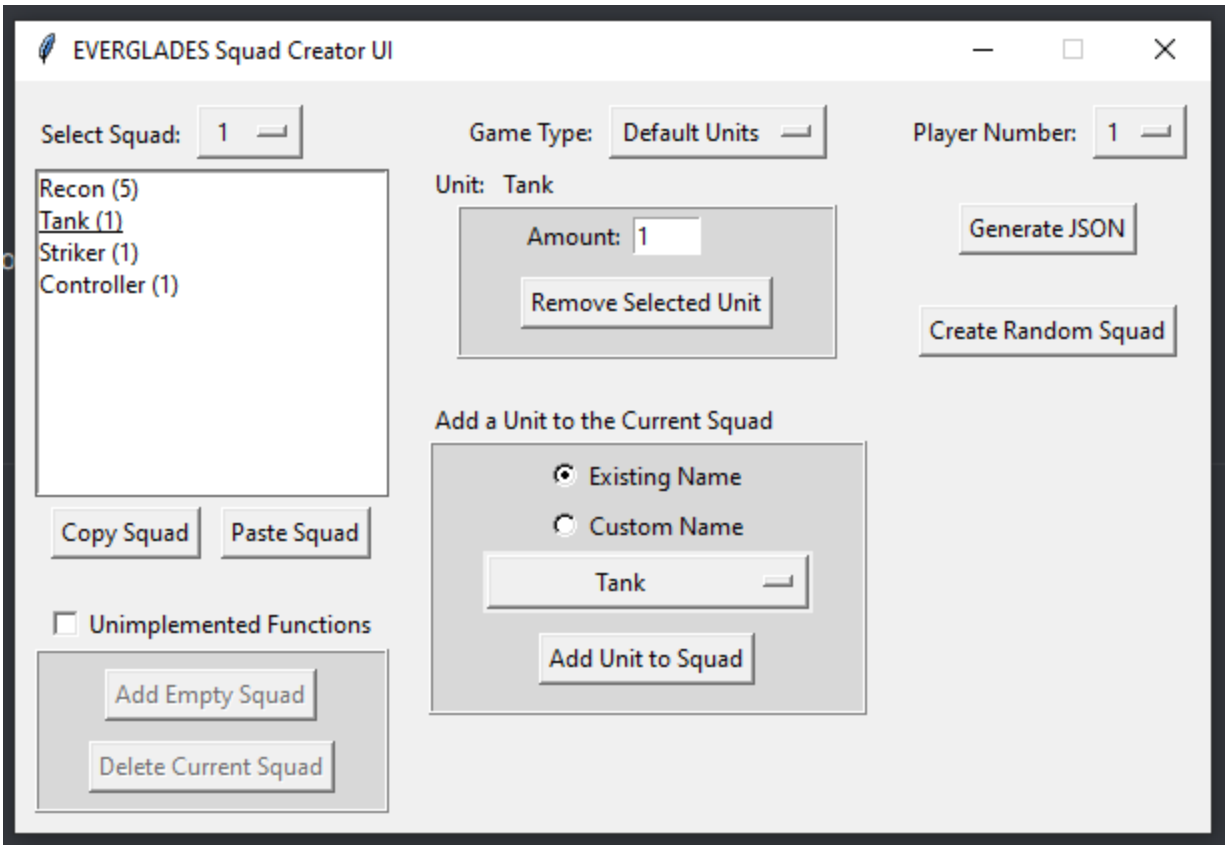


Figure 56: The final design of the Squad Creator UI

Aside from improving the visuals of the UI to be both more appealing and more readable, a final design was created to solve three major issues.

- Changing the number of squads to a number other than 12 was attempted and mostly possible given further-reaching changes to the server's codebase, but was not completed by the end of the semester. As a result, the buttons to modify must either be removed or the user must be warned not to use them presently. Since this project will continue to be worked on by groups in the future we opted for the latter, so that they may simply remove the warning message from the UIs code when it is implemented fully.
- Adding the name of a unit that isn't defined by the server will prevent the game from running. By only allowing the user to enter their own strings, we are making it more difficult for them to enter existing unit types into the game. Additionally, every unnecessary custom text entry is a chance for a misinput that halts progress in setting up the server. This design adds a radio button that changes the unit-adding process from the custom name entry to a dropdown menu populated with all possible unit types for the game type the user is in.

- A major oversight left untouched by the previous versions of the UI was that the structure of the server loads a different JSON depending on which type of units of the game is being played (default unit, unit presets, or custom units). Since there was no UI element to change it, the interface would always write to the same game type's JSON – whether it was correct or not. This was changed by adding a dropdown menu at the top that changes which game type's loadout the player will write to.

Additionally, a feature was added to generate a random squad. This is useful for testing a variety of different squad makeups very quickly.

Difficulties with Loadouts

The previous implementation of loadouts was a hardcoded list determining the exact layout of each squad. After unraveling this, it was understood that multiple foundational problem resulted. The main areas were in squad movement and combat.

```
self.unit_config = {
    0: [('controller',1), ('striker', 4), ('recon', 1)],# 6
    1: [('controller',2), ('striker', 2), ('tank', 3), ('recon', 2)],# 15
    2: [('tank', 5)],# 20
    3: [('controller', 2), ('tank', 1), ('recon', 3)],# 26
    4: [('striker', 5), ('recon', 5)],# 36
    5: [('controller', 3), ('striker', 2), ('recon', 1)],# 42
    6: [('striker', 3), ('recon', 1)],# 46
    7: [('controller', 1), ('striker', 2), ('tank', 2), ('recon', 1)],# 52
    8: [('controller', 1), ('recon', 2)],# 55
    9: [('controller', 1), ('striker', 3), ('recon', 2)],# 61
    10: [('striker', 9)],# 70
    11: [('controller', 10), ('striker', 8), ('tank', 2), ('recon', 10)],# 100
}
```

Figure 57: Hardcoded loadout from previous project

For instance, as mentioned one of the problems lies in the movement system of a group. Previously a group of drones was seen as a single entity with a set movement speed. With multiple units of varying speeds, the problem arises that there is no infrastructure in place to determine what speed the group should use. The temporary measure to implement was to reduce the group to the speed of the lowest unit in the squad. This will significantly impact the usefulness of speedy units when in a squad with slow units. After the loadout system was fully implemented, we evaluated this decision and decided to leave this in place as the system for movement calculation. A future group may find the desire to expand the movement systems such that squads can break apart

into subsections. Toggling this on and off would allow for groups to either stay together or break apart to have the speedy units travel to a position first before the remaining groups “reinforcements” arrive. This could be useful strategy wise to send Controller units to positions faster when immediate enemy drone contests are not expected.

Recon Drones

Special consideration had to be taken with respect to the recon drones’ unique implementation. Original requirements had only Strikers, Tanks and Controllers being maintained as units, while Recon drones were left as a previous groups addition with no intent of being carried over to future projects. In the interest of continuing the previous year’s work and expanding electronic communication, it was deemed to attempt to bring the Recon drone over to the updated project.

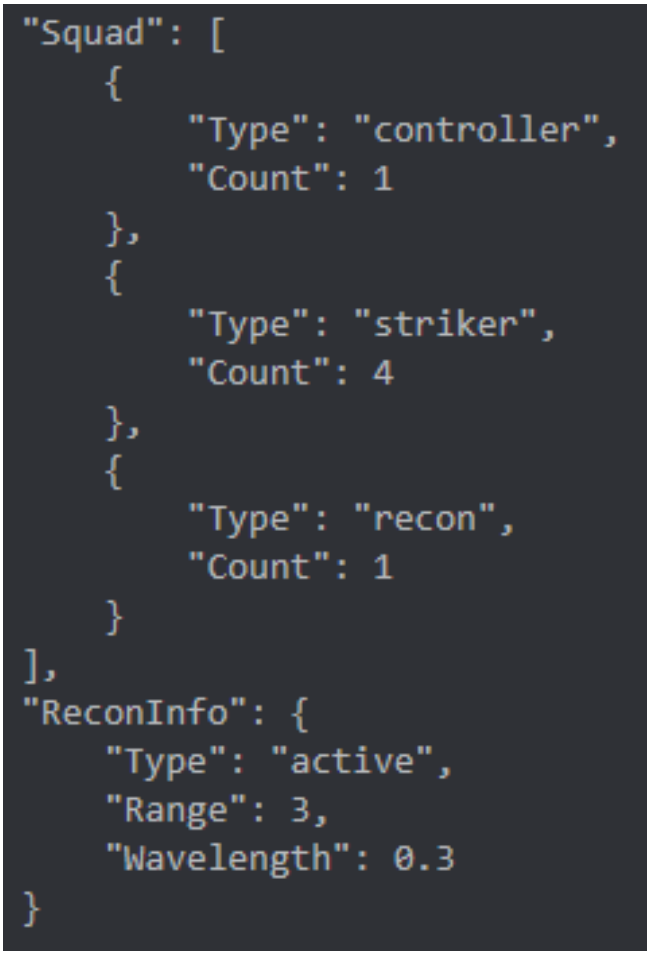
Due to requiring certain information for each squad to be able to utilize a recon drone, the recon drones were implemented partially with the foundations in place for a future group to finish integration. While some aspects of the Recon drone remain, selecting frequencies for electromagnetic spectrums, determining scanning range and determining active or passive mode are all set to defaults with no ability to change. Should the Recon drone wish to be later refurbished to the original condition back when all these values were hardcoded in the loadouts, some groundwork will have already been done for this.

The Recon drones did pose an interesting dilemma, as in the unit creator goal, all units are structured in a way that each unit has a set of attributes that define how they work. Meanwhile a recon drone is based on a squad wide attribute and generally incompatible with being a single attribute. It was decided to maintain the original approach to the recon drone despite the attribute system posed in the unit creator section, for several reasons. The first reason is that the current implementation works, and if it can be maintained then there is less chance of unintentionally adding bugs to it, as well as the benefit of not expending time and effort to an unnecessary task. The second reason lies in the relatively low benefit a squad can get if they have two drones with differing abilities. A squad with an active and passive drone actively undermines the passive drone’s greatest strength of remaining undetected, making little strategy for mixing active and passive drones. As far as the other fields go, there are almost no real benefits to having two different recon drone frequencies.

Below are two figures representing the original hardcoded information on sensor configurations, as well as a theoretical representation on how squads could retain recon drone information.

```
self.sensor_config = {
    0: ['active', 3, '0.38'],
    1: ['active', 3, '0.45'],
    3: ['active', 3, '1.20'],
    4: ['active', 2, '2.30'],
    5: ['active', 3, '0.40'],
    6: ['active', 3, '2.20'],
    7: ['active', 1, '1.13'],
    8: ['active', 3, '1.08'],
    9: ['active', 3, '0.77'],
    11: ['passive', 3]
}
```

Figure 58: Hardcoded sensor values that worked with original hardcoded loadouts.



```
"Squad": [
    {
        "Type": "controller",
        "Count": 1
    },
    {
        "Type": "striker",
        "Count": 4
    },
    {
        "Type": "recon",
        "Count": 1
    }
],
"ReconInfo": {
    "Type": "active",
    "Range": 3,
    "Wavelength": 0.3
}
```

Figure 59: Theoretical representation of how the JSON could retain information on Recon drones.

Improved Playability

Unit Creator

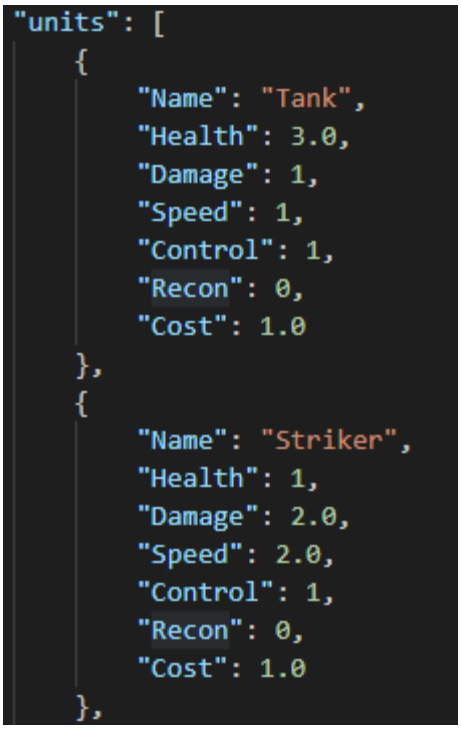
Being able to create custom units is often an appeal to many video games, as often players enjoy customization and creating new and interesting units. Due to this, the idea emerged about creating a unit creator for this project such that new units could be added to the game in a more modular system that could be easily used by new developers or players of the game. As one of the goals of the project is Improved Playability, a goal towards creating a unit creator was set to reach the goal. As well as being a tool for increased enjoyment, a unit creator also lends an interesting ability to create scenarios for AI in which to see how they perform. For instance, asking the question of how would an AI perform if fast point capturing units were in the game, or if their team had access to medic like units, or even if their team had access to only defensive units, how would that AI's strategy adapt or hold up? These questions are research questions that fall under the scope of why this game was created, as this game is a teaching device in one of its main purposes.

Making the Unit Creator

Units had to be redefined to determine what is necessary to understand about a unit to generate the information for one. To make a modular system that would allow easier addition of units, units were determined to contain the following properties:

- Name: This determines what the unit will be referred to in the game data.
- List of Attributes: To create a system of balances, a new system called Attributes, was created. See the Attributes section after this section for more information.
- Balance Level: A value determining if this unit is considered a Default (striker, tank, controller, recon), Extended (A preset approved to be considered 'balanced'), or Custom (Any custom preset made by a player)

Note: This is not how the project previously stored information on the units. The project stored the information as shown in UnitDefinitions.py. UnitDefinitions.py will maintain its format, with additional fields added to store information for gameplay when new features are added. There was a converter created that converts the unit preset JSON information into the games UnitDefinition.py format when the game is launched. This was decided as to not unnecessarily change a working format, and because the new format for storing information is more readable and manipulatable.



```
"units": [
    {
        "Name": "Tank",
        "Health": 3.0,
        "Damage": 1,
        "Speed": 1,
        "Control": 1,
        "Recon": 0,
        "Cost": 1.0
    },
    {
        "Name": "Striker",
        "Health": 1,
        "Damage": 2.0,
        "Speed": 2.0,
        "Control": 1,
        "Recon": 0,
        "Cost": 1.0
    },
```


Figure 60: JSON representation of units in UnitDefinitions.py

A major decision was made to create an attributes system. The idea behind the attribute system is that each unit is comprised of certain attributes, or ‘upgrades’. These attributes enhance or add features upon how the unit operates. For instance, a normal ‘Tank’ as shown above has 3 times more health then normal, as the default value for health is 1. This correlates to the “Enhanced Health” attribute being on the Tank twice. A normal ‘Striker’ as also shown above, has double damage and speed. This correlates to the “Enhanced Speed” and “Enhanced Damage” attribute being on the Striker. Using this method, given that each of those attributes consist of half of a units point cost, a new unit could be created by matching attributes around, shown below. In this new unit, the ‘Mobile Heavy’, the unit holds the attributes of “Enhanced Speed” and “Enhanced Health”.

```
{
    "Name": "Mobile Heavy",
    "Health": 2.0,
    "Damage": 1,
    "Speed": 2.0,
    "Control": 1,
    "Recon": 0,
    "Cost": 1.0
}
```


Figure 61: JSON representation of a new unit

While there is a small range of mixing that can be done amongst the base attributes created for the default units, as new attributes are added, new possibilities for units are created. This approach was seen as highly advantageous toward playability of the game and towards making units. The three approaches considered were adding new units, adding attributes to create units, allowing full creation.

- Adding New Units: This was an appealing option, simply adding in new units to be used. However, this lacked the creation aspect for players to utilize. As creating units is a fun endeavor for some, it was decided to make sure creation was an option for players
- Full Creation: A tempting option would be to give players the ability to customize units without any bounds. The problem here is while It may add more enjoyment then the attribute system in some regards, balance is lost. As this is a game meant to teach how AI works and to have AI come up with strategies, creating unbalanced and unfair units would quickly remove the core aspects of the game.
- Attribute Assignment: Creating new attributes and allowing players to pull these attributes onto new units was deemed the best option. Doing this maintains the creation aspect while allowing for balance to be maintained. As more attributes are added in, more options become available. For instance, as opposed to adding in a medic unit that has higher speed and the ability to heal, adding in a medic attribute allows for a preset to be made for a medic using “Enhanced Speed” and “Repair Tool” attributes. At the same time, a player can create a custom unit such as “Combat Medic” and give it “Enhanced Damage” and “Repair Tool”. The units that can result are of a greater amount then if the adding new units method was used, while the balance is maintained as opposed to the full creation aspect.

The format chosen maintains an easy to read and understandable storage of what units have what properties. As stated previously, the UnitDefinitions file is unchanged, rather updated at game start to reflect what units are based on the stored JSON information on the attributes they possess. Note: balance level is not stored in the file, rather the balance level determined what file data is stored in. This is for future file protection ability.

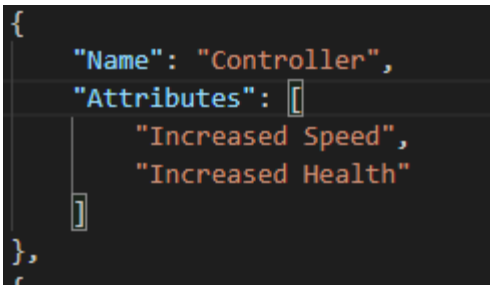
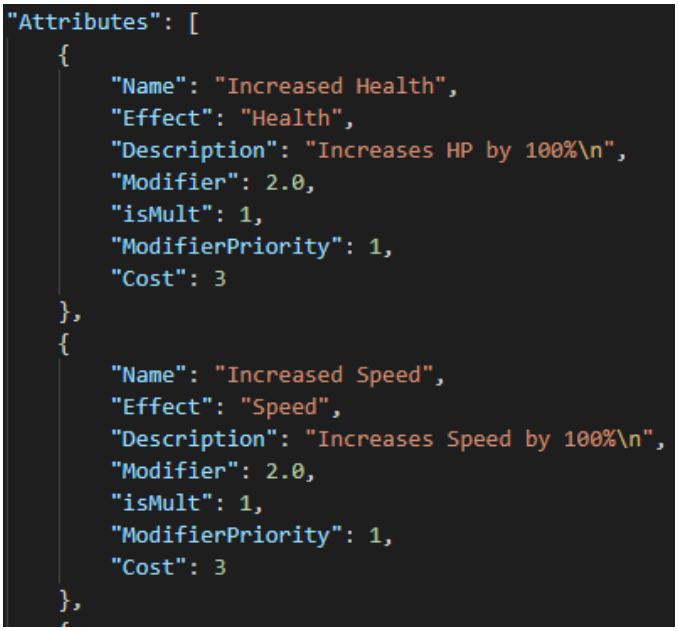


Figure 62: JSON storage of a unit based on attributes

An attribute is defined by the following properties:

- Name: The designation for what the attribute is referred to as.
- Description: A description on what the attribute does.
- Effect: The name of the variable to be modified in the UnitDefinitions.json file. For instance, “Enhanced Health” would use ‘health’, as it modifies the health attribute.
- Modifier: The multiplier/addition to be applied to the chosen variable.
- isMult: A Boolean true or false (1 or 0) on whether the modifier is a multiplier (true), or an additive (false).
- Cost: The point cost of the attribute, used to apply balance. Currently the units are considered to be balanced if their point total is 6. Game settings will determine if overpowered or underpowered drones are allowed onto the battlefield.

As shown below, this format allows attributes to be easily read, understood, and even manipulated straight from the file. This also means if an attribute is adjusted for being too weak or powerful, upon loading the game, all units that have the changed attribute will have the new balances loaded in alongside.



```
"Attributes": [
    {
        "Name": "Increased Health",
        "Effect": "Health",
        "Description": "Increases HP by 100%\n",
        "Modifier": 2.0,
        "isMult": 1,
        "ModifierPriority": 1,
        "Cost": 3
    },
    {
        "Name": "Increased Speed",
        "Effect": "Speed",
        "Description": "Increases Speed by 100%\n",
        "Modifier": 2.0,
        "isMult": 1,
        "ModifierPriority": 1,
        "Cost": 3
    },
```

Figure 63: JSON representation of two attributes

In the original design there was a value named ‘modifierPriority’ as seen in the image above, which is no longer used. The idea behind the field was as follows:

- Priority: An integer designation greater than 0 that determines when the modifier should be added. If multiple attributes effect a given variable, the attributes will be calculated in the following order: Priority 1 Addition, Priority 1 Multiplication, Priority 2 Addition, Priority 2 Multiplication, Priority 3 Addition…

Using this priority method, the game would understand in which order to apply multiple attributes that apply the same effect as order of operations between additions and multiplications can change the final value. Using this field was ultimately scrapped as the quantity of attributes and complexity of the game in general did not require this system. Should the game evolve in the future to use multiple modifier enhancements such as a situation where multiple attributes effect health, special slots in the squad effect health, items can be equipped that effect health, or other methods that effect health, then in this situation knowing whether a +.5 health goes at base health or final health can be very important for balance if there is a multiplication involved. Due to this, the field has been left in with framework in place to allow for a future team to pick up the functionality and add it to suit changing needs.

Unit Attributes

As part of the increased playability goal, adding in new units to the game would add more depth to the game. The following attributes have been successfully added to the game:

- Militia: Gain an additional 50% movement speed to base movement while traveling from a node controlled by the player. This attribute costs 1 of the 6 points and therefore 3 points grants 150% additional movement speed which is greater then the 100% movement speed the base enhanced speed attribute gives. The catch here is that the speed Is only better while moving from controlled nodes, making moving in your own territory as reinforcements more viable then offensively pushing through enemy territory.
- Guerrilla: Gain an additional 50% movement speed to base movement while traveling from a node that is not controlled by the player, which can either be enemy controlled or neutral. This attribute costs 1 of the 6 points and therefore 3 points grants 150% additional movement speed which is greater then the 100% movement speed the base enhanced speed attribute gives. The catch here is that the speed Is only better while moving on enemy or neutral territory and therefore serves better as an aggressive assault squad to hunt down enemies or rush the enemy base.
- Jammer: Units within a specific game configured radius, will experience electronic warfare tactics, reducing movement speed between nodes by multiplying the current movement speed by X^Y where X is a game configured multiplier between 0 and 1, and Y is the number of jammers currently jamming in range. Due to this formula, additional jammers have diminishing returns on effectiveness but still make an impact.
- Assault Commander: While a unit with this attribute is present in a squad, all units gain a 50% multiplier to total damage dealt. This is a 6 cost attribute with a large impact, yet has the major weakness of taking up all attribute cost and being a prime target for enemy targeting. A unit can only benefit from one assault commander at a time.
- Speed Commander: While a unit with this attribute is present in a squad, all units gain a 50% bonus to total movement speed. This is a 6 cost attribute with a large impact, yet has the major weakness of taking up all attribute cost and being a prime target for enemy targeting. A unit can only benefit from one speed commander at a time.

Militia and Guerrilla were added to allow a squad to specialize on defensive or offensive movement, while also being a 1 cost attribute to be more freely used in large attribute low cost builds. The commander units were added to showcase the importance of strategic targeting in drone battles where a single unit is a larger threat then the rest of the units and therefore should be targeted first. The jammer unit was added to lay groundwork for the electronic communications stretch goal, showing the effects offensive electronic warfare can have on a groups ability to communicate and move effectively.

Unimplemented Ideas

With the addition of more attributes, units can be created with more unique skillsets. The following are brief ideas on other attributes that could not be completed during this project but were thought to add to the games complexity and strategy if implemented. Future groups may consider adding some of these attributes to strengthen the games strategy and enjoyment potential.

- Armored: Unit gains a flat percentage damage reduction. Would compound with health for higher effective health. Cost 3.
- Armor Piercing: Attacks made by this unit would bypass and remove the armored attribute from any armored unit hit, providing a direct counter to the ability meant to make a single unit with armor piercing equally matched against a single unit with armored, while allowing the armor piercing unit to have 2 greater attribute points to allot. Cost 1.
- Fortification: Gain a stacking armor bonus the longer this unit remains on the same node, to a maximum stacked value. A fully fortified unit should have a higher effective health then a unit with enhanced health. Cost 3.
- Entrenchment: Gain a stacking damage bonus the longer this unit remains on the same node, to a maximum stacked value. A fully entrenched unit should have a higher effective damage then a unit with enhanced damage. Cost 3.
- Quick Deploy: Gain a stacking bonus to speed for each round occupying the same node. When leaving a node this bonus begins diminishing for every new node entered. This makes a fast reinforcement drone that can quickly move short distances to join a contested node. Cost 1.
- Explosive Rounds: Every attack dealt to an enemy unit causes splash damage to all other units in that squad, dealing a portion of the damage dealt. This allows for strong group damaging while is less effective versus small groups. Cost 3.
- Shrapnel Rounds: Every attack dealt to an enemy unit causes two random units in the same squad to take a portion of the damage dealt. Additionally any damage dealt to the main target past the damage needed to destroy, is added to the damage of the two random units. Cost 3.
- Repair Tool: When unmoving in a node and not in combat, this unit provides a small squad wide health regeneration. Cost 3.
- Self-Repair: This unit regains a small amount of health each round, even during combat. Cost 1.

- Protector: This unit can transfer damage against a single allied unit each round to themselves, protecting that unit from damage. Cannot protect another unit with the protector attribute. Cost 2.
- Sniper: This unit can assist in combat from 1 node away. Cost 3.
- Leader: All units in squad gain a bonus to multiple stats such as health, damage and speed. This unit would likely become a priority target by enemies. Cost 6.
- Training Officer: Provides a bonus to the squad similar to the leader, however the bonus increases every turn to a maximum halfway through the game. This is meant to be a more powerful leader that is less effective in early game but more effective late game. Cost 6.
- Adaptable/Decaying: This unit gain a stacking bonus to stats over the course of the game/ starts with a bonus to stats that decays over time. Cost 2
- Disruptor: Nodes within a specified range do not gain points for the enemy team. Allows the nullification of point collection from a distance to try and force behavior. Cost 6.
- Commander: This drone provides a large bonus to an entire squad, however there can only be one in a game, and if the commander is destroyed, the enemy team instantly captures the commanders teams base. This would add a new win condition to consider. Cost 6.

Interfaces

Purpose for Creation

As we began our playability improvements, a stretch goal included an interface for creating attributes for units and for creating new types of units. This interface would be viewable outside the Unreal Engine application, with the purpose of making the tools to modify aspects of the tool accessible, easy to use, and designed to look professional.

This application served a variety of purposes:

- It is a powerful development tool, that allows us to create a multitude of unit types in a short amount of time. As a result, having it before the second half of development gives us a longer period of time to develop unit types presently, so time will not be dedicated to it over future developments.
- It allowed for the testing of surrounding code: having concrete output of sizes significantly larger than testing data might consist of allowed us to test that programs that dealt with the output of these interfaces.

    - This point is especially pertinent, as we found errors in surrounding code that causes data to be processed incorrectly. Discovering and debugging earlier likely saved a significant amount of time we would have spent making sure that data transmitted correctly from Unreal and to Python.
- It allowed us to make use of a prototype interface that would share a design with the final layout. Through the use of these programs, we can learn what does and does not work well with these designs, and improve them for the final, more clearly ingrained final interface.

Functionality

Two interfaces were created for the unit creator feature, one for creating units and one for creating attributes. For unit creation, the interface allows the user to select and add various attributes to the new unit and see the description on each. This interface, or a similar improvement made in the future, would be available to the user, likely through the loadout creation screen.

The attribute creator loads the list of attributes from the server's files upon clicking the load button and populates a selection field with them. Once loaded, the list of attributes could be changed by pressing the buttons associated with creating a new attribute or deleting the selected attribute. The list of attributes can be saved by clicking the save button. Entry fields exist for the user to enter and edit various information about the attributes:

- Name
- Description
- Its effected attribute
- The value of the impact on the effected value
- The function of the impact the modifier applies
    - If "Multiplicative", the existing value is multiplied by the modifier's value
    - If "Static Value", the existing value is overwritten by the modifier's value
- The cost added to the unit for having this attribute

The unit creator loads both the list of units (and all of their associated information) along with the list of attributes from the server's files upon clicking the load button. The list of units and the list of attributes each populate different selection fields, and contain the following information:

- Units' Information
    - Name
    - Currently applied attributes

- Attributes' Information
    - Name
    - Description
    - Cost

The list of units could be changed by pressing the buttons associated with creating a new attribute or deleting the selected attribute. Creating a new unit adds a unit to the end of the list of units with the name currently entered in the "Name" entry field. Changing the currently selected unit causes the total cost of attributes on the unit is recalculated. The list of units can be saved by clicking the save button.

By selecting an attribute from the list, its description automatically populates the description box. If a selection is made in the list of all attributes and the button to add an attribute is pressed, the selected attribute is applied to the currently selected unit. If a selection is made in the list of the currently selected unit's attributes and the button to remove an attribute is pressed, the selected attribute is no longer applied to the currently selected unit. Upon an attribute being added or removed to a unit, the total cost of attributes on the unit is recalculated.

Development

The team proposed interfaces of the following designs, labeled Figure 6 for the attribute creator and Figure 7 for the unit editor:

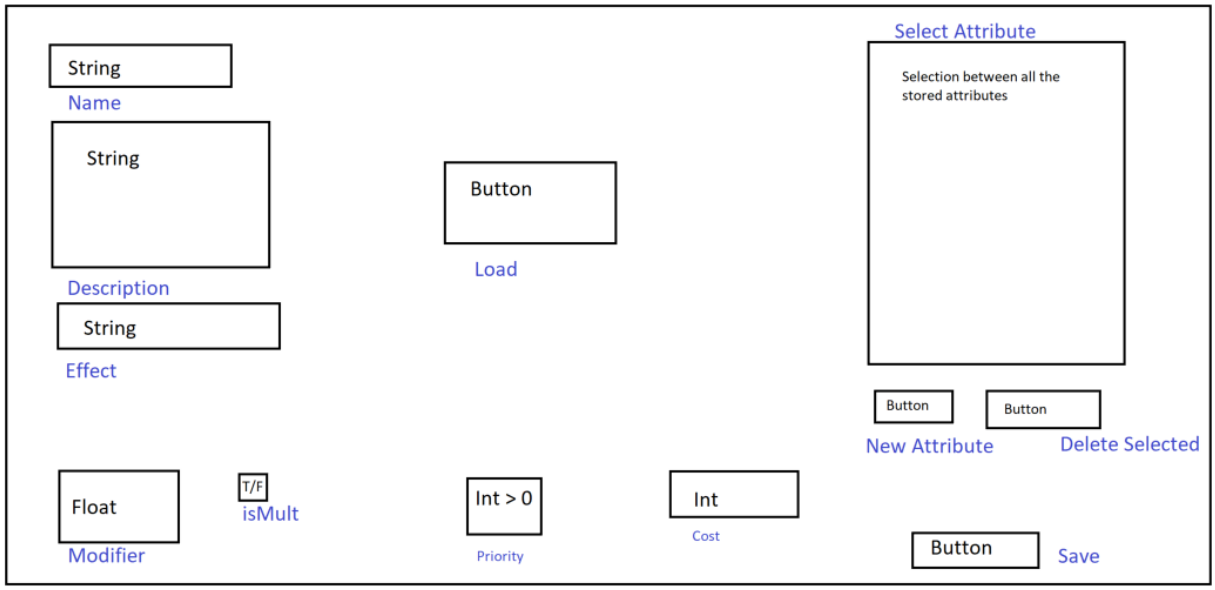


Figure 64: Proposed Attribute Creator UI

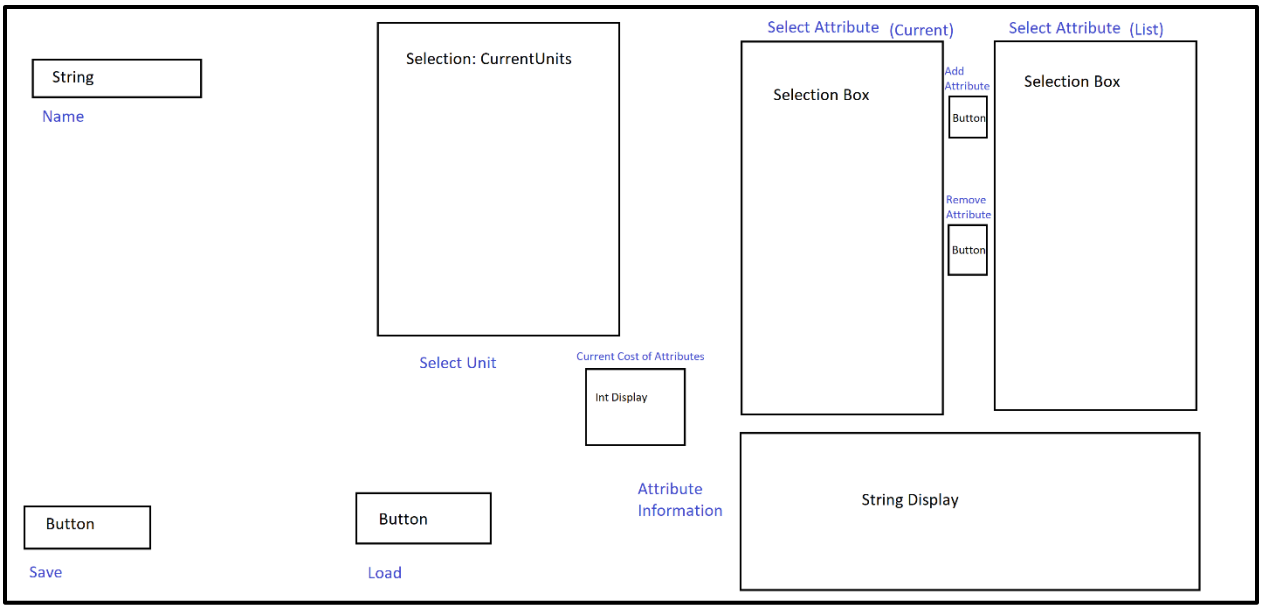


Figure 65: Proposed Unit Creator

The exporting functions from these interfaces needed to call Python functions in other parts of the server's files. For ease of development given these circumstances, Python was chosen as the method of developing the interfaces. By extending the other Python files whose functions needed to be called, or even just by extending the functions that needed to be called, the solution to calling disparate code was as simple as adding one line to the otherwise independent code segment.

To keep the necessary time for research, installation, and learning low, the Python code was developed in an IDE of the same version as the Python server. The visualization package used was Tkinter, one that is both accessible and familiar to the team.

In Tkinter, objects are placed on a grid. Within the grid, frames are used to further relate items, with the window being the largest frame. Within frames, any column is as wide as the widest cell it contains, and any row is as tall as the tallest cell it contains. If objects themselves change width or height while the interface is up, the column or row expands to accommodate. Knowing that the objects in the user interface would have to conform to a rather rigid grid, they were redesigned to better fit into one. Additionally, aspects were revised or repositioned as follows:

- For ease of use:
  - The save button was placed next to the load button
  - Deleting units from the list was added to the attribute editor
    - Because units could be added to the list through the name entry in the top-left corner, it was important that erroneously added units be able to be removed

- The list of attributes in the unit creator and the list of units in the attribute creator were changed to dropdown boxes
  - Because these selections are made much less frequently than the options that remained selection boxes, they were made to dropdowns to reduce the amount of UI space they take up while inactive

- Options to load from and save to a selected unit / attribute were added
  - This system, over having the UI automatically load and save changes to the edited object, has the benefits of:
    - Easily copying entire lists of properties between units
    - Not forcing users to undo mistakes made to units

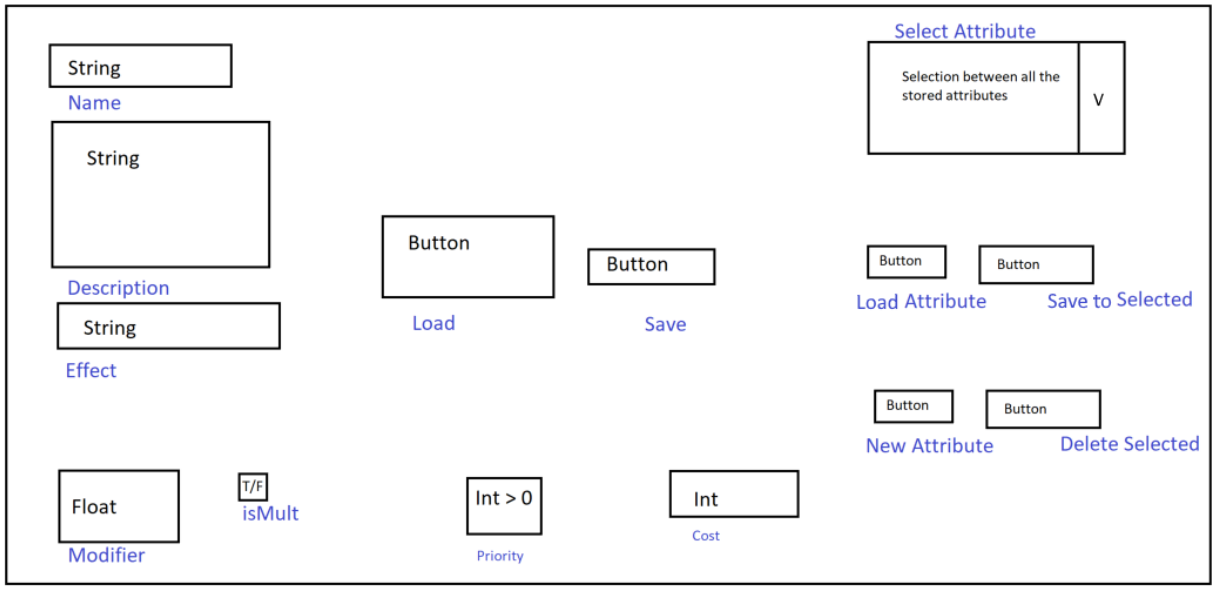


Figure 66: Revised Plan for the Attribute Creator

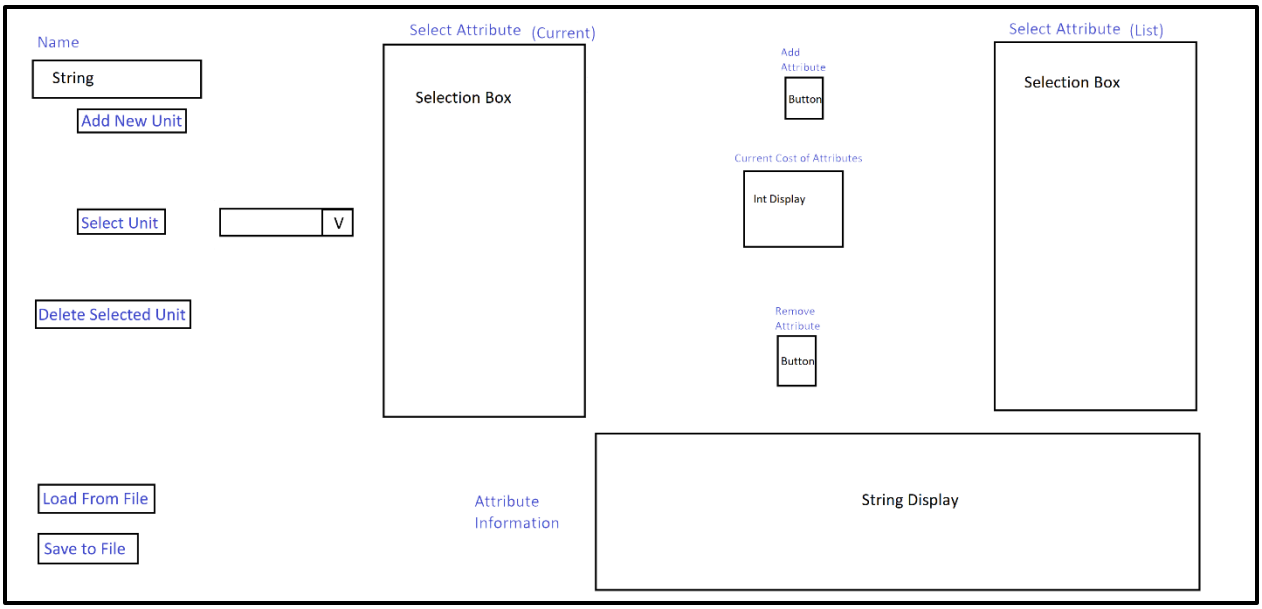


Figure 67: Revised Plan for the Unit Creator

Because the spacing in these designs were relatively complex, usage of data frames was essential to a palatable user experience. To keep coding as stream-lined and bug-fixing-free as possible, data frames were drawn before coding began.

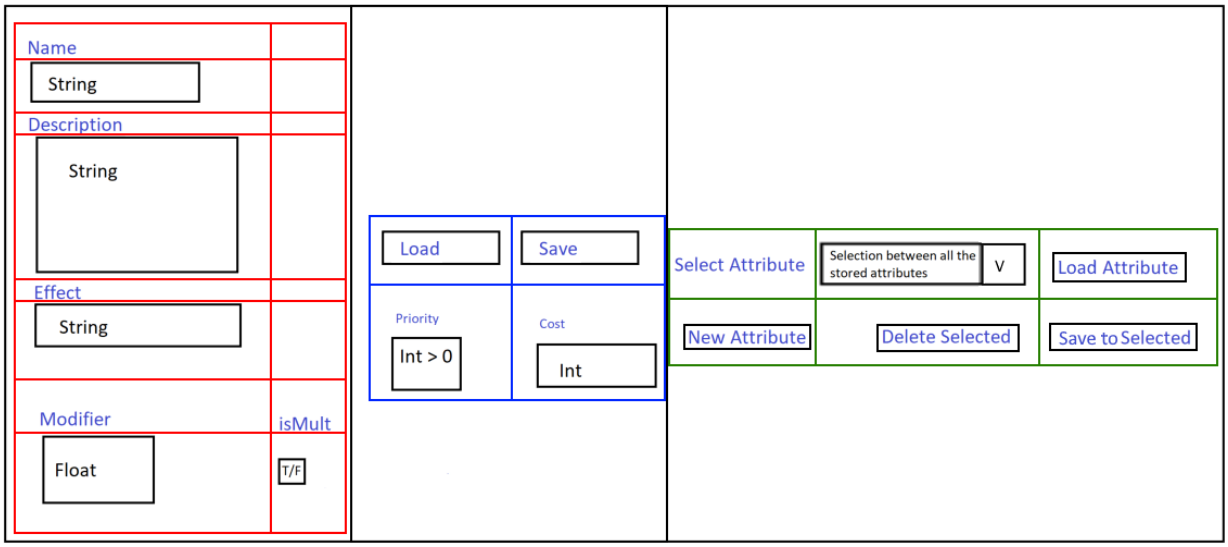


Figure 68: Attribute Creator with Data Frame Grids Drawn

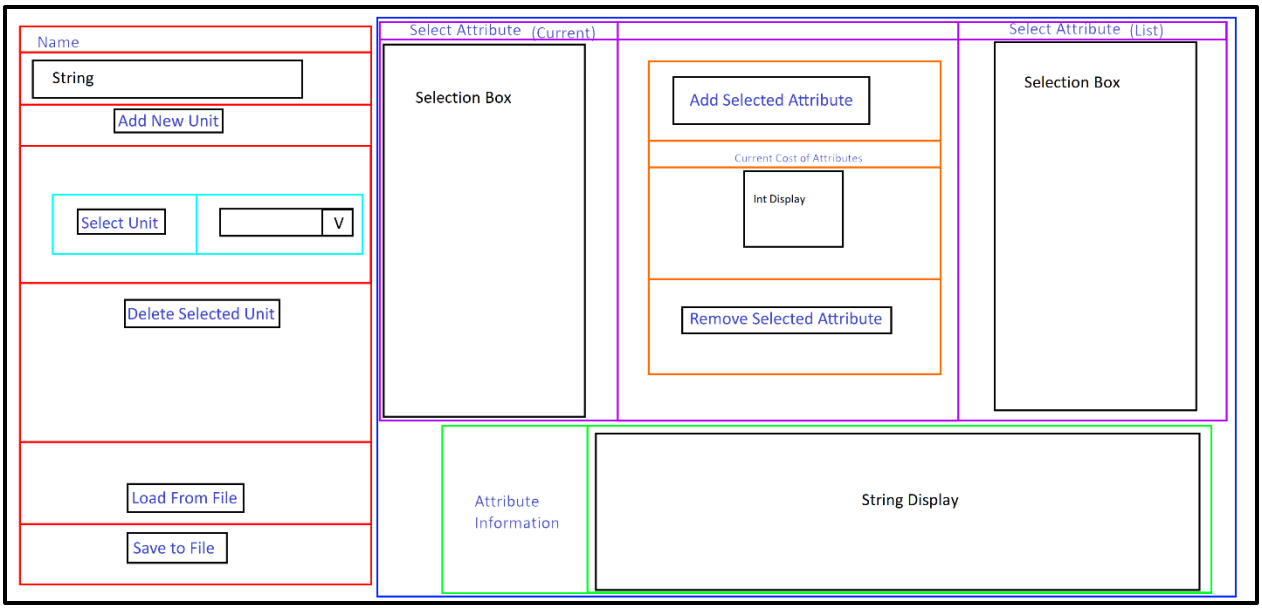


Figure 69: Unit Creator with Data Frame Grids Drawn

As a result of not taking the spacing of the frames perfectly into consideration, the spacing of the final product is not identical to the plan. However, it is functionally identical and near enough that the difference does not harm the end user.

Results and Limitations

Pictured below are the current versions of each of the interfaces. They have been used by the team to modify the JSON since their creation and have played an important role in development.

Figure 70: Current Version of the Attribute Creator

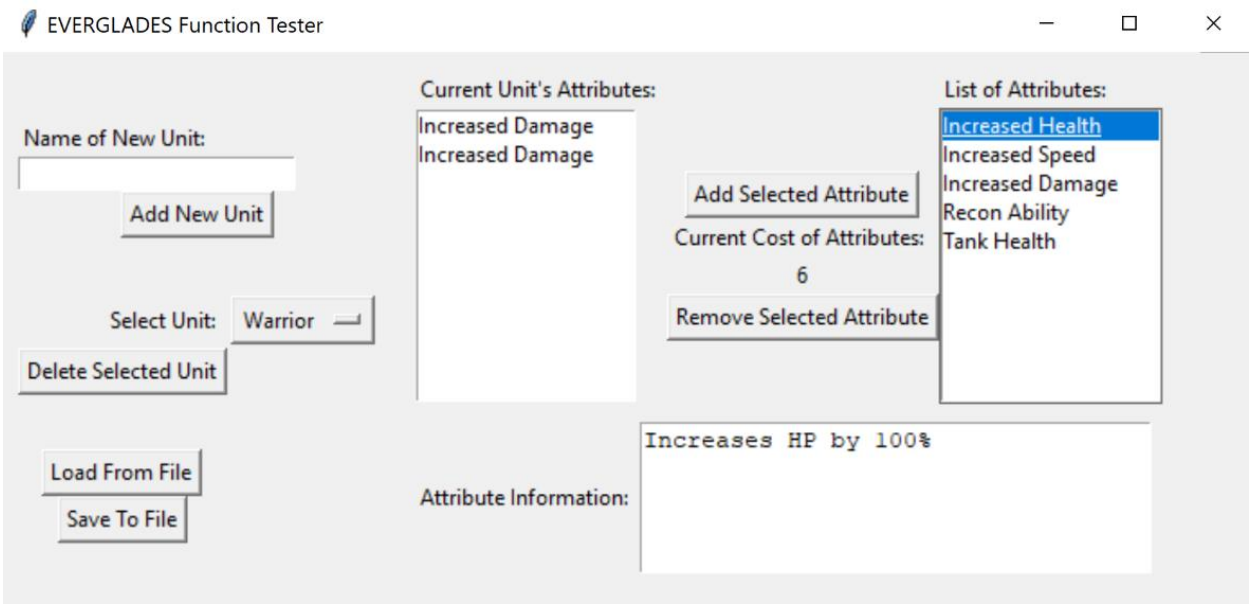


Figure 71: Current Version of the Unit Creator

Although not the most visually pleasing, they were functional – with stipulations. In pursuit of making them a more error-proof developer tool and a more accessible and comprehensive user tool, we proposed that the following changes be made:

- Attribute Creator
    - Ensure that the Name entry field cannot be empty
        - Though this is unlikely to cause errors in the rest of the code, this precaution will ensure it does not and that all attributes adhere to having a descriptive name
    - Display a visual warning if the Description entry field is empty
    - Ensure that the Effect entry field contains a valid keyword
    - Prevent the user from typing anything other than a decimal number into the Modifier entry field
    - Prevent the user from typing anything other than a 0 or 1 into the isMult entry field
        - Alternatively, change the isMult entry to a dropdown menu or other form of entry that restricts the user to a predefined list.
    - Prevent the user from typing non-integers into the Priority entry field
    - Prevent the user from typing non-integers into the Cost entry field
    - Reorganize the buttons surrounding the attribute selector
        - This ensures that no large changes, such as deleting or saving over units, is committed when attempting to select an attribute from the dropdown menu.
    - More descriptively name and label all aspects of the UI
    - Include a help tab or other way of teaching the user how to use the interface
- Unit Creator
    - Prevent user from editing text in the Attribute Information textbox
        - Editing the text is an unintended action, as the textbox is purely meant to be a consistent formatting tool for the descriptions of the currently selected attribute.
        - Editing the text has no impact on the saved description of the attributes. Though this means there is no functional harm in the current implementation, it is unsightly and may have new users wrongfully believe it has a purpose
    - Rearrange UI elements to be more visually pleasing and more conducive to understanding
        - Buttons aligned and sized differently on the left are ugly
        - By placing the Add Attribute and Remove Attribute buttons under the right and left lists, respectively, it would reinforce that they affect selections made in the box they are positioned under
    - Section off and label locations of the UI
    - Include a help tab or other way of teaching the user how to use the interface

Iterating the Design

To make the UIs more visually appealing and to facilitate ease of use by making similar functions visually tied, we iterated on the designs, resulting in the final designs shown below.

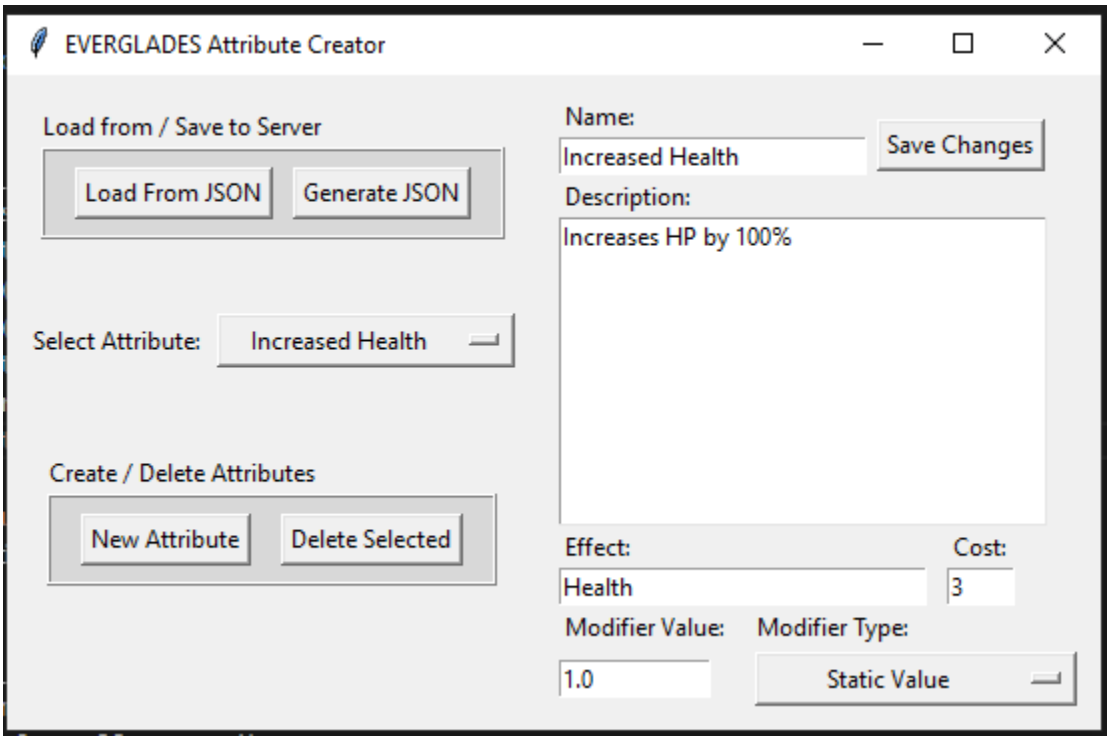


Figure 72: The final design of the Attribute Creator UI.

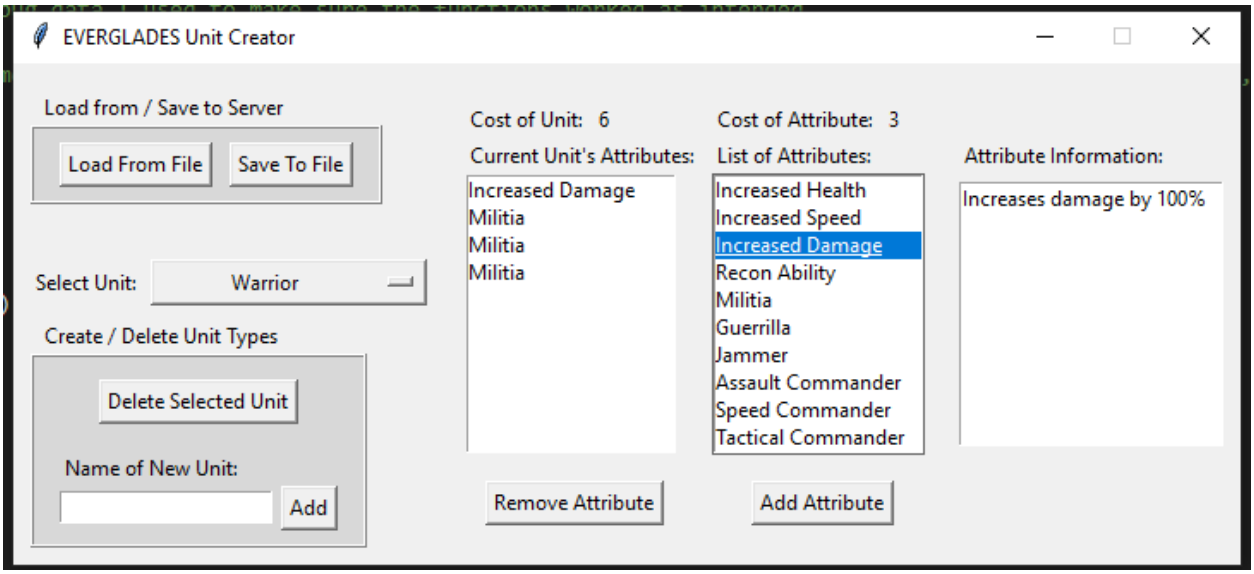


Figure 73: The final design of the Unit Creator UI.

This answered many of our proposed changes: edits cannot be made inappropriately to fields, done by selectively enabling and disabling them as well as vetting the types of characters that can be entered. Additionally, vestigial features like attribute priority were removed, and formatting of large text forms was adjusted to match the rest of the text in the UI, and groups of functions are in similarly-shaded boxes to show their category.

Everglades Desktop Application

A desktop application was developed to give the user an easy to use interface to setup and view the games. We chose to use the wxWidgets toolkit to develop the application in C++. Some other frameworks we considered are Qt, NW.js, and Electron. Qt has some features locked behind licenses whereas wxWidgets is completely open source, and both Electron and NW.js are chromium based so they use a lot of resources.

C++

Pros

- High performance
- Precompiled
- Static link libraries
- More control over application

Cons

- More complex language

Python

Pros

- Simple language
- Easy to debug
- Increasingly more popular

Cons

- Code is interpreted on runtime
- Need 3rd party modules to static link libraries
- Uses more memory

wxWidgets is a great toolkit for this because:

- Cross-platform
- Uses the native UI from the host operating system
- Open source
- L-GPL
- Allows for static link distribution

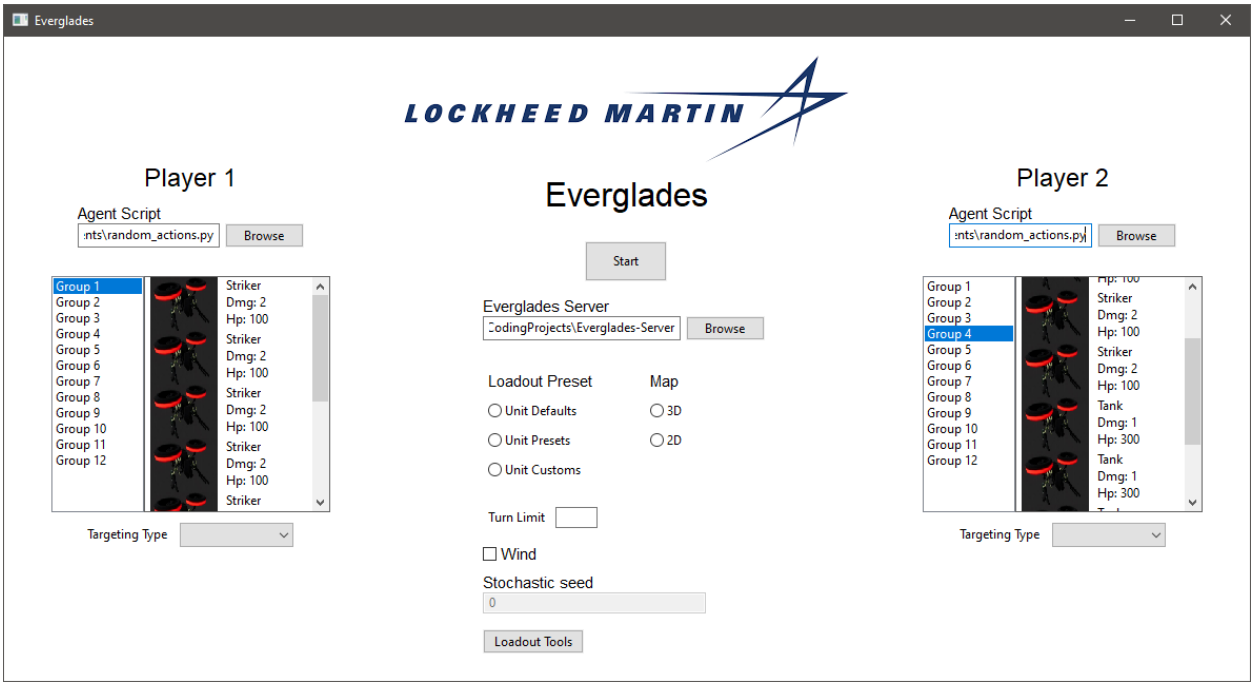


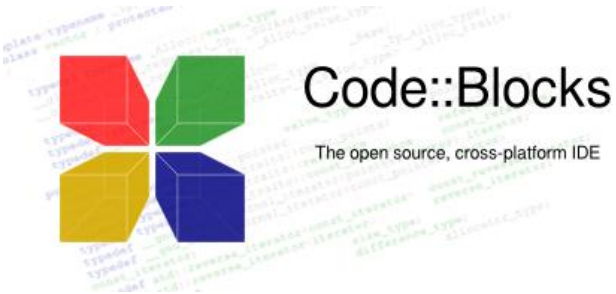
Figure 74: Desktop UI Screenshot

The application gives both players a list to view their loadouts and options to change the map. You can specify the agent script location and it will load in the team configuration for the respective agent. The server will use these specified agent scripts to run in the everglade server.

In the middle has a directory specifier that must be set to the Everglades folder in order for the application to run. It will also check if the specified directory is a valid Everglades folder by checking if the contents of the directory match a defined manifest of folders that have to be inside Everglades-server. Underneath that you can specify either a 3D or a 2D map and will generate the file, and which loadout preset to use. Next is a check box to disable or enable wind stochasticity effects and an input field that lets you specify the seed. And a button to open up all of the loadout UIs. The targeting types are automatically populated using the Python C API.

Desktop Application Dependencies

Below are the dependencies required for the desktop application with their respective version numbers.



20.03

Figure 75: Codeblocks Logo [3]

3.1.4

Figure 76: wxWidgets Logo [4]

8.1.0

Figure 77: MingGW Logo [5]

This application provides simple interfaces to change different characteristics of the game, such as map-type, wind stochastic seed, loadout.

The structure is to generate a GameSetup.json that will define different aspects of an Everglades instance on start up.


```
{
    "__type": "Setup",
    "MapFile": "DemoMap.json",
    "UnitFile": "UnitDefinitions.json",
    "PlayerFile": "PlayerConfig.json",
    "UnitBudget": 100,
    "TurnLimit": 150,
    "CaptureBonus": 1000,
    "Stochasticity": 0,
    "FocusTurnMin": 4,
    "FocusTurnMax": 6,
    "FocusHeatMovement": 15,
    "FocusHeatCombat": 25,
    "FocusHeatCooloff": 10,
    "RL_IMAGE_X": 600,
    "RL_IMAGE_Y": 380,
    "RL_ORTHO_X": 12,
    "RL_ORTHO_Y": 7,
    "RL_Render_P1": 1,
    "RL_Render_P2": 0,
    "RL_Render_SaveToDisk": 0,
    "SubSocketAddr0": "opp-agent",
    "SubSocketPort0": 5556,
    "SubSocketAddr1": "agent",
    "SubSocketPort1": 5555
}
```


Figure 78: GameSetup.json Sample

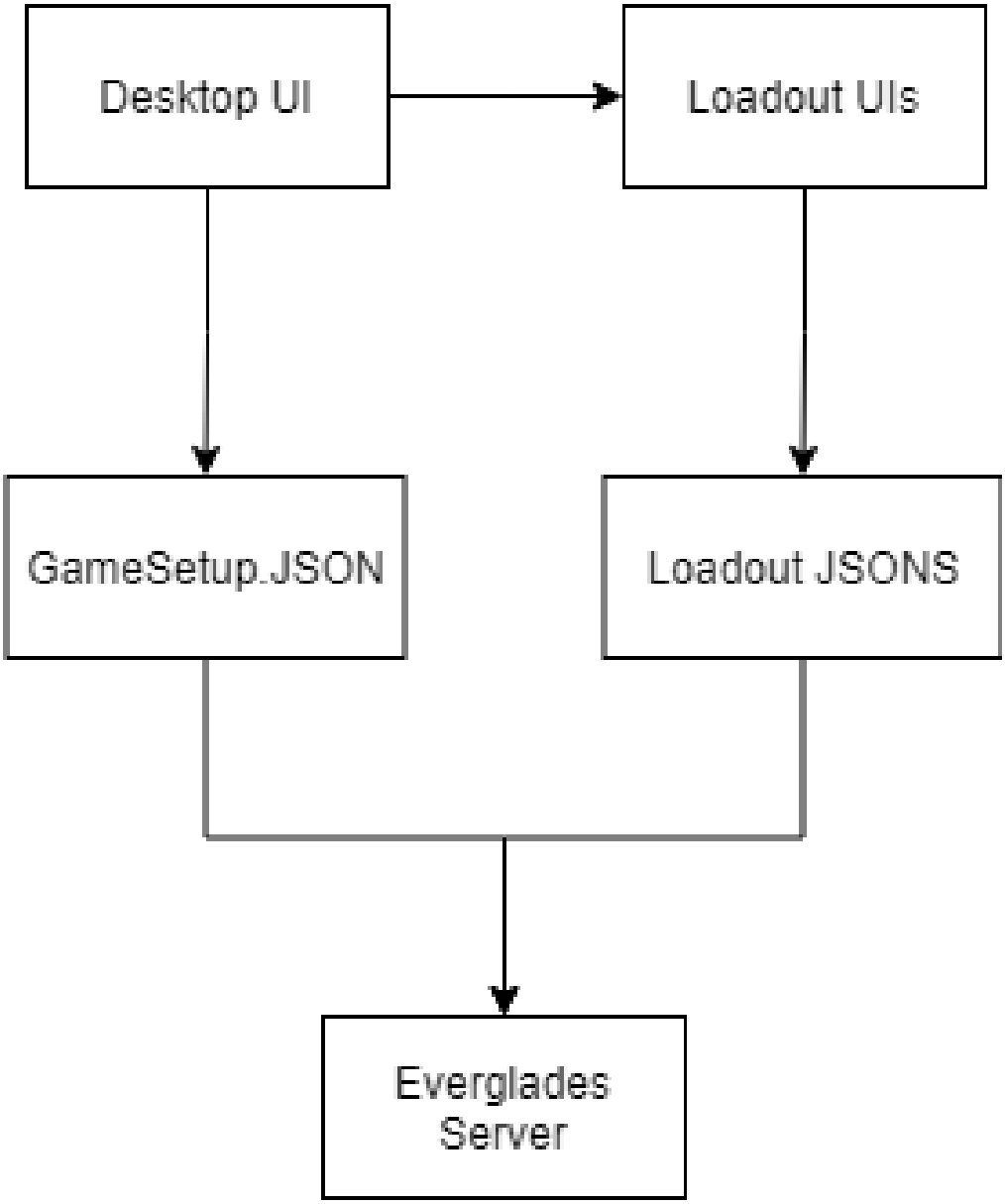


Figure 79: Desktop Application Flowchart

Unit Class Refactor

Justification

Towards the end of the first semester, we began looking into what needed to be done to fulfill the enhanced targeting requirement. To reiterate this requirement, we were tasked with improving the system by which drones made targeting decisions, namely by utilizing a callback function that allowed for multiple targeting functions. By the very end of the first semester, we had our doubts regarding the efficacy of implementing this targeting requirement off of the previous project's iteration.

There were two primary problems with the previous implementation that would make the targeting requirement exceedingly difficult to implement. The first was the cumbersome way in which the unit and group classes were implemented, which was the largest issue with the previous iteration. The second problem was how the *combat()* function was implemented, which naturally drives all aspects of combat, damage application, and drone death within the game.

It is vitally important that the previous implementation is discussed, as it will provide a full understanding for the changes we made to the previous systems, and why our iteration is structured the way it is. So, we will start by discussing the previous iteration's unit and group classes, as the issues borne from this were the most egregious.

The group class – *EvgGroup* – was structured similarly to how it is in the current iteration, with a few exceptions where necessary member variables were added. However, the main distinction lies in how it stored the units that constituted its group. In the previous iteration, a single instance of *EvgUnit* – the class which holds all information pertaining to individual units – was created for each unique type within the group. For example, if a group has five striker units and two tank units, then the group's unit array would be of size 2, since there are two unique unit types. So, in the previous method, units do not represent themselves within the group's unit array, but rather they are grouped together by type.

To look into the individual *EvgUnit* instances within the group, each instance of *EvgUnit* contained a *count* variable and *unitHealth* array. The *count* field dictated how many of that type of unit appeared in the group, e.g. the *count* would be two if there were two tanks in the group. The *unitHealth* array would be of size *count*, and would contain the current health of each unit of that type within the group.

From there, each instance of *EvgUnit* had a reference to an instance of *EvgUnitDefinition* which it created at instantiation. *EvgUnitDefinition* kept track of various static values associated with that individual unit type, such as the starting amount of health, the base damage amount, and so forth.

To better illustrate the concept of how the previous iteration implemented groups and units, the image below represents this structure. It is based off of the previous example of a group with five striker units and two tank units.

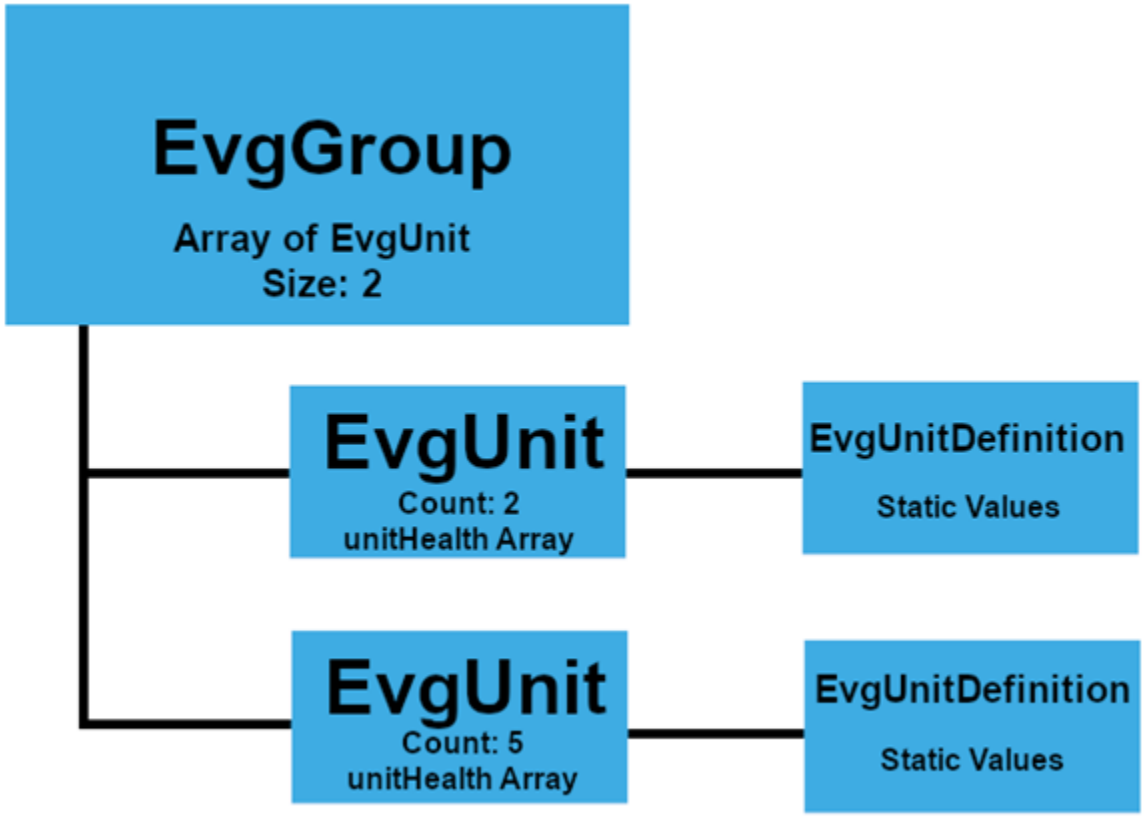


Figure 80: The previous group implementation.

On paper, this solution may seem fairly innovative and elegant, at it keeps the size of a group's unit array small, while still providing units all the information they need to fully simulate them. In practice, this methodology is plagued with issues that make it difficult to work with.

This implementation bundled up its information too tightly, making it hard to find a desired unit within a desired group. To find a certain unit in a group and apply damage to it, one would need to know the type of unit that needs to be targeted – or at least its index within the group's unit array – and also know the exact index of the unit within the *EvgUnit*'s health array.

Additionally, this implementation ends up being space inefficient. The *EvgUnitDefinition* class does not change between unit types, regardless of what team the unit type is a part of. Instantiating a new instance of the definition class for each unique unit type within a group is a horrible waste of space, because there is the potential for a lot of unnecessary repeated instantiation. If a team has twelve groups each with at least one striker unit, they are created twelve separate instances of *EvgUnitDefinition*, all of which have the same information that is guaranteed to not change at all through the

course of the game. The definition class contains only static values that are core attributes of the unit type and thus would only be instantiated and referenced when needed.
The second issue lay with the previous implementation of the *combat()* function. Understanding the *combat()* function and ensuring it worked well is instrumental to fulfilling a targeting requirement correctly, since the two systems went hand in hand. Combat and damage application could not happen without a way to select what enemies to apply damage to, and a targeting system would be useless without a way to enact some sort of change based on the selected target.

The difficulties with the *combat()* function were a result of the awkward combat and unit class structure, with the complexity required to get a combat function working introducing added difficulties in reading and understanding the code. The code that was there – while it assumedly worked – was very poorly commented with variable names that made it difficult to discern what the intended purpose was.

For approximately two-thirds of the previous *combat()* function, the desired functionality was understood with little effort, as it was abundantly clear what the purpose of each step was, with some comments being present to better help ascertain the function of each line. It is in the last third of the function where the code becomes unreadable. The last third revolves around applying the damage that was built previously to the correct units in the correct groups.

Due to the group and unit class implementation, a lot of bizarre indexing and looping had to be done to get the correct unit. It is here, too, where the comments that proved useful in the previous segments of the function disappear. While some comments exist, they are not descriptive enough to give a full understanding of what is happening. Since this is where damage is actually being applied to units, it is arguably the most important segment of the function to understand as it justifies the efforts in the previous two steps.

While it was apparent that the latter portion of the *combat()* function was moving an index up and down a group's unit array in response to various factors – like unit health or the number of units in the group – it also appeared to be too general. Our targeting goal required us to have specific units targeted, though the previous iteration appeared to be content with just targeting whatever unit it found first that was of the correct type.

With the start of the second semester, we agreed that a refactor was necessary to better fulfill the targeting requirement. The *combat()* function itself was too convoluted to try to fully understand. It also would have made the creation of the targeting requirement difficult as we would have had to bow to the awkward indexing and class structure created by the previous iteration.

Proposal

Thus, we approached our sponsors at Lockheed Martin with a request for a serious refactor of the group and unit classes, and of the *combat()* function. To do so, we created a full proposal that outlined a new unit class system along with all necessary changes that would have to be made across the entire program to fully refactor the code. Since we would be changing the group and unit classes directly, there would be many areas where problems would arise, since other functions utilize these classes other than combat.

Shortly after submitting it to Lockheed Martin, the proposal was approved with one stipulation: we had to perform regression testing after finishing the refactor to ensure the changes did not drastically change the intended outcome of the program.

To craft the proposal, a full workup of the current state of the project was done to fully understand how all the various systems interacted with one another, and to know what areas would assuredly be affected by the change.

Obviously, the proposal outlined that the group, unit, and unit definition classes alongside the *combat()* function would need to be modified. What was initially unforeseen, but discovered through the investigation required for the proposal was changes necessary to *game_init(), sensor_state(), player_state(), capture(), game_end(),* and *build_knowledge_output()* as all of these functions ended up requiring some modification to allow the refactor to work completely. For most of these functions, the changes were not major – as it often just required some small implementation to emulate a previous feature, such as the number of unit types in a group – to get it to work. The most demanding by far was *game_init()*, as the core structure of the function had to be modified to fit the new method.

With the proposal approved, we started work on the refactor immediately with intentions to test our new implementation and compare it to the output of the previous implementation. The goal was to compare our version of randomly selected targeting to theirs, as that was the only form of targeting the previous iteration had implemented. So long as the output ended up being the same, the refactor would be considered a success and would be accepted by Lockheed Martin.

Implementation

This section will outline the current implementation of the group, unit, and unit definition classes, and then discuss the current state of the *combat()* function. It will not delve into the targeting requirement, though it will mention it within the section discussing the *combat()* function and how it relates to it.

The new method of representing groups and units is much more direct. The primary frustration with the old method primarily revolved around its tightly compressed way of storing information about the units in a given group, which made it difficult to

access the information in a reliable and straightforward manner. The new method attempts to eliminate this trouble in many ways.

To begin, we will evaluate the new *EvgGroup* class, which altogether is not too different from its previous implementation, save for the unit array. The unit array still keeps track of what units are part of the given group but expands it. Before, if a group has seven units with two unique types, the group would have a unit array of size two. Now, a group with seven units has a unit array of size seven. This allows for an easy, straightforward way to iterate across all units contained within a group, as all one needs is a reference to the unit array.

This naturally ties in with the new implementation of the *EvgUnit* class. Now, units represent themselves within the group, as seen by the way the group's unit array is the same size as the number of units that are in the group. The *unitHealth* array has been removed in favor of tying the current health value directly to the unit it corresponds with. All dynamic attributes associated with a unit that can conceivably change within the course of the game are now part of the unit class, namely the unit's current health value. Associating the health value with the unit directly makes for a simple way to get the health of any single unit, while still allowing for ways to get the health of all units in the group via lambda functions and sorting methods.

All static values associated with units are still contained with the *EvgUnitDefinition* class, but they the definition class is not instantiated for each new unit type in a group, nor does the unit class contain a direct reference to the definition class anymore. Instead, a global dictionary called *unit_type* is maintained.

In the *unitTypes_init()* function – which is called when the *EvergladesGame* class within *server.py* is first instantiated – the *unit_type* dictionary is set up with all static information that pertains to each unit type. These definition classes are only instantiated once for each unique unit type within the game, and it is fully scalable to support new unit types later down the line. The definitions are hashed to the dictionary based on a unique unit type ID which is just a raw integer that starts at 0 and increases for each new unit type.

To better illustrate the current implementation of the group, unit, and unit definition classes, the image below represents the same example group used previously, with two tank units and five striker units. Not all instances of *EvgUnit* are shown in this image, but assume that there are four additional instances of the unit class within the group.

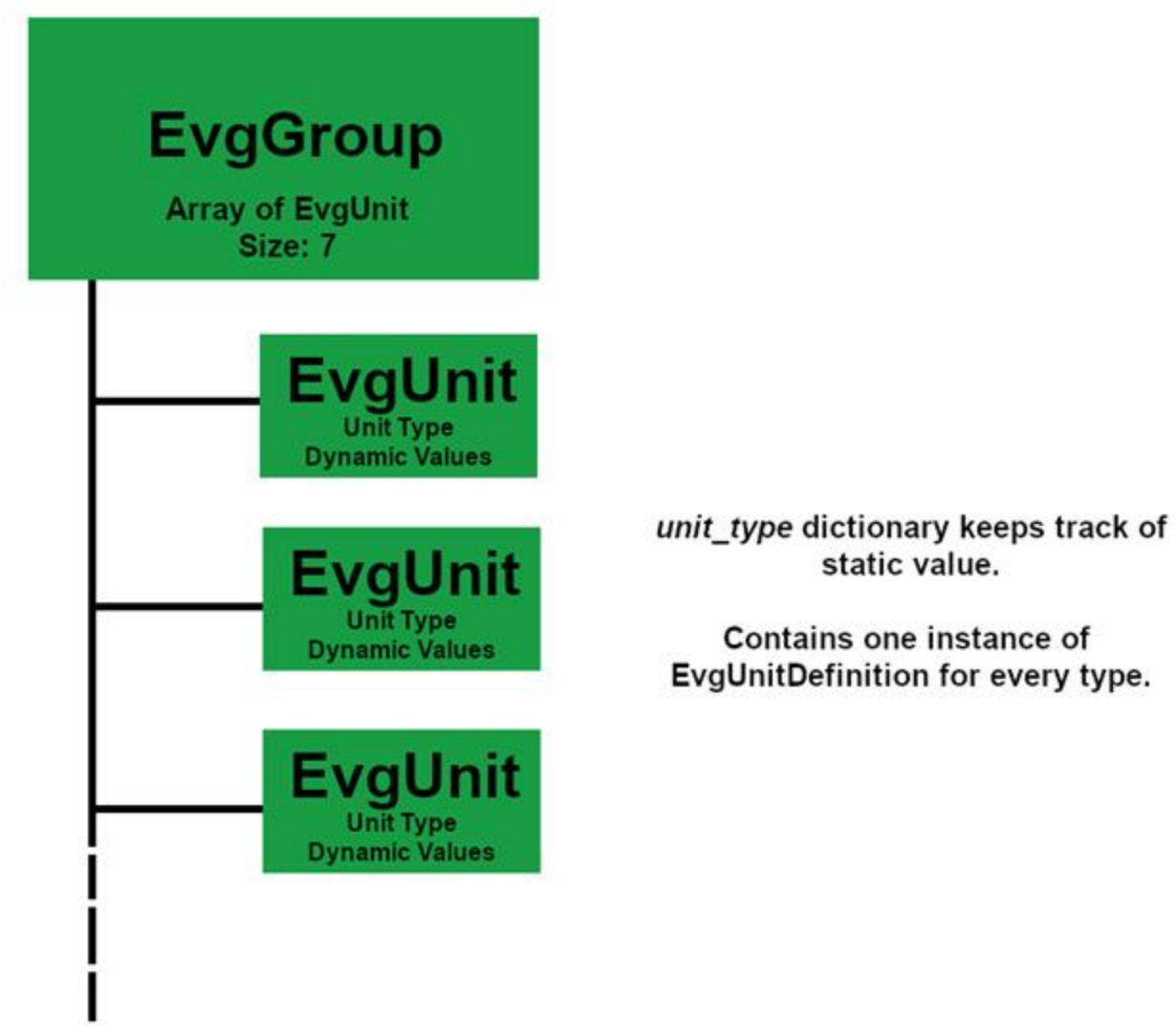


Figure 81: Original implementation.

It can be difficult to comprehend exactly what changes were made and how they were impactful. The following table compares the two systems in the key areas where they differ and how the program has to accommodate their implementation.

Unit Class Structure Comparison

	Previous Method	Current Method
Type	Substitutive – Units are substituted by their type in the group unit array.	Representative – Units represent themselves in the group unit array.
Dynamic Values	Only supports health in health array tied to unit type.	Dynamic values are tied to each unit instance. Can support as many dynamic values as needed.
Static Values	In definition instance tied to each unit type instance.	Placed in global dictionary that is referenced whenever a static value is needed.
Space Efficiency Issues	Instantiates the same static definition class multiple times across groups.	Instantiates the static definition class once per unit type.
Iteration Methods to Find a Unit	Must have a reference to the player, group, and unit type, and then the correct index within the *unitHealth* array.	Must have a reference to the player and group, then can either have a reference to the unit or the unit's index within the group.

Now that the new implementation of groups, units, and unit definitions has been explored, the new *combat()* function will be discussed in detail. Although he previous implementation of combat was confusing and would require a great deal of understanding and tweaking to get it working with enhanced targeting, the core concepts that were discerned from the combat code appeared solid. Thus, the general format of the old combat was repurposed in the new combat function and given sufficient comments to make its function clear.

The current *combat()* function is broken up into three main steps, which were touched on in previous sections but will be fully discussed here: detection, construction, and destruction. Each step corresponds to one of the three main tasks of the *combat()* function: detecting contested nodes that require a combat simulation, building the

damage generated by units targeting enemies, and applying that damage to the correct units, removing dead units, and disbanding empty groups.

It is important to note that the *combat()* function is called every turn to detect and simulate combat should it occur. Combat will only happen between two groups if their of them are in transit, ensuring that it happens after movement is finished. So, a group leaving or entering a node cannot engage in combat until their status is marked as "arrived".

The following three sections will discuss each step in detail.

Detection Phase

The detection phase is the first to happen. Since the *combat()* function is called every turn of the game and there is not an existing method or system that evaluates the status of each node, the *combat()* function must do this itself.

In this phase, it will search (in linear time) through all nodes on the gameboard. Since there are no attributes distinguishing nodes from one another in terms of their contested status, the linear search through all nodes is unavoidable, though this is certainly one area of improvement for future groups.

A node is marked as being contested in this phase if there is at least one group present at the node from both players. Combat cannot occur if only a single team is at a node, because it does not make sense for a team to fight its own units. As noted previously, combat occurs after movement and will disregard units in transit to or from the given node. So as long as there are one or more groups from each player at the node, and those groups are not marked as being "in transit", combat is set to occur at that node.

The contested node is added to a list of all contested nodes found and groups that are marked as having "arrived" at the node are added to a list showing that they are available for combat. All units within that group are evaluated, and if they are alive – meaning their health is greater than 0 – then they are added to a list of units available for combat in that group.

Some additional considerations are made at this stage for units with the commander attribute, as a unit with this attribute will provide a combat damage modifier for all allied drones present.

This process repeats for all groups that are present at all contested nodes. It should be noted that the *combat()* function will simulate damage as they come across contested nodes. This means that it won't mark all contested nodes first, then come back and simulate damage on all nodes which were contested. Instead, it will simulate combat at a node if it detects it is contested before moving on to the next node in the map. If a node is marked as not being contested, the detection phase will still move on to the next node in the map and repeat until all nodes have been evaluated.

Since all of this happens inside of a single function call, all of the map nodes will be evaluated and simulated within a single turn. Combat does not happen one turn on one node and then happens on another turn just because combat is being simulated on another node. Put simply, if there are five contested nodes, the combat will be simulated on all them equally on the same turn, not on five separate turns.

The detection phase's main purpose is to detect the presence of contested nodes, see what groups and units are present at that node, and evaluate whether the present groups and units are truly available for combat. If all of these checks are passed, the groups and units are added to the *activeGroups* and *activeUnits* dictionaries respectively, and further execution of the *combat()* function is authorized.

Construction Phase

The main purpose of the construction phase is to build the damage that is to be applied to all drones on both teams. This process involves selecting the correct targeting function based on the player and making a call to that targeting function to decide what enemies are targeted by what units.

It is important to note that building damage does not mean subtracting actual damage values from the current health of a unit. Rather, it is determining ahead of time what drones will have damage applied to them, and how much damage will be applied to them.

There is a simple reason for this method: it prevents unfairness. Since Project Everglades is a turn-based game, combat is simulated in a turn-based way as well. At not point do units engage in truly live combat, where every second that elapses in real life causes a second to elapse in the game. Instead, contested nodes are guaranteed to have combat occur, and drones within a contested node are guaranteed to be a valid target for damage.

If damage was not built before being applied, an unfair advantage would be given to whatever player or group is allowed to attack first. So, Player 0 – which will always appear first in the list of players – would have their first group attack before any others, including any of Player 1's groups. This would potentially mean that Player 0's group could completely wipe out Player 1's group before they get a chance to attack themselves.

In a real-time game, this would be a completely acceptable scenario, but since there are no systems in Project Everglades that determine real time combat, sneak attacks, or firing accuracy, there is a need for building damage.

By building damage before applying it, all groups on both sides of the game have a chance to attack before possibly being destroyed. Applying damage to units in an enemy group is just as important as killing enemy units, so the opportunity to attack should never be dismissed as being unimportant.

The very first step in the construction phase is to loop through the players list – essentially just evaluating player 0 and player 1 – and calling the relevant callback function that allows for targeting to take place. We won't delve into how the callback function or targeting works here, but for now know that reference to a list of combat actions is passed to the callback functions.

Once those functions finish execution, the *combatActions* list will be filled with tuples, each labelled as an action in their own right. These tuples have four separate indices which correspond to something different. The following quickly lists what each index is for:

Combat Actions Tuple Entries	
Index	**Value**
0	The targeted enemy's player ID (0 or 1)
1	The targeted enemy's group ID.
2	The targeted enemy's unit ID.
3	The base damage to be applied to that targeted enemy.

Each combat action in the *combatActions* list is guaranteed by the targeting functions to have this information in it in the listed order. The data is extracted from each action, with the first three indices being used to index the amount of damage that will be applied to that unit. The base damage is placed inside of a dictionary called *infliction* which sorts the built damage by player, group, and unit ID. If an entry already exists for that unit, then the damage is simply added on to whatever damage already exists.

In summary, the construction phase builds damage. It is perhaps the shortest phase within the *combat()* function, although its call to the targeting functions is a notable part of the phase. The reason damage building is important is that it ensures fairness in a turn-based game like Project Everglades. After the units have selected their targets, it will set up the base damage values that will be applied to each individual unit in the next step.

Destruction Phase

In the final phase of the combat function, damage that was built from the construction phase is actually applied to the current health of the targeted units, and any logical follow-throughs that can happen as a result of losing health will be evaluated.

The destruction phase works by looping through both players and going through all groups that were placed within the *infliction* dictionary set up in the previous phase. The units that were targeted and placed in that same dictionary are then looped through based on the group achieved in the previous loop. These units are the "target unit" and are the unit which the damage built previously will be applied to.

Once the reference to the target unit is found, the damage equation can start to be calculated. Firstly, there is a consideration to be made if the group has a unit with the commander attribute present. If this is the case the base damage generated from the *infliction* array is immediately modified with 1.5 times itself, as this is the bonus granted by a commander's presence.

Next, we delve into the core components of the damage equation. The node's defenses are first brought into consideration, with bonuses being conferred if the team of the unit being attack controls the node, and if the node has special resources available. If the node is controlled and the node has a stronghold resource present, then those factor into the base defense granted by the node in the following equation:

$$\text{node defense} = (\text{controlled} + \text{stronghold}) * \text{node defense value}$$

Figure 82: The node defense equation.

Where the *controlled* variable is set to 1 if the player being attacked controls the node, the *stronghold* variable is set to 1 if there is a stronghold present at the node, and the node defense value is an intrinsic attribute of all nodes, which varies throughout the map.

With the node defense value calculated, the actual damage equation can be calculated properly. The following equation is used to calculate the damage amount that will be dealt to the targeted unit:

$$\text{damage dealt} = \frac{10 * \text{base damage}}{\text{target health} + \text{node defense}}$$

Figure 83: The combat equation.

Where the *base damage* variable is the value achieved from the *infliction* array. The *target health* variable is the base health attribute of the targeted unit's type, which can be seen as its armor attribute. The *node defense* variable is, of course, the variable we calculated in the previous equation.

Once *damage dealt* or "true damage" is calculated, it can be applied to the targeted unit. This is done in a very straightforward fashion – since we already have a reference to the targeted unit, its current health can be referenced and subtracted from easily.

Once damage has been applied to the targeted unit, various checks are necessary as the state of the game has potentially been changed in a very important way. First, if the unit's health has dropped to or below zero, then the unit's health needs to be marked as zero. Systems outside of the *combat()* function will register a unit with zero health as being dead, so ensuring that a dead unit has zero health is essential. Additionally, it does

not make sense for a unit to have less than zero health, because if health is an indication of how “in-tact” a unit is, a unit can be less than 0% in-tact. There is no point beyond “obliterated”.

Additionally, a destroyed unit requires various updates to occur with the group it was a part of. Every group in the game maintains a *counts* dictionary which helps determine how many units of a single type are in that group. Naturally, if a unit of any given type dies, *counts* need to be updated to reflect the fact that it has lost one of its members.

When a unit is destroyed, the group also needs to be checked to see if there are still other units within the group. If the sum of the *counts* dictionary’s values is zero, then that means there are no units left in the group. As such, the group should be disbanded because there are no units left to take and execute orders. At this step, the group’s *destroyed* attribute is set to True, with the *moving* and *ready* attributes set to False.

This process continues for all groups and units which were placed inside of the *infliction* dictionary from the construction phase until they have all had damage applied to them. Since these phases operate solely off the *infliction* array, which guarantees that only affected units are placed within it, there will not be a case where an untargeted unit is processed within the destruction phase.

One all units in the *infliction* dictionary have been processed and have had damage applied to them, the entire process starts over again at the next contested node, should the detection phase find any such node remaining in the map on that turn.

In summary, the destruction phase is responsible for handling all matters of damage application. It will ensure that only targeted drones will have damage applied to them since it operates solely off of the *infliction* dictionary. The damage value it pulls from the *infliction* dictionary is processed through the damage equation and applied to the targeted unit and continues until all units are processed.

Telemetry Data

While there are three main steps to the core loop of the *combat()* function, there is a final step that must be touched on before the function terminates. Telemetry data must be generated and formatted to reflect the actions which took place during the execution of the *combat()* function on the current turn.

Throughout the destruction phase, some lists were appended to with information regarding the groups and units that were affected, and the amount of health the affected units had at the end of that combat turn. These lists are accessed when the telemetry data is being generated as they describe what occurred at that node during that turn.

The telemetry data will only output units that were affected during the combat turn. This means that if a unit is never targeted and never has any damage applied to it during

the destruction phase, it will not appear in the combat update telemetry data. This does not mean that the unit suddenly disappeared or is randomly appearing throughout the course of the game – the telemetry data is just structured to only show data when there is a definite change in the state of the units.

The rows of the telemetry data are formatted with the following entries:

Telemetry Data Columns

Column Name	Value
0	The turn the data was collected on.
player	The player whose groups and units are displayed on that line.
node	The node ID the combat took place.
groups	The universal group IDs.
units	The universal unit IDs.
health	The health of the corresponding units after combat.

The arrays displayed in the telemetry data are parallel arrays, which are, “multiple arrays of the same size such that i-th element of each array is closely related…” [9] Effectively, they are 2D arrays where the rows of the array are divided up amongst multiple separate 1D arrays. Each row in the parallel array is either the groups, the units, or the units’ health listed in the telemetry row.

To help illustrate these concepts, look at the image below, which shows the first line of a sample telemetry file:

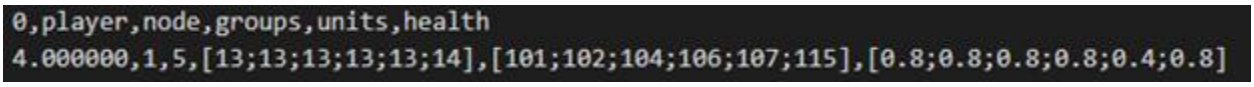


Figure 84: A sample line from a telemetry file.

From this line, we know that the groups and units listed engaged in combat on turn four, at node 5, and belong to Player 1. With how the parallel arrays are structured, the unit at position *i* of the unit array will have their group ID and current health value at the same position *i* in the other arrays. For instance, the unit whose ID is 115 in the second array belong to group 14 and currently has 0.8 health.

Regression Testing

As mentioned previously, Lockheed Martin accepted our proposal to perform a major refactor of key classes and functions so long as we performed regression testing afterwards. Regression testing is “… a type of software testing to confirm that a recent program or code change has not adversely affected existing features.” [6] Essentially, our sponsors just wanted us to confirm that the major refactor we performed did not drastically change the desired output of the program.

Approach

Much of the work we performed this semester ended up changing the output of the program in some way, so instead of coupling the refactor with other changes we made to meet various other requirements, the refactor took place on a version that contained the original code that comprised the previous iteration.

Once the refactor was complete, all errors were solved, and the program was outputting telemetry data again, it was time to begin regression testing. To achieve this, the old, unmodified *server.py* and *definitions.py* files were reintroduced to the program and labelled as *server_old.py* and *definitions_old.py*. These files were then hooked up into the *Everglades_env.py* file, which kickstarts the entirety of the program. It is set up in such a way to run the game both under the refactored server and definitions files and under the old server and definitions files. Since these were the only files affected by the refactor, they were the only ones included from the previous iteration.

Now that the Everglades environment was calling both our refactored files and the old files one after the other, further set up had to be done to ensure there was deterministic output. In the previous iteration, the only way units knew which enemies to target was through the use of random number generators to pick a random unit from a collated list of available enemies. At this point in time, the targeting functions were not finished, but this was fine as the only targeting function needed was one which randomly selected units. By having a targeting function that mimicked the way targeting was done in the previous iteration, the two random number generators could be seeded with the same value to ensure deterministic output.

With seeded random number generators, this means that output can be reliably deterministic and easily compared to see if there were any drastic changes between the two implementations. Deterministic output means that the telemetry data that is written to the file should be the same every time the game is run with a given seed. So, if the random number generators are seeded with the number ‘5’ and a unit dies on turn 10, that same unit should always die on turn 10.

Ensuring deterministic output was only half of the setup that was required, however. To better test combat, we created a way to ensure that combat always occurs exactly once per game at the same node. To do this, we created a custom agent, map, and test file. The test file – *testcombat.py* – served to always call the testing agent and map files whenever it was run. The agent file – *reggtest.py* – has a very simple *get_action()* function which always sends the same groups from both players to the same node on the map. The map – *reggestmap.json* – is only comprised of three nodes: the player start positions and a middle, uncontested node.

Whenever someone runs the *testcombat.py* file, the two players on either side will always send their first group to the middle node. These groups will do combat until the

node becomes uncontested, and the game will naturally end after 150 turns. The agent script can be further modified to send more than one group to the middle node, or a group other than the first group a player has to the middle node.

To modify the *reggest.py* agent to move different groups, the *get_action()* function would need to be modified. First, the *action* variable must be set to *np.zeros((x,2))*, where *x* is the number of actions that are desired. Note that a player can only take seven total actions per turn. To select a group to act with. The following code snippet shows how to modify this script for further testing purposes.

```
# This is what is determining the actions for the current turn
def get_action(self, obs):
    action = np.zeros((2,2))
    # action[x, 0] = g selects group g for action x
    # action[x, 1] = n sends group g to node n for action x
    action[0, 0] = 0 # Group 0
    action[0, 1] = 5 # Node ID 5

    action[1, 0] = 1 # Group 1
    action[1, 1] = 5 # Node ID 5
    return action
```

Figure 85: The get_action() function.

With the custom regression testing scripts in place and a way to ensure deterministic output across both the new and old files, everything was set up for regression testing. To achieve this, the number of groups and the compositions of the groups were altered in various ways, with the test script being run multiple times on the same configuration using different seeds. Each time the test script was run, the output files were compared to see if there were any differences.

To give a good idea of the ways in which the refactored code was tested, here is a comprehensive list of all the group configurations used:

- Single group, one unit type per group
- Single group, two unit types per group
- Single group, comprised solely of recon units
- Two groups, one unit type per group
- Two groups, two unit types per group
- All twelve groups in the default loadout files
- No groups

Each configuration was run at least ten times, each time with a different seed. Each configuration highlighted some issue, which typically came down to how telemetry data was being collected and formatted. It was here that we discovered that the previous implementation hid units from the telemetry file who not affected in combat.

By the end of regression testing, we were able to have the refactored telemetry data match up with the old telemetry data, except in cases where there are multiple groups at a node. Thankfully, this was not a genuine issue that had a significant bearing on the course of the game, as it did not force battles to be long and drawn out wars that lasted until the very last turn of the game.

The reason for the discrepancy between the new and old implementations is due to how random targeting was handled. In the old implementation, a unit was selected randomly by taking the total number of units across all present groups and randomly picking a number between 0 and the total, with the result being the index of the desired unit to target.

In our implementation, since we are operating more with precise references to objects instead of blind integers, in cases where there are more than one group at the node, a random group is first selected and then a random unit from that group is selected. So, we have an additional layer of choice. We brought this to the attention of Lockheed Martin, and since it did not drastically affect the outcome of the program for the worse, the slight difference in output was deemed acceptable.

Unit Health Representation

A final important note is how the health of units is currently represented within the project. Effectively, there are two different systems of representing unit health, the reason for which is directly caused by regression testing.

Before explaining the connection, the two representations should be discussed. Currently, health is represented differently on the backend than it is in the telemetry data. Within the code and unit classes, the health of all units starts at 100, represented as a float. It is meant to represent the unit's total health out of a percent, so when it is first created its health is naturally at 100%, and as they take damage the percentage decreases.

When output to the telemetry data, the health is represented as a decimal out of the unit's base health specified in the telemetry files. For instance, the base health of strikers is set at 2.0, so a striker at full health would be displayed as having 2.0 health in the telemetry files.

This health representation was not chosen by us, but we had to use it to ensure that the outputs we generated were the same as the previous implementation. The combat function they set up could not be changed – nor did it really need to – and it was set up in such a way to output damage values that made more sense for health representations out of 100. For instance, the damage applied to a unit generated by the damage equation would typically be a number significantly larger than 3.0, which is the highest base health any unit can have.

There were attempts on our part to represent all health under one system, typically by dividing the result of the combat equation by 100 to get a value that made more sense within the context of the health values set in the *UnitDefinitions.json* file. Oddly enough, the presence of tank units broke this attempt, and would cause the battles to draw out well beyond the 150 turn limit. It became apparent that the previous implementation was designed to work with these two separate health representations, and so it would simply be easier and more straightforward to follow their example.

So in the current implementation, the *currentHealth* attribute of all units is set to be out of 100. When damage is applied to a unit, it is done so in a straightforward manner, and that damage is then added to a separate attribute on the unit that keeps track of the *outputHealth* attribute, which is the health that is actually sent to the telemetry data. To achieve this number, a simple calculation must be made to convert the percentage representation to the simplistic representation, whose equation is shown below:

$$\text{output health} = \text{unit base health} * \frac{\text{unit current health}}{100}$$

With this change implemented, regression testing was completed. As stated previously, the only major difference between our output and their output comes when multiple groups fight at the same node. However, the actual implications on the course of the game are so minor that it is a non-issue.

With regression testing completed, the unit class and *combat()* refactor were completely finished, allowing for a smooth development of the targeting functions that allowed us to fulfill the requirement. The ability to iterate over units easily and quickly within a group, with units being indexed by player and by group proved incredibly useful in this endeavor. Additionally, the refactored *combat()* function made it very easy to adapt to new requirements with targeting, and ensuring the two systems aligned correctly, because we had a solid understanding of how the *combat()* function operated.

Drone Targeting

Overall Requirements

Improvements for how drones target other enemy drones is one of the few “nice-to-have” goals Lockheed Martin had provided us with. Previously, drones in Project Everglades did not exhibit intelligent targeting behavior. It was originally designed so that when drones occupy the same node as enemy drones, they targeted enemies entirely at random.

Lockheed Martin wanted to see this system improved upon, because a lack of attack priorities is not a very realistic representation of real-world battles and enabling such a system would allow for a new, complex layer of strategy.

The overarching goal of this requirement was to make drones prioritize their attacks based on different parameters. Initially, we created two targeting functions where drones would purposefully attack enemies based on their overall health attribute. The first targeting function was based on the enemy drones with the highest health and the second targeting function preferred enemy drones with the lowest health.

Once this system was fully implemented, then it was expanded upon to operate off various other drone attributes. Health is not the only possible prioritization option, as drones also have speed, damage, and drone type attributes that could be used in attack prioritization. The system allows a user to create their own targeting system based on a few unique inputs, such as priority and type. For example, a user would be able to specify that they want a certain number of drones in one squadron to only attack enemies with the highest control stat.

System Implementation

To be able to target the drones, a helper function was created to sort the enemy drones. The function takes in the groups of enemy units that exist at the combatted node and flattens the individual drones so that they can be ordered in the same list. The drones’ group ID is stored as a field of the drone so that the targeting system will be able to know which group the drone belongs to.

During our testing, we found there were two scenarios that could occur. The attacking drones could target the least indexed drone which would be the most viable target until it is destroyed, then move on to the next drone, and so on. Or the attacking drones could attack each opposing drone in the list in the order of least index.

Each targeting design had a different problem when implemented. The first targeting method has all attacking drones synthesize their damage to the most viable drones would mean that gameplay would go on for a very long time with each team at

most, destroying 2 opposing drones at a time. Though the most viable targets would be guaranteed to be eliminated, this method created slow gameplay.

The second targeting functions drawback was that there would be common scenarios of each attacking drone targeting all of the opposing drones and ergo have no sense of prioritization. In some cases, if the number of attacking drones were less than the number of opposing drones, then the most viable would be targeted, but not likely eliminated.

Due to the drawbacks of these two methods, we decided to incorporate another parameter for the targeting functions. By including the node, itself, multiple values can be used for calculating things such as damage and defense bonuses. These calculations were done in another helper function named “predictHealth.”

By calculating damage, it can be easier for the attacking drones to see the predicted health of the drones so that they may target another drone without overkilling. When iterating through each enemy drone, if the enemy at the present index has a predicted attribute value that rivals that of the largest found thus far, it will target that enemy.

Of course, the primary goal of creating such a system was to incorporate AI utilization of the system. Since Project Everglades is primarily concerned about its AI players ensuring that this system is usable by AI was a necessity. The custom AI targeting function would be found in the random_actions.py file, inside of the agents folder.

When calling the custom targeting function, the function inside of the targeting.py is tasked with ensuring that the custom targeting function returns valid data, and that the AI script does not abuse the information given.

To prevent the custom targeting function from providing their own damage calculation that could be based on any number they see fit, the return value is a list of tuples with the drone itself rather than an integer value. The reason for this is that the AI’s custom targeting script may put any value inside the tuples. For damage they could put a thousand and in consequence instantly eliminate every opponent drone.

Now, by providing the attacking drone’s self-object, it is also possible to check that the custom targeting function is not attacking more times than the number of drones they currently have at disposal. The way that we check this is by providing a copy of the list of attacking drones that were provided to the targeting function as a point of reference. Once the custom targeting function is called and its resulting list of tuples are assigned to a variable, the list is iterated to check each tuple.

The evaluation of the tuple’s validity occurs in a try, except block. First the drone that exists in the tuple is used as a reference to remove from the list of attacking drones. If the list of tuples includes a drone attacking twice, the second attack would not complete because that drone was already removed from the list of attacking drones.

Once it is confirmed that the attacking drone does exist in the original information, then its damage is calculated based upon its drone type and other factors that may arise. A new tuple is then created with the same information provided by the custom targeting function; however, the last element is the calculated damage rather than the attacking drone itself. The new tuple is then appended to the list of combat actions that was provided as a parameter.

If at any point the validation check produces an error, a message is printed that states that the damage provided was not valid. This of course would only happen for three reasons; The first being that the attacking unit provided in the tuple was not valid. The second is that the damage could not be found from the attacking unit's unit type. The last reasoning is that the formatted tuple could not be appended to the list of combat actions.

To prevent the custom function from changing the data, copies of the parameters are put in so that the custom function cannot alter the original data. This is thanks in part to python's utilization of pass-by-reference variables.

Gameplay Implications

Without drone targeting, the strategy required for the placement of drones on the gameboard is trivial. If a player ensures that they have enough drones placed in the right control points, then they will likely achieve victory. In this way, the current game is much like chess or checkers. While there is strategy involved, all that is required is that the player places their drones in the right areas to block the enemy from proceeding, while also diminishing the enemy resources to achieve victory faster.

Once drone targeting was implemented, it put more importance on what a squadron is comprised of and where it is placed on the gameboard. Players would have to first decide how to create their squadrons, deciding how the squadron will operate within a node based on enemy drone attributes. Will they attack the strongest of the weakest?

With a diverse number of squadrons that focus on different enemy types would come a new layer of strategy, as players would have to carefully position their squadrons to engage enemies in such a way that it gives them an advantage. It would no longer suffice to ensure that squadrons are in the right place at the right time; now, you would need to have the *right squadron* in the right place at the right time. Having a squadron that specializes in a weakest-first approach in a node filled with very strong drones could lead to a crushing defeat, which would force players to ensure their squadrons are positioned intelligently.

System Implications

Of course, creating such a system would be useless if there were no way to have it implemented within the overall project. The targeting prioritization would have to operate in conjunction with either the drones themselves or the overarching squadrons.

Our team originally thought that having the system operate drone-by-drone would be the most realistic and provide the most amount of strategical thinking. This would allow for multiple drones of different targeting types to occupy a single squadron, making the squadron more robust. This approach would still allow players to create squadrons with only a single targeting type, should they so desire.

If the system operated based solely on squadrons, then the overall game would be made simpler, which is not an inherently bad thing. It would be much clearer to the player what strengths a given squadron has and how to use that squadron effectively. The only downside, of course, would be that the power to make a diverse squadron filled with different targeting styles would not be possible.

The issue with both implementations comes from the previously hard coded nature of drones and squadrons themselves. So, if either of the implementations were chosen, the code base would have had to be modified in such a way that allowed for drones to intelligently target enemies, either based on their own targeting attribute or their squadron's targeting attribute.

We initially thought that the best option would have been to have a hybrid of both implementations – giving unique targeting systems to sub-groups within each squadron. In Project Everglades, each team has a variety of squadrons at their disposal, which are comprised of drones of different unit types. There used to exist "sub-groups" within the squadrons that were groups consisting of only one type of drone.

Basing the targeting system off of the sub-groups would have worked best within the greater system. This implementation would have required the least amount of refactoring to become viable.

The only other implication that we conceived on the system came with the AI players, as they would have to be able to intelligently interact with the options that come with picking the targeting system they desire. While AI could pick their targeting system assignments randomly, it would be in their best interests if they chose targeting systems that work in conjunction with one another. In short, they would need to be able to cleverly develop their own strategies when it comes to picking what types of targeting systems their drones utilize.

Of course, the easiest solution to skirt around most of these issues would have been to take such a choice away from the player entirely and have each drone type use a specific targeting system. For example, instead of having players choose what targeting system a striker unit operates on, all striker units would only ever target lowest-health

enemies first. This approach could still lead to a decent level of strategic thinking while still satisfying the requirement, though it certainly does not allow for the same levels of strategy mentioned earlier.

```
# EvgUnitDefinition
**The class of the Everglades drone unit definition.**

## Variables
|Variable       |Type       |Definition                                                                |
|---------------|-----------|--------------------------------------------------------------------------|
|**unitType**   |*string*   |The name of the type of unit.                                             |
|**health**     |*int*      |The initial health of this unit type.                                     |
|**damage**     |*int*      |The damage of this unit type.                                             |
|**speed**      |*int*      |The speed of this unit type.                                              |
|**control**    |*int*      |The control of this unit type. This affects capture speed.                |
|**cost**       |*int*      |The cost of this unit type. The current cost of all unit types is 1.      |
```

Figure 86: A full list of possible attributes.

Lockheed Martin clarified however that they wanted us to utilize targeting in the scope of all drones of a player. This made things much simpler where all that was required was targeting to occur twice during combat; Once for each player to target all enemy drones that exist on the node.

The main functioning of the targeting system was designed so that the enemy drones would be put in a list and sorted so that the most viable drone that fits the targeting priority would be chosen at the smallest index. If there are multiple drones that have the same evaluated attributes, such as health or armor, then they are sorted randomly.

Previous Implementation

As stated previously, drones in the original implementation drones did not exhibit intelligent targeting behaviors. Instead, attacks were processed at each turn at every given node, with damage, health, and node attributes being brought into consideration to calculate the overall damage dealt to each individual drone in all squadrons present at the node.

The targeting in the current implementation was entirely random.

This section will cover the entirety of the original targeting implementation, as there is a lack of sufficient documentation on the subject. Since the new implementation that this group introduced relied heavily on the code originally in the project, it is vitally important that it is detailed in full.

To start, it is important to note that the entirety of the targeting is contained squarely within the *combat()* function. Understanding how this function operates is important to understand how targeting currently operates and will eventually operate. The *combat()* function does not only contain targeting, however, as that is just one part of it. It also contains processes by which the Python server would determine which drones are at the

node, which drones are available for combat, and how much damage is dealt and to what drones.

Now, when taking a look through the function, the first process that is executed combs through all nodes on the gameboard. This process is undertaken every turn of the game, as the *combat()* function is called after every turn to calculate targeting, damage, and death. The reason the function must iterate through all the nodes is simple – it has to look for all nodes which are currently being contested. In other words, it has to find nodes that contain one or more squadrons from both teams. Whenever this is the case, combat occurs because the two enemies would naturally start fighting with one another.

Once a node has been found with more than one team present at it, more calculations can take place. The first counts up the number of sub-groups at the node available for combat. An important distinction is made here at the start: squadrons that are currently moving to or away from the node - i.e. they are currently not present at the node – are not counted as being available for combat.

Pictured below is the series of loops which determines what drones are available to take part in combat at the given node. The first loop will go through all sub-groups in a given squadron, with the inner loop going through each unit within that given sub-group. It is important to note at this stage that the variable member “groups” displayed in this code snippet and future ones is referring to the sub-groups talked about previously. They are groups of one type of drone.

Once a sub-group is selected, the *unitHealth* attribute is accessed, which contains the health of all units of that one type in the given squadron. If the unit is alive, then it is added to a counts array which tracks what units in what sub-groups are available for combat, as well as incrementing an overall count to represent the full amount of force a player has at this node.

```
for i, unit in enumerate(self.players[player].groups[gid].units):
       for j in range(len(unit.unitHealth)):
           if unit.unitHealth[j] > 0 :
              counts_units[player][gid].append(i)
              count += 1

counts[player].append(count)
```

Figure 87: Code that adds active units

Once all the viable units have been counted for, damage can begin to be built. A dictionary called *infliction* is created for the purpose of calculating how much damage will get applied to certain drones, pairing a unit ID with a total amount of damage to be dealt as an integer.

A short array containing the two player IDs in the game is created and the looped through to gain access to the squadrons and units contained within. At the start of the loop, a reference to the opposing player's ID is found, which is used to get references to enemy units in the node.

Once all of this has been set up, the actual damage can be "built", or in other words, the amount of raw damage that will be applied to each opposing enemy drone will be tallied. As can be seen in the following code snippet, the first steps are to get an index for the current desired drone on the current player's team that will do the attacking, along with a reference which states what type of drone is doing the attacking. Once this is chosen, an enemy drone from the enemy team is chosen at random, regardless of type, squadron, or sub-group. This line is where targeting occurs.

Previously, targeting in Project Everglades was nothing more than a single line which chooses a random integer to serve as an identifier for the targeted drone. This unit ID – *uid* – has a value between zero and the total number of enemy units currently at the node (exclusive). This single line was where we substituted out deterministic targeting system.

Once the target has been selected, the damage has to be built for that target. This is where the important *infliction* dictionary comes into play, which stores an integer indicative of the amount of damage that is to be dealt to a certain enemy. If the unit already exists in the dictionary, then the amount of damage to be applied to that enemy is simply added upon, otherwise the value there is initialized to whatever damage the attacking unit deals. Remember that the base amount of damage the unit deals is dependent on the type of unit it is.

```
for i, gid in enumerate(player_gids[pid]):
      nulled_ids[pid][i] = []
      for j in range(counts[pid][i]):
            unittype_idx = counts_units[pid][gid][j]
            unittype = self.players[pid].groups[gid].units[unittype_idx]
            uid = np.random.randint(opp_player_units)
            if uid in infliction[pid]:
               infliction[pid][uid] += unittype.definition.damage
            else:
               infliction[pid][uid] = unittype.definition.damage
```

Figure 88: The infliction dictionary in practice.

Though this is the one line which determines targeting currently, it is still important to understand how the rest of the function works, as whole purpose of updating the targeting system is to have drones intelligently apply damage. Understanding how they apply the damage that is built at this stage is pertinent to creating a viable targeting system.

Furthermore, this random selection of targets means that it is possible that some units escape combat without having any damage applied to them, while others have an inordinate amount of damage dealt. The new targeting system would likely distribute damage a bit more evenly across units, as this system does not check if a drone is applying damage to a unit that has recently been killed.

Once all targets have been decided, the actual damage needs to be applied. This can be a little bit confusing at first to some, as most will think that “building” damage is the same as applying it. The step which just occurred calculated a base amount of damage that will be assigned to various enemy drones, but it has yet to apply it and deal the damage. Think of it like all the drones at the node have chosen their target and have fired their weapons; the last step now is to calculate what happens when the bullets hit their mark.

Note in this analogy that *none* of the bullets have hit their targets. This is currently the case in the code as well, as the function does not process damage drone-by-drone, but rather the damage is built all at once and applied all at once. This means that every drone will have a chance to fire their weapons before any damage is applied to them.

At this point, the function has obtained a reference to the drone it wants to apply damage to from the *infliction* dictionary, and it will attempt to find the exact index of that target drone from the *counts* array which was specified at the start of the function. It does this by starting at the highest position in the *counts* array and moving down through each sub-group added to the array from the start.

If the value at the given index from the *counts* array is too large, this means that the function has not found its target. It will then decrement the index and increment the target group index in hopes of finding the correct drone.

Once the correct drone is found, the following calculation is fairly straightforward. First, the actual damage is calculated. The damage that was built previously was just the raw damage a unit of a given type applies to other drones; there are multiple factors that are taken into consideration in calculating the actual amount of damage dealt. The following formula is used for this calculation:

$$\text{damage dealt} = \frac{10 * \text{raw damage}}{(\text{target health} + \text{fort bonus}) * \text{node defense}}$$

The amount of raw damage is multiplied by ten to make it more weighted at first, as it will then be reduced by several external factors. The first factor to take into consideration is a fort bonus. If there is a fort at the current node, and if that node is controlled by the enemy, their units gain a health bonus. Plus, all nodes have an inherent defense value which further enhances this value for whoever controls the node.

After this is calculated, the damage can finally be applied to its target, which is as simple as subtracting the amount of true damage from the target's health. Should the target's health drop below zero, it is removed from play by adding it to the *nulled_ids* dictionary and updating relevant attributes in the squadron and sub-group it was a part of.

With that, the *combat()* function has been explored in full, explaining the most pertinent aspects of its functionality. In short, the function looks through all the nodes on the gameboard, finds the nodes that are contested, picks random targets for each of the drones, calculates the damage applied to those targets, and removes dead drones from relevant groups.

There were a fair, few problems with that implementation that had to be patched in order for the new targeting requirement to be met. The most glaring change that had to happen was the replacement of the random targeting behavior outlined previously.

One difficult, persisting issue was the convoluted nature of many systems within this function, as well as the hard-coded status of some pre-existing definitions, such as the drone groups. This lead to some difficulty in trying to elegantly integrate new systems into the existing one, as some odd choices were made in storing drone groups. For instance, instead of storing a reference to drones themselves inside a sub-group, the sub-group only stores the drones' health. Nonetheless, we persevered and uprooted the hardcoded design.

Targeting Statistics

As we approached the final presentation deadline, we realized that we needed a good way to showcase our development with the targeting requirement. The Unreal portion of Everglades is currently not equipped to deal with visualizing combat, nor would it be a good indication of units intelligently targeting other units. Showing code snippets is not much of a demonstration and showing raw output to a console is not ideal either. Thus, the best choice we ended up going with to show off our targeting development was to run statistics on the various targeting systems we created.

While showing raw output in the console seems it could be a viable solution to showcasing the targeting requirement, the number of units that are affected by combat each turn is usually fairly great. The console would quickly become cluttered with print statements stating which unit is targeting which enemy and for what reason. It is feasible, but it would take considerable explanation and would not give an immediate indication of the targeting systems working.

The statistical approach – even in a simplified capacity – allows us to create graphs that can visually show that our implementation is affecting combat in a considerable way.

Whether combat is improved or hindered as a result of the targeting systems is irrelevant, as the only goal with targeting is to force drones to exhibit intelligent selection in their selection. So long as there is a difference between our systems and the default random selection targeting, it is easily provable that our targeting systems are effectual.

Approach

With a means of showcasing our targeting systems selected, we began to work on collecting the necessary data. A special script was written – *target_testing.py* – which autonomously runs *testbattle.py* the specified number of times. While it does so, data is collected on the amount of damage done, the number of units killed, and the number of groups killed by the specified targeting systems. With the test script, different targeting systems for Player 0 and Player 1 can also be specified to compare two different targeting systems.

For this to work, the *combat()* function had to be modified slightly to collect the necessary data. It now contains a Boolean variable – *runTargetingStatistics* – which is set to False by default. To collect targeting data, this Boolean must be set to True and then the test script can be run. When the Boolean is set to False, every area where the *combat()* function collects combat data will be ignored, and vice versa if it is set to True. The *combat()* function will collect all data pertaining to the amount of damage dealt that turn, the number of enemy units killed that turn, and the number of enemy groups disbanded that turn.

While the telemetry data could have been used to gain some statistical insight into the targeting systems, it only outputs changes in the state of a group's units, thus it would not provide a wealth of information that could be used for statistical means. Scraping the data as the program runs live and outputting it into a file for storage was the best way of accomplishing this task.

Once all of the specified iterations are finished running, the test script will call the function, *processData()* – which exists in the *target_testing.py* file – which reads through the output file where all the data from the *combat()* function was stored. It reads in this output file as a .csv file and uses the *pandas* library to process it. *pandas* is a very handy library for processing .csv files and taking various statistics, such as the average of an entire column, the standard deviation, and other useful statistical functions. Once the output file is read in as a .csv file, the *pandas* library is directed to process statistics from Player 0 and Player 1 for the amount of damage dealt per turn, the number of enemy units killed per turn, and the number of enemy groups disbanded per turn.

For each of these categories, it will take the mean, mode, median, minimum number, and maximum number. The reason these statistical methods were chosen is to keep things simple while also providing enough information to give a solid understanding

of how the targeting systems relate to one another. The student's t-test was considered as a statistical method, though it did not give an immediate understanding of how the targeting systems compared to one another. Hence, using averages across multiple games as considered to be the best method of showcasing the differences.

Statistics

The following are three separate bar graphs which showcase the average damage dealt per turn, the average amount of enemy units killed per turn, and the average amount of enemy groups disbanded per turn. Player 0 is shown in blue, while Player 1 is shown is red. The targeting systems chosen is shown beneath the bars, with the first targeting system listed pertaining to Player 0's targeting system and the second system being Player 1's targeting system. For example, the bar for "Highest vs Lowest" means Player 0 used the "highest health" targeting system, and Player 1 used the "lowest health" target system.

These graphs show every possible combination of targeting systems contending against other targeting systems. The player that utilized a target system does not matter, because both players used the exact same group and unit composition on a fair three-by-three gameboard. This means we did not have to run a test with Highest vs Lowest and then run another test with Lowest vs Highest, because the end result would have been virtually the same, with some negligible differences.

For all of the statistics shown, the targeting configuration was run on 500 separate games. As stated, both players had the exact same group and unit configurations. All games were played on a randomly generated three-by-three two-dimensional map. These maps are always guaranteed to be fair, so no player gets more nodes, and a new map was generated each time the test script ran a new game. The *random_actions* agent script was used to power the decisions of the players, so while it is possible that combat never occurred during one of these games, it is highly unlikely on a map as small as the ones used for testing.

The following table reiterates the targeting systems we created so that the following data is contextualized:

Current Targeting System Descriptions

Targeting System Name	Abbreviated Name	Description
Randomly Select	Random	Picks a random group and unit to target.
Lowest Health	Lowest	Targets units with the lowest current health first.
Highest Health	Highest	Targets units with the highest current health first.
Most Lethal	Lethal	Targets units with the highest health and highest base damage first.

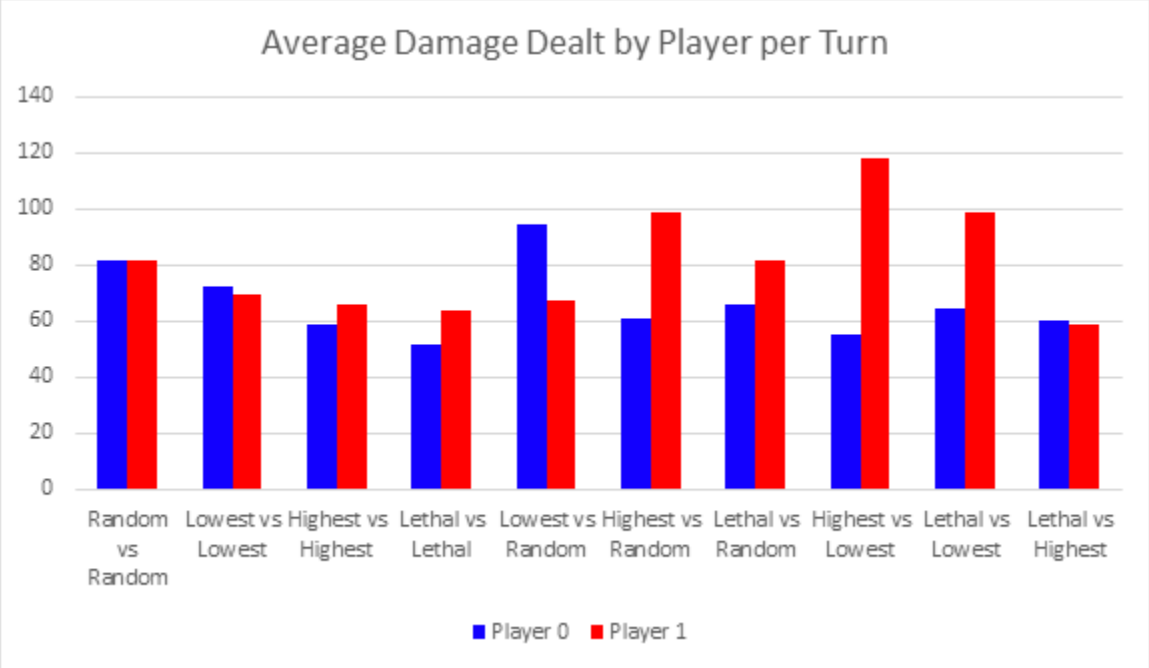


Figure 89: Average damage dealt by player per turn.

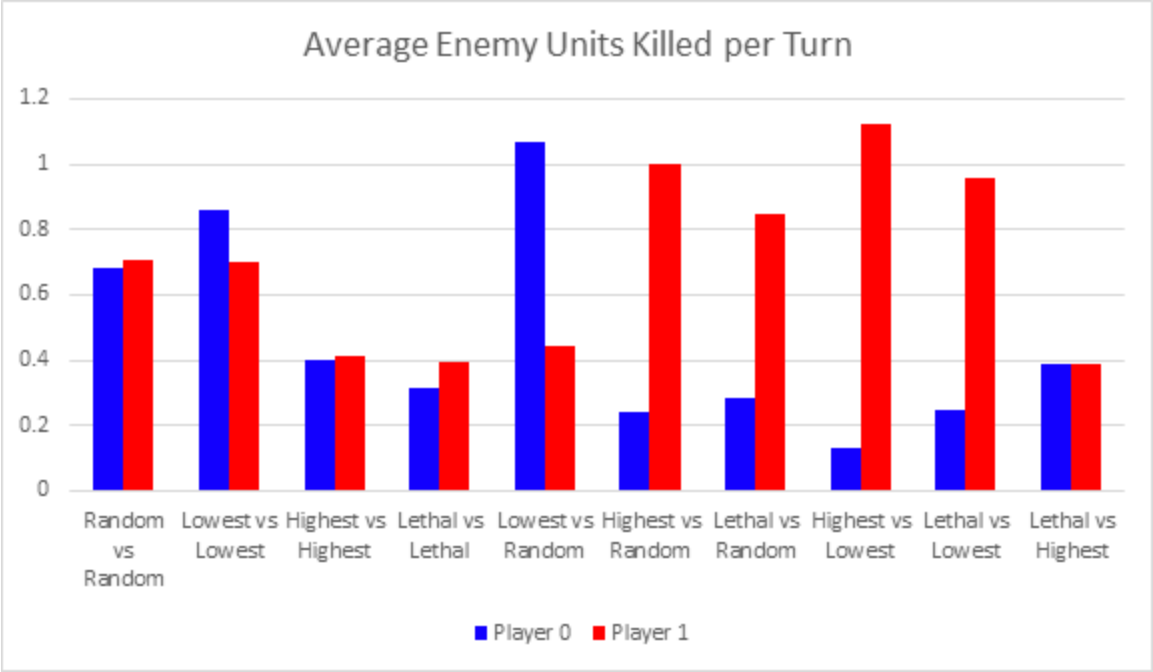


Figure 90: Average enemy units killed per turn.

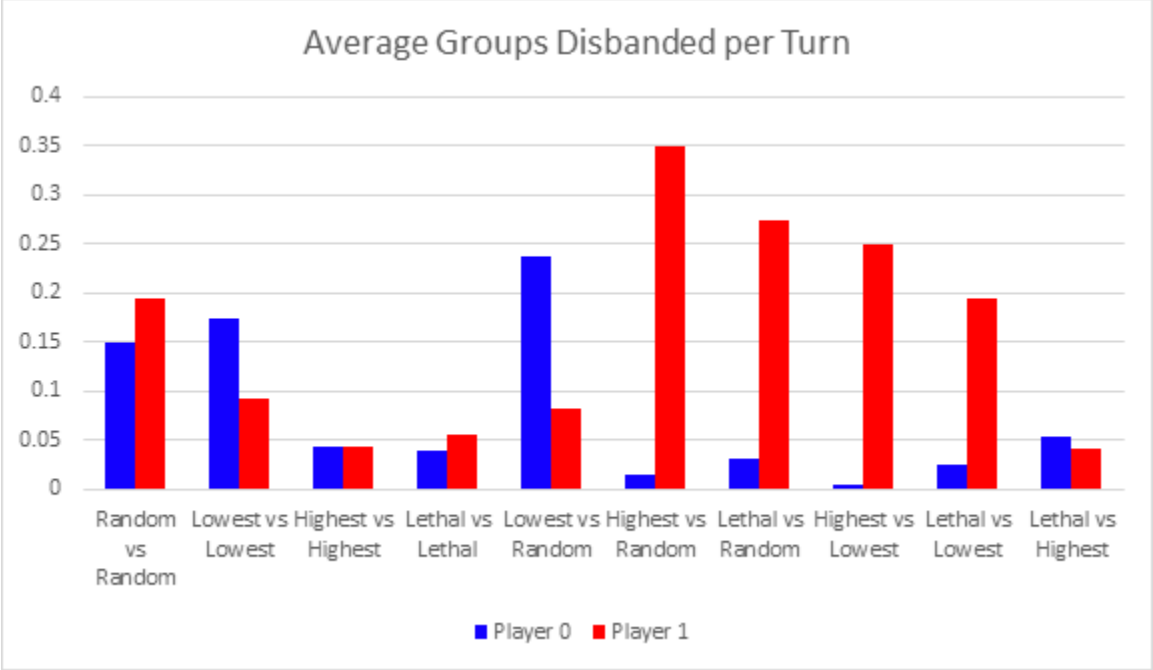


Figure 91: Average groups disbanded per turn.

Interpretation

In this section, we will briefly discuss the findings from the data collection.

Unsurprisingly, the differences between any targeting system against itself are very small across all three graphs. It should be noted that the “Average Groups Disbanded per Turn” graph operates on a very small scale, as no single targeting system is capable of disbanding an entire enemy group on a single turn; the highest value does not even reach half a group disbanded per turn.

Whenever a player opposed a player using the Randomly Select targeting system – unless they, too, were using Randomly Select – they were most assuredly defeated in all three areas. This is not the case if the player chose Lowest Health, though, which dominates over Randomly Select in all three areas.

We believe the reason for Randomly Select’s dominance over Highest Health and Most Lethal is due to its tendency to stack damage on a single unit at random, accidentally creating coordinated efforts while also having outliers whereby other enemy drones are damaged. Overall, it reduces the combat effectiveness of an enemy group by pure random chance. Highest Health and Most Lethal and very particular about which units to target; targeting by highest health – regardless of additional qualifiers – seems to be a poor strategy as it does not stack damage on a single unit.

Lowest Health likely dominates most other targeting systems in all three areas because it is effective at ensuring units die as quickly as possible. It kicks an enemy unit while it is down. When a player targets by highest health first, it is harder to guarantee that a unit will die in a single turn because the amount of firepower required to kill a single unit in a single turn is considerable. Targeting by lowest health first makes the task of killing a unit easier, because it requires less drones to bring the enemy’s health down to zero. Since Lowest Health is so successful in eliminating enemies, it is able to commit more damage per turn as there are less enemies to eliminate their allies, and thus more allies to deal more damage.

Finally, when Most Lethal is pitted against Highest Health, they are effectively equal. This is likely because they both target units by their highest health first, although Most Lethal will also prioritize units which deal the most damage in conjunction with that priority. It would be interesting to see what happens if Most Lethal is changed to target by highest health first or by most damage first, instead of both at the same time.

Electronic Communication

Overview

Laying the foundation for electronic communication between drones was a “home-run” requirement that was set by Lockheed Martin without much detail until the second semester. Our team initially believed that the main purpose of electronic communication was to help provide a more realistic portrayal of the drones. We believed this implementation had to be done so that the drones were essentially utilizing the information around them to the best of their ability.

Though there were no specific examples or guidelines as to what Lockheed Martin wanted in terms of implementing drone communication, one of the main goals for this project was to make the drones have a behavior that is as realistic as to how real-world drones behave as possible. We took this to mean that there should be more added depth to the drones’ decision making when targeting opponents at a contested node. Whether that be done by using already implemented existing decisions or creating new ones.

In the second semester, we received information as to how our sponsors wanted to portray electronic communication. To lay the foundations of electronic communication and electronic warfare, we were tasked with creating a Jammer unit that would slow enemies within range. This would represent the disruptive effects that electronic warfare could impose upon enemy forces.

Nonetheless, through the incorporation of electronic communication for the drones, there is an increase in sophistication for the gameplay. Players and the AI may choose drone loadouts with the specific intent of countering their opponents possible group combination of Jammer units.

Previously, the drones did not have any priority when it came to achieving a tactical advantage to win and capture a contested node. As of last semester, drones in combat did not “target” another drone with any purpose whatsoever. Rather, the drones attacked any randomly selected healthy drone in each round of combat.

To elaborate on the targeting system’s relationship with the drone communication, the targeting system is a separate function of code. The implementation of drone targeting would only influence decisions that are made by the drones already defined priorities that are established in their targeting system.

Battling

As previously stated, in project Everglades the drones initially did not have any focus or decision for attacking. The targeting system was incorporated as it was one of our “nice to have” goals to accomplish. By incorporating drone communication, there can be an added layer of sophistication implemented to the drones’ targeting system as well.

The decision of a drone that is without allies is fundamentally different than one with allies. Without teammates, the only things to worry about in battle are how strong the enemy is and how strong you are in comparison to the enemy. This can be decided based upon more complex knowledge such as the enemy’s current health, the enemies current defense, the enemies current damage, etc. Which is also needed to be compared to your current health, current defense, and current damage, etc.

When in a team setting during a battle however, there are many more things that you need to have to worry about. One variable to take into consideration when in combat at a contesting node is the count of the opposing drones that are present. This variable can help to understand more factors such as by how much the enemy group outnumbers the allied group. Who is the strongest teammate on the enemy team, is the enemy team’s strongest teammate stronger than your strongest teammate? What is the enemy team’s weakest teammate, is the enemy team’s weakest teammate stronger than your weakest teammate?

It can get very convoluted and overtly sophisticated with information such as this. There are roles that teammates may take upon themselves to achieve success. These roles are based upon the strengths and weaknesses of each drone and how well they can synergize their attributes.

For the sake of having a sense of simplicity, the drones can have a limited number of decisions that they can reach. They can be influenced to decide which target to focus on and whether to retreat. Anything more would then deter from the rules of gameplay that are already established, even though the drones are expected to behave as realistic as possible.

For example, a tank drone will most likely be best utilized to draw fire from the enemy team as they are most equipped to handle damage. This would allow for more offensive drones such as drones with a unit type of a striker, to be able to deal large amounts of damage to the enemy team.

One possible way to add more decision making without disrupting the rules of the game would be to implement a sense of strategizing when receiving/delivering fire. To have drones being able to have this ability would possibly provide an even more rich feeling of realism.

Though a tank drone is expected to draw fire from enemy drones, it would not be able to do that as efficiently if the drones with a unit typing of a striker did not stand close

by from behind. This could also be implemented in relation to the drone targeting system as well.

If a group consists of only striker units (which are inherently weak in the aspect of defense, but in contrast are strong in attacking) they may form a different formation from, let us say a group of drones with a unit type of a tank. A group of drones with a unit type of a striker may want to stay at a close distance together so that they can target individual drones one at a time, similar to how a pack of piranhas will target their prey in nature as a group.

A team of only strikers may also want to stay as far apart from each other so that it is harder for the enemy drones at a contested node to be able to target them. This is given from the fact that a bulky drone would have more mass but being lightweight provides less area.

Not only would a group formation work in terms of holding down a line, but it could be more dynamic in terms of continuous movement to prevent getting hurt from enemy fire. This would prove to be the most challenging aspect of all from incorporating drone movement at a node as a decision-making option.

When currently in movement while also targeting enemy drones, would prove to be difficult as there is a need for there to be a reference to the targeted drones' locations. Not only that but the targeted drone may also choose to move to evade getting fired upon.

Giving drones the capability to create formations could also be another area to study as there are already so many complex variables being added, Lockheed Martin could do well to use their reformation as a way to discover better formations outside of drones.

Already there are things that exist in nature that help to develop new technologies. This is especially true in terms of AI as it is able to create incomprehensible amounts of scenarios at an extremely fast rate. Which is already what this project is about but in terms of AI drones strategizing the best course of action. To add on further, implementing drone movement would definitely be a step in the right direction for creating the most advanced, realistic simulation for drone combat.

Currently there is no reference as to the position of drones and how far apart they are and how far away the enemy is. If this could be reworked to get a more lifelike snapshot as to what is going on during battle, implementing a formation could be proven to be easy.

Due to the fact that a team of drones can become somewhat complex, the function for team coordination may have to be implemented in the group code, as well as the code for each drone individually. The reason for this is simply that if each drone were to decide their best action, there may not be any team coordination. In other words, there is no drone communication if not every drone is listening to each other and coming up with the best decision for the team.

The best way to make sure each drone is listening would be to make sure that they all get the same basic information in a group function. The group function would act as a hive mind that delegates possible decisions for each drone inside of a given node. The hive mind would pass on these decisions to each drone so that there would be no miscommunication.

By having the communication coded into each drone, the process of gathering information would be much more efficient. Each drone could consider the enemies and allies around them and help the hive mind with deciding the best decision for the drone itself and for its allies. The following image can best illustrate this dynamic between drones and the hive mind.

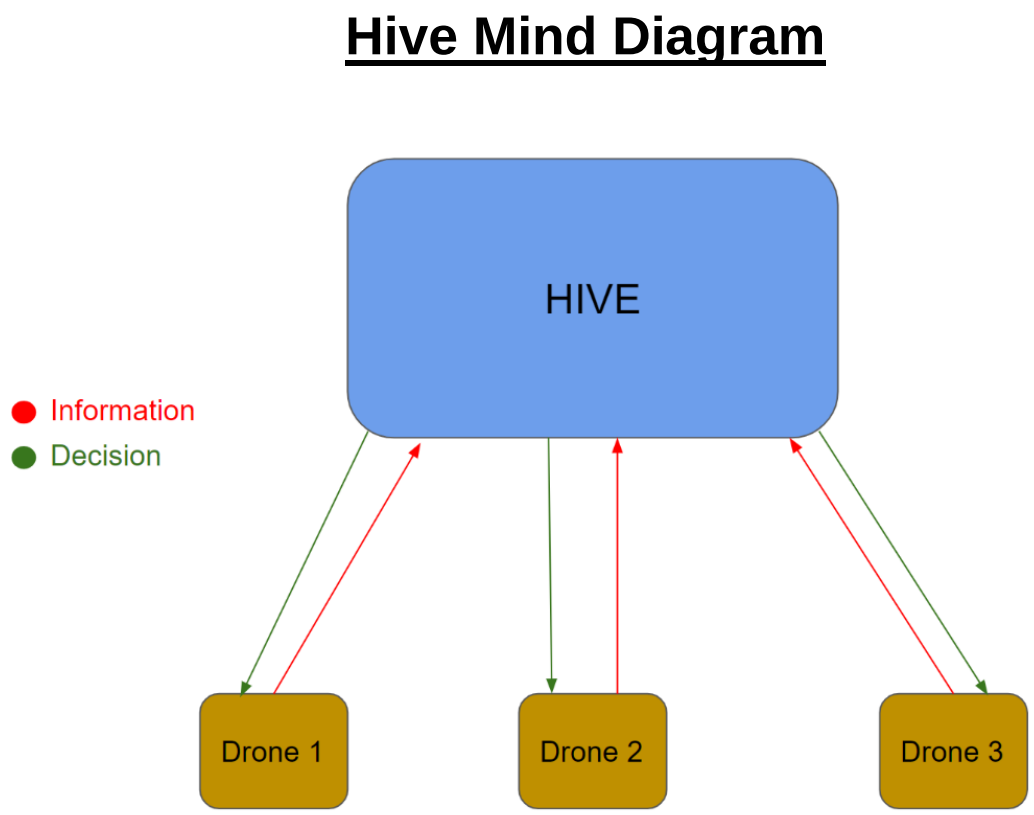


Figure 92: Hive Mind Diagram

To better explain the black box that is the hive mind in the diagram, the hive mind can take in information that each drone relays from what they have sensed about the enemy. From there, the hive mind would not return the best decision for each drone until it has received all the information from each drone.

As it stands right now, the information about the number of drones on each side is already inputted into the node. The hive mind could access this data to get a rough estimate as to whether to flee or not. This information could also be used in conjunction

with other information given from each drone, such as the ally drones' current health and their overall combined attack, defense, and specialty stat.

The specialty stat can be calculated by using an algorithm to see how well their abilities synergize given how healthy and strong the team is. Whether that information can be pulled from the other team as well would be up to future teams to determine.

The hive mind would not be an actual entity present in the unreal engine, but a function to help run the electronic communication between ally drones at a node. The drones themselves would most likely relay their data through the reference of their group.

There would have to be two plugins for a hive mind for each side present at each node. Each hive mind would have to dynamically get a reference to each ally drone present through their group code so that it can provide the best decision for each drone and help to coordinate decisions. This would work well with multiple groups if the information is passed to the function as an array or list of groups.

A possible addon would maybe be the option for a drone to "tap in" to the enemy hive mind to gather data as one might due when intercepting a frequency radio. This is a realistic approach that could be looked into further down the road of development. It would most likely be an ability that can be chosen and applied to in the drone unit creator.

There may be some limits that would have to be applied to the communication interceptor feature if it is too powerful, such as a cooldown for activation. Another limit may be how much and what information the interceptor drone may be allowed to have access to. There may also be a limit as to the number of drones that can have this ability in a group, or as a whole being as if all drones are chosen to have this ability, the other team would be at a constant disadvantage.

In summary, the purpose of a hive mind would be to better delegate the best decisions as well as to keep all ally drones inside of a given node on the same page. There may be a chance that each drone may not make the best decision for the team, but for itself. Or there may be a bug that causes the drone to crash and render it useless or even worse, counterproductive.

Flee

When the chance of success seems unachievable, then the drones can have the failsafe option to flee to a nearby node. The reason for fleeing would be to help protect the player from losing due to complete annihilation of their drones. A tactical retreat could either be implemented at any given time or only at the start of a contested node.

There are a multitude of variables that can be used to influence the decision to retreat. One such variable is the number of enemies in comparison to the number of allies in a given node. Obviously, there is strength in numbers so being on the team with the least number of drones is a disadvantage.

Another such variable would be the health of the enemy and also ally drones. This information can be picked up from the drones' sensors also relayed to the decision function. There may need to be sacrifices to capture a node and also the drones or player must decide whether that sacrifice will be beneficial for achieving a win.

All these variables are things that can be gathered from each drone and also relayed to the hive mind so that a consensus can be made for the possibility of retreating. This consensus can be called through a function that locates the nearest nodes that are safe.

To determine whether a node is safe would most likely have to be determined by whether the node is occupied by an enemy group. It could also be determined if the enemy group of drones is not as strong as the retreating drones. This would definitely be able to add more sophistication to strategizing between the drones present in the game world.

To determine if the enemy group at a node considered for retreat is safer to contest than the current one would most likely be easy to implement. It would ultimately depend on the functionality of the hive mind. The major concern would be how to get the retreating node's hive mind to relay information to the current one.

This is a bit of a gray area as to whether the retreating drones would be able to actually see enemy drones at a node for possible retreat. The reason for this would be dependent on how perceptive the drones will be in their sensors. Another reason it is a gray area is due to the fact of how long it would realistically take for a drone to be able to gather enough information from such a great distance from another drone.

What is more questionable is whether the retreating drones would be able to calculate the danger of retreating to a node occupied by enemy drones. It is one again ultimately up to the perceptive range of the drones, as well as how accurate they can be from long range.

When all conditions are met to determine a retreat is necessary, the function can mark a Boolean retreat trigger as true, so that the drones can successfully and also safely retreat to a neighboring node. An example as to how this might be implemented in code is shown in the following image.

```
// Function present at every node that takes in list of drones with their gathered information
Hive(List<Drones> drones)
{
        // Determine whether to retreat based on visible difference in strength between teams
        bool retreat = compareTeams(drones, enemy);

        // If drones should retreat, find nearest node and tell drones to retreat there
        if (retreat)
        {
                // Calculates best nearby node to retreat to and tells drones to move there
                newNode = getNearestNode(currentLocation);
                Retreat(drones, newNode);
                return;
        }

        // The best course of action for each drone will be assigned based on the variables present
        assignRoles(drones, variables)
        return;
}
```

Figure 93: Hive mind pseudocode

The function would most likely run much more efficiently if it could predict beforehand whether the battle will be lost. This could add more sophistication to the drones' behavior and also create more complex decisions, such as deciding to retreat for the purpose of grouping up with an ally group nearby. This would help ensure victory in the sense of strength in numbers.

If the drones could retreat before battle has started, then the gameplay would become more closely related to that of chess. This is due to the fact that if drones were to repeatedly retreat if they knew that they would not be able to succeed. With this "auto-retreat" existing in the game, it may end up being a chase around the map until the losing team is cornered.

Limitations

There will be some limitations to add a more sophisticated and also realistic model of what would happen in the real world. This will be brought about in two ways: a limited range for how far each drone can communicate with other drones, and also having each drone have a vulnerability to disruption in the communication between each other.

Range

Background

In the real world, drones can have a variation for the possible distance of communication, whether that is between drones or a drone and a controller. The most common frequencies that are currently used for drone communication are 2.4 Ghz and 900 Mhz. More common household drones have a frequency of 2.4 Ghz whereas 900 Mhz is for more professional and durable drones.

Figure 94: ATOYX-Quadcopter [10]

A drone with 2.4 Ghz, such as the quadcopter shown above, has a relatively short distance, with a range of about one mile for communication. Given a drone that has a 2.4

Ghz frequency, this also means that the drone will not do well in an area where objects can come between it and the other end of the communication.

A drone with a frequency of 900 Mhz can reach a much farther distance than a drone with 2.4 Ghz frequency, being able to reach a distance of over 60 miles in range . A drone with this frequency will not have any problem if objects intercept the path of the electronic communication. The following is a freefly quadcopter drone, which is primarily used for cinematography and photography in an outdoor setting. This type of drone is an example of one that uses a frequency of 900 Mhz.

Figure 95: Freefly Alta X Drone [7]

A graph would be best to exemplify the changes in these types of frequencies. The following images show the movement of a 900 Mhz Sine Wave and a 2.4 Ghz Sine Wave over time to help deliver a better understanding.

900 Mhz Sine Wave

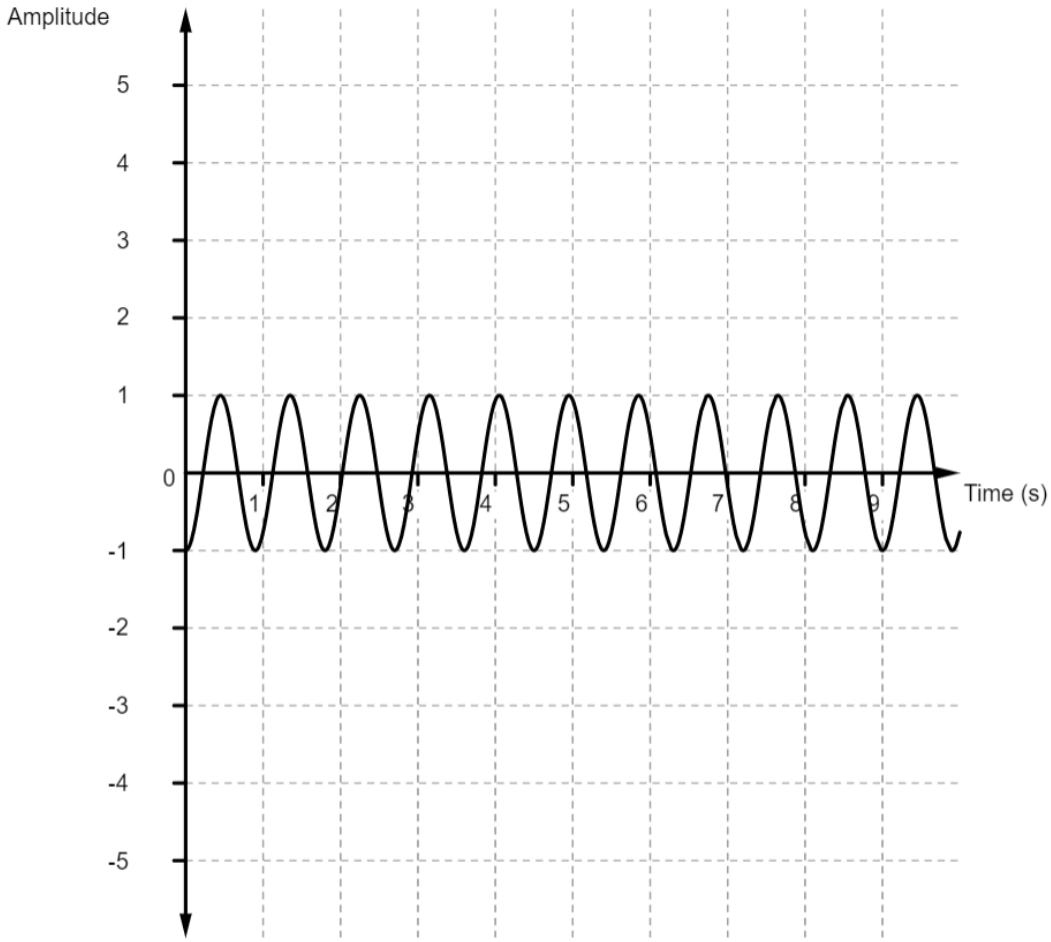


Figure 96: 900 Mhz Sine Wave

2.4 Ghz Sine Wave

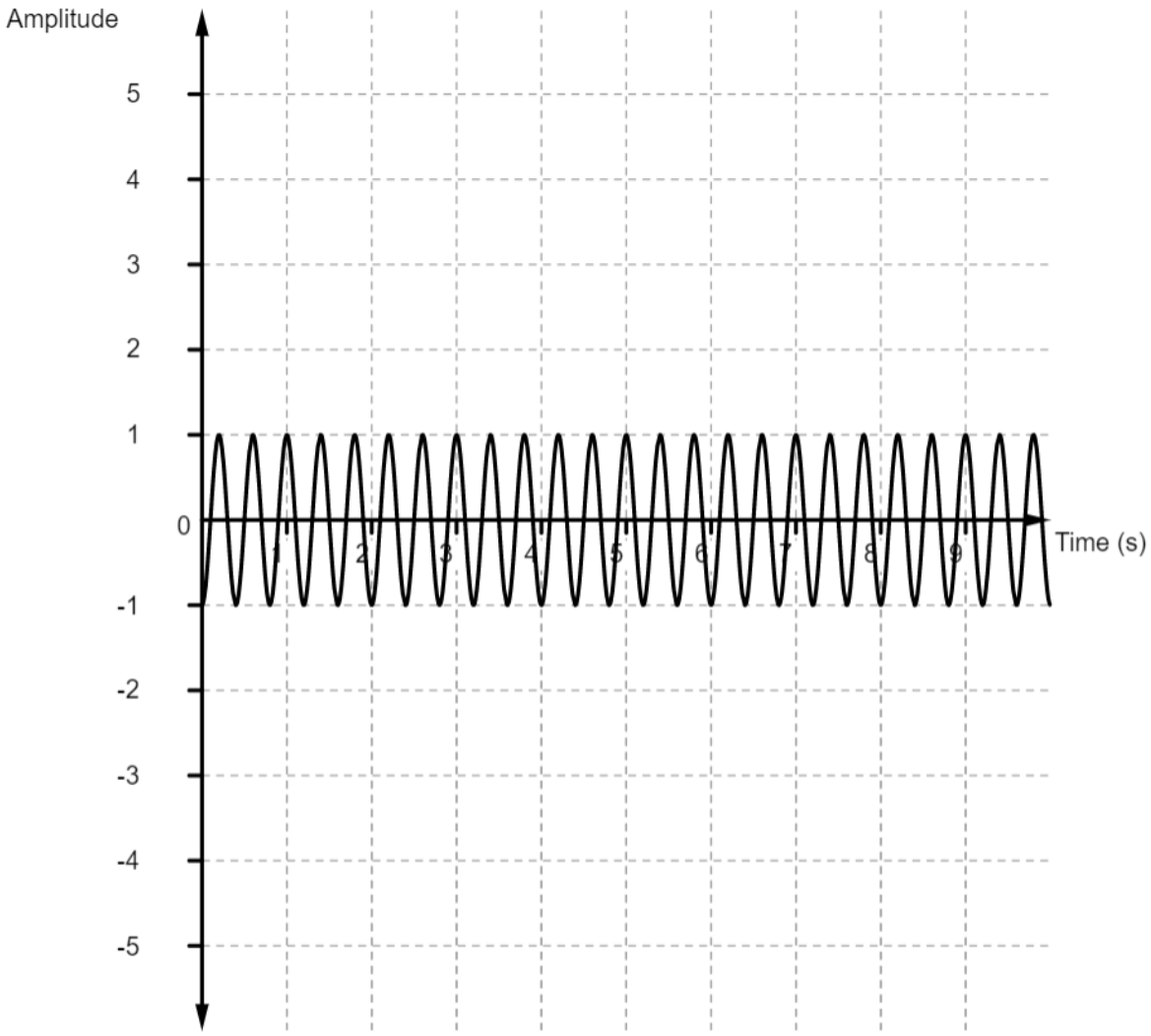


Figure 97: 2.4 Ghz Sine Wave

The reason for the difference in range and penetration from the 900 Mhz frequency and the 2.4 Ghz frequency is evident in the preceding graphs. The 900 Mhz causes a longer wavelength which means it can reach a farther distance without using up too much energy. In contrast, the 2.4 Ghz wavelength is much shorter, ergo it can only have a short distance in comparison to the 900 Mhz frequency.

Game Implementation

The drones in this game most resemble those in real life that have a frequency of around 2.4 Ghz. The reason for this is that we want to localize effects in the same node and contain calculations so that buffs and debuffs are widespread. The distance between drones that are located inside of a given node are less than a mile. There are not any obstacles present in the game that will come between the drones' communication so there was no need to incorporate a low frequency.

Having the drones being able to communicate with drones that are located in other nodes is not necessarily unrealistic, however a shorter range of communication is more suited for the way that this game is played.

There is a possibility to implement an ability for selected drones created in the unit creator. This ability could help to relay information to ally drones that are a node away to better strategize in terms of retreating for the purpose of regrouping.

For example, say a group of ally drones are outmatched by one on a contested node. The ally drones on the adjacent node however are more than capable to take on the enemy drones in that node. With the ability to "call out for help" the strong team of ally drones could decide to regroup with the team of ally drones that need their help.

EMP

Background

The only implication that was made was having the drones vulnerable to communication jams that occur when jammer drones are present. This feature added more complexity to the drones' group composition, as jammer units slow down enemy units. For future implications, this could impact decision making in an effort to make the drones' behavior more realistic. An electromagnetic pulse (EMP), which is also sometimes referred to as a transient electromagnetic disturbance, is a short burst of electromagnetic energy.

An EMP can be present as a radiated, electric, or magnetic field or as a conducted electric current. There are two ways that an EMP can be created, it may be either from a natural occurrence or it can be brought about through man-made technology. For the purpose of the project, we planned on there being only man-made EMP attacks. Though there is a possibility of a natural occurring one later on when the environment has more of an impact on the gameplay.

A lightning strike is an example of a natural occurrence of an EMP, where it emits high bursts of electricity when it strikes. The effect can be better understood in the

scenario where all electronic appliances in a house are shut down, however they soon restart given some time.

If weather were to have more of an effect on the gameplay in future implementations, this could be one way in which the weather could be used to add more hazards for the drones. There could be random chances for lightning to strike based on certain variables that would deter the AI or player from moving drones to higher elevations.

EMP interference is more often used in the military due to the fact that it can cause possible serious harm to others. For the purpose of this project, our team planned on using EMP attacks that could be carried out by the military on a small scale. An example of an EMP created by the military is the nuclear bomb. Back in 1962, there was a nuclear test called the Starfish Prime test. In the test, a 1.4 megaton bomb was detonated above the Pacific Ocean.

The explosion caused damage to electrical equipment that was more than 800 miles away. It was not due to the physical shockwaves themselves however, but rather it was due to the high concentration of energy that was emitted from the explosion. This is actually what led to the discovery of EMP.

An EMP is more often than not, used for the disruption or damage to electronic equipment. At a higher level of energy, a powerful EMP such as a lightning strike or atomic bomb, can damage physical objects such as buildings and also aircraft structures.

Game Implementation

The presence of EMP in this project however will be kept at a level of power where it solely disrupts communication between ally drones at a node. There can be some effects implemented due to the disruption of communication that may give an advantage to an enemy team. This advantage however should not be too great of a scale where it is a guaranteed victory for whichever team uses the EMP first.

EMPs can be found in a multitude of video games such as Apex Legends, Deus Ex, Watchdogs, and even Mario Kart. The effects and power of each EMP varies between video games but they all have a similar goal.

Apex Legends

Apex Legends is a first-person shooter, battle royale game developed by Respawn Entertainment and published by Electronic Arts. The main goal of this game is to be the last team standing at the end of the game. There are a multitude of weapons with different class types as well as different characters to choose from. Each character has a unique set of skills and abilities, similar as to how the drones in project Everglades are set up.

There is one character that exists in Apex Legends that utilizes EMP attacks. This character's name is Crypto and he utilizes a personal drone as his ability. This drone acts similarly to the recon drone that exists in the current implementation of project Everglades. The drone in Apex Legends is only allowed to be used for the purpose of providing surveillance and cannot harm others.

There is one exception however as to how the surveillance drone in Apex Legends operates. When a certain meter has reached capacity, (Referred to in the game as the player's Ultimate) then the drone is able to emit an EMP attack. This meter fills up over time and empties when the player uses their character's ultimate ability.

The EMP attack utilized in Apex Legends has a radius of effect of up to 30 meters. Inside of this area, all enemy players are dealt 50 damage to their shields, as well as having their movement slowed. Once the EMP attack takes effect, the players' movements are only slowed for about one to two seconds. [8]

Deus Ex

Deus Ex is a role-playing video game franchise that has been developed and published by multiple game companies. The game utilizes EMP grenades as a source of weaponry to accomplish objectives. The EMP grenade is described in the game as creating a localized pulse that temporarily disables all electronics that are within its area of effect. This also includes cameras and security grids if they are present.

The EMP grenade that exists in Deus Ex has a range of 32 feet and primarily affects enemies that have EMP health points. It does not affect humans as the only enemies that have EMP health points are robots and electronic devices. When a robot takes EMP damage however, they can be disabled and once they reach an EMP health point total of zero, they shut down.

Watchdogs

Another video game that implements EMP weaponry is the Watchdogs series. Watchdogs is an action-adventure game developed and published by Ubisoft. In the third installment and most recent of the series, Watchdogs: Legions, there exists a weapon known as an Electro-Shock Trap. The Electro-Shock Trap has an area of effect that shocks enemies that are close enough in distance.

It does not utilize an EMP until it is upgraded once. Once upgraded, the Electro-Shock Trap can disrupt enemies, including drones. To further improve its effectiveness, the Electro-Shock Trap can also be upgraded a second time to be able to disrupt enemy weapons by causing them to jam.

Mario Kart

Mario Kart is a cartoon racing game developed by a multitude of game companies, but primarily those owned by Nintendo. Mario Kart was also published by multiple companies that were also primarily owned by Nintendo. Though it may sound odd to think that such a cartoonish game would incorporate something as sophisticated as an EMP, Mario Kart does utilize EMPs in a unique manner.

The lightning item when used during a race in Mario Kart causes all opponents to get struck by lighting and consequently shrink in size. Though that does not necessarily happen in real life during an EMP attack such as lighting, the overall effect of shrinking the opponent's does slow their vehicles.

The lightning EMP attack is designed in a more creative and fun manner due to the fact that Nintendo creates games for all ages. For this reason, our team did not intend to cause opposing drones to shrink as this does not necessarily reflect how an EMP attack works in the real world. The scope of this project does not currently have plans for appeasing users of all ages. The effect of slowing down opposing vehicles is something that our team created through jammer units.

The main takeaway from all these incorporations of EMPs in video games would be the following. The purpose for an EMP is to at the very least cause some form of a crippling effect on opposing electronic devices or vehicles. This can be either to stun, disarm, or possibly both. The effect of crippling an enemy drone can range from a short amount of time or become a permanent effect.

All EMPs are not usually found to only cripple a single device at a time in video games. Rather, they have an area of effect that can impact all electronic devices that are within range. In some cases, however, such as Mario Kart, an EMP can affect all opposing vehicles that exist in a game.

EMP Incorporation

Since Lockheed Martin clarified that they wanted us to set up electronic communications, we did not incorporate EMP attacks exactly, but units that slow down enemy units' movement. The design and implementation of EMP attacks will be left up to future contributors.

The original plan of action was to let the EMP be represented as an ability for select drones that are customizable in the unit creator, similar to how Apex Legends utilizes their incorporation of EMP attacks. To have every drone able to cause an EMP attack would be too linear in terms of strategy as there would be no counter for it to cause actual strategizing.

The lethality of an EMP attack was not determined as more research had to go on as to how we wanted to balance our game implementations. Our first thought was to have

EMP attacks cause drones to become stunned. By becoming stunned, the enemy drones could skip their turn. The duration of such an event in which an EMP attack's effects would last could be for a round or up to possibly three rounds.

We also thought that the EMP attacks could also be implemented to cause the enemy drones to have a reduced movement speed similar to how the effect of the EMP in Apex Legends slows enemies. This would add more complexity to strategy by using EMP attacks to prevent enemy drones from following out their plan of movement across the board even if they were to end up failing to capture the contested node.

There was not likely going to be any drone ability present in the game that would be able to counter an EMP that would also be existent in the real world. The only known way to prevent an EMP attack is to be inside of a faraday cage. A faraday cage does not necessarily need to be a cage, but it does need to be a mesh covering made out of conductive material that blocks electromagnetic waves from passing through.

This however is counterproductive as a faraday would also prevent electronic communication from being sent out from a drone, as well as EMP attacks from being received by a drone. Perhaps if more technology were to come out to prevent EMP attacks, they could be implemented into the game. It could be possible to put it in the game as an ability for select drones that are made in the unit creator.

The only possible way of implementing a counter to lessen the effects of an EMP attack inside of the game without changing the structure of how it works, would be by adding a new ability. This ability could be a "Recharge" ability that helps to recover electronic communications between drones at a node at a faster rate.

This Recharge ability could work similar to that of the abilities in Apex Legend by having a cooldown set in place. This could also just be put in as a passive ability that is designated to a specific drone type. This could work as a stacked bonus where the more drones that have this ability, the faster the communications between the drones can be brought back online.

To monitor this "rate of charge" in regard to recovering from an EMP attack, an EMP health point system could be put in place. By adding a measurement of EMP health, it would help to implement an algorithmic function to recover EMP health points through the Recharge ability. It would also be able to apply stacking bonuses as well by increasing the rate of recovery.

To fully recover from an EMP attack, the EMP health would most likely have to reach its full capacity. This could automatically replenish over rounds at a certain rate; however, the rate must not be too fast where the EMP attacks are obsolete. The Replenish ability would simply increase this rate either though an algorithmic process or by a factor increase.

The disruption of the electronic communication between the drones will cause not only the problem for team coordination, but the prevention of utilizing the best decision.

An example can be seen when a group of drones are outnumbered and to preserve the number of allies, the best course of action would be to retreat to another node.

When there is no chance of success, it would be an easy victory for the opponent to take advantage of the opportunity to keep the drones from escaping. This scenario could also be used as a strategy for the player or AI in that they can create drones in the drone selector to have at least one drone in each squad to have an EMP ability. There are other strategies that the AI or even players can come up with to better utilize the EMP attack, which is what future teams will have to research later.

In terms of game balance, the use of EMP attack will be fine-tuned so that it does not cause an unbalance in fairness. The EMP attack will most likely be implemented with a cooldown or limit to the number of drones that can have this ability. There would also most likely have to be a timer that indicates how long the drone's communication will be down for when dealing with an EMP attack.

The effect caused by an EMP attack will at most, only affect drones that are in the same node that the EMP attack occurs in. To have a greater range for an EMP attack would most likely cause too great of an advantage for the user.

From what we accomplished for electronic communication; our Jammer units are the main presence of "EMP" attacks. Their intensity exhibits inly a decrease in movement speed between nodes. There is no damage calculated and their presence alone causes movement speed to decrease for opposing units.

Crucial Changes to Everglades Server

OpenAI Gym

OpenAI Gym is an environment used to train AI agents. Everglades has its own custom-built gym environment so that agents can train against each other to learn. The previous gym-Everglades class that was handed off to us was outdated and needed to be refreshed.

The previous gym configuration for the Everglades game parameters resembled

```
# Game parameters
self.num_turns = 100
self.num_units = 100
self.num_groups = 12
self.num_nodes = 11
self.num_actions_per_turn = 7
```

However, with the introduction of map generation, the num_turns needs to be changed to allow for longer playtime, and num_nodes needs to accurately account for the

dynamic map generation. We use the GameSetup.json to get the turn limit parameter, and the name of the map to count the nodes.

```
# Game parameters
self.num_turns = self.setup['TurnLimit']
self.num_units = 100
self.num_groups = 12
self.num_nodes = len(self.mapfile['nodes'])
self.num_actions_per_turn = 7
```

File Paths

Previously, file paths in the server were hard coded to look for a folder named “Everglades” inside of the C: drive. This has been overhauled so that all file paths are relative so that you can store the Everglades-server folder anywhere on your computer.

Unreal Visualizer

Developing inside of the Unreal portion of Project Everglades was not required for this iteration of the project. Regardless, we found some additional time within our schedule to work on the visualizer and update it to support 3D maps while improving performance and the addition of a minor UI feature.

How to Use the Unreal Visualizer

To edit the Unreal portion, the Unreal Engine will need to be installed on version 4.26.1. When the project is opened, navigate to the folder titled Everglades in the Content Browser to find all the blueprints and scripts that pertain to taking in server telemetry data and visualizing it. Most of this processing is done inside of the blueprints found in the Blueprints folder. The Unreal visualizer will look in a folder designated as:

C:/Everglades/game_telemetry

for telemetry data. The current iteration of the Project Everglades server will generate telemetry folders complete with a map JSON, which the visualizer will load in and render. Currently, this will only work for maps designated as “3D”, as the functionality for rendering 2D maps was uprooted in favor of this. The previous iteration had to use hard-coded node placements for them to get 2D maps to work, thus there are separate implementations for 2D and 3D maps.

Once there is at least a telemetry folder which contains a 3D map inside of the directory specified above, the Unreal visualizer can be run – either in the editor or outside – and the desired telemetry folder can be chosen. To run the game, follow these steps:

1. Run the game either in-editor or from the build.
2. Click “Telemetry”
3. Click the desired telemetry folder. If none appear, then the valid directory specified above is empty.
4. Once clicked, the map will generate the map specified in the telemetry folder and simulate the game.

Camera movement is achieved by using the WASD keys, with camera rotation affected by holding right-clicking and dragging.

Improved Performance and UI

Besides implementing the 3D maps generated by this iteration of the project, performance improvements were implemented. Before, the Unreal editor often ran at 20 frames per second, even on high-end systems specially-built for gaming. There were many times where the game would refuse to load, even inside of the editor.

The main performance improvements make more use of the GPU while also improving the general efficiency of various operations. For instance, map nodes were indexed inside of an array, with map nodes needing to be searched through many times every turn. Naturally, this is a CPU-intensive process which puts more stress on the CPU as more map nodes are added. In lieu of the O(n) array search, a hash map was implemented which stores all nodes in the map by their ID, creating an O(1) search method. Additionally, unnecessary tick functions which were being called each frame were removed, further freeing up CPU processing for more important processes. This led to a marked improvement in framerate, with most games on high-end systems running at 60 frames per second.

The final feature added were group icons which were attached to each group instance. They read the player ID and group ID associated with that group and display that information in an easy-to-read way. Player 0's groups are shown in blue while Player 1's groups are shown in red, with the universal group ID being displayed on the icon. When the camera gets close to the icon, it will disappear, and reappear once the camera gets sufficiently far away again. This makes it much easier to tell where any group is on the gameboard at any given time.

Since this was not one of our main requirements, this was the most that was accomplished with the visualizer this semester. If we had more time, we would have liked to further iterate on this portion of the project by polishing it up more. In terms of straggling features or general polish, here is what we hoped we could have done if we had the time:

- Re-implement 2D maps.
- Display combat, unit death, and the disbanding of groups.
- Allow users to specify their own directory for telemetry folders.
- Fix control issues. When the space bar or shift key is hit, the game freezes. The likely reason for this is these control pausing the game tick and iterating it one turn, though it is unreliable and glitchy.

Build, Prototype, Test, and Evaluation Plan

The testing structure is an important part of software development. When developing new features, you have to account for not only for issues within the scope of the feature, but also any issues or bugs that may occur when integrating it into the main project. This process can be separated into three separate testing phases: unit, integration, and system testing. The use of GitHub's branching structure allows us to easily implement these testing phases.

Unit Testing

Unit testing is the process of testing a feature independently away from any dependencies it has. This phase ensures that the feature will work as intended and that the logic is foolproof.

Integration Testing

This process is usually done after unit testing has been performed to make sure there are no overlapping issues between modules. This includes running two different modules and checking to make sure they no issues occurring.

System Testing

System testing involved testing the entire system to assure that the integrated modules are working cohesively with each other. This process happens right before delivering a release version of your product.

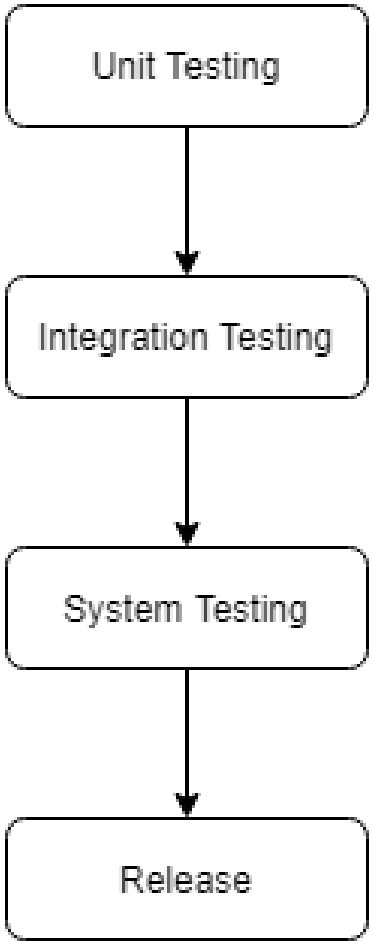


Figure 98:Testing Strategy

GitHub Development

We used GitHub's branch structure to independently develop our features, and to smoothly integrate into our project. Our branching structure consists of a master branch, development branch, and supplemental feature branches.

Supplemental feature branches are the main development branches. This is where the bulk of the work is done by our team members. For example, we have a branch for team configurations and a branch for the 3-D gameboard. This independent structure allows for thorough unit testing.

The development branch is where the integration testing and system testing happens. Once a feature branch has passed all its unit tests, it is then merged into this branch for further testing. This step assures that the integration process does not caused any problems, and if so, fixes are applied directly to this branch. System tests are also applied to this branch to make sure the project as a whole is running properly.

The master branch consists of working release versions that include new features and have passed the system testing phase.

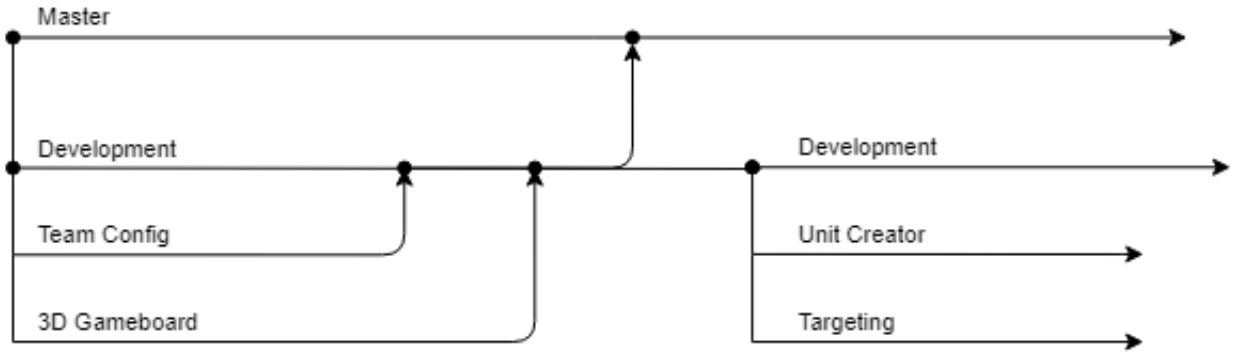


Figure 99: GitHub Development Flow

Unit Tests

Team Loadout

- JSON formats correctly on saving from UI.
- Loadout system correctly saves and loads the loadout corresponding to the player.

3D Gameboard

- Generate complete maps (no empty layers).
- Adjust weights.
- Format of json is correct.
- Connection nodes are accurately defined.

Improved Targeting Behavior

- Unique targeting systems are applied to drones.
- Drones prioritize targets based on their targeting system.
- Drone damage is built properly.
- Drone damage is applied to proper targets.
- *nulled_ids* array is updated correctly with killed units.
- Telemetry data is generated properly with new targeting system.

Improved Playability (Unit Creator)

- Units attributes initialized correctly.
- Units functions in game.

Unit Integration

Team Loadout

- JSON loads in correct format for conversion to players.
- Loadout is playable and does not trigger any default checks.

3D Gameboard

- Integrate wind stochasticity with the newly generated 3D gameboard.
- Assure that drones are able to play on the 3D map.

Improved Targeting Behavior

- Integrating drones with sensors and evaluate if drones limit their targeting options to only those they see in their sensors.
- Utilizing a custom targeting format, evaluate if the targeting behaviors change and operate based off of the custom input file.

Budget and Financing

Overview

The development of our project is contingent on two technologies: The Python server and the Unreal Engine rendering. Therefore, when finding the cost of our project, we have to be mindful of the possible costs incurred by the development environments of both halves of the project, the hosting of the Python server, and the version control of the repository. Thankfully, even at the most expensive estimate, our project will still cost less than $50. Although Lockheed Martin has agreed to cover all costs as they are incurred, we still attempted to minimize them as much as possible.

Python Server Costs

The development environment used by our contributors will have no impact on the performance of the final project and is purely a means by which the members will view, edit, and append Python code. For these reasons, we have decided to stick entirely to free and familiar development environments that will not incur costs to our project as a result. Some examples of Python IDEs used for this project are Atom and Eclipse via PyDev.

Unreal Engine Costs

As of 2015, the Unreal Engine End User License Agreement for Creators was updated to have no licensing fee in addition to its existing policy of charging no royalties for internal or free projects. This project is not intended to be sold and is only for internal use at Lockheed Martin and for free distribution as a learning tool, so our usage of Unreal Engine falls into this category. As a result, continuing development in Unreal Engine 4 also comes at no expense to us.

There are various server-hosting services that offer free use to students that we can take advantage of to host version control for Unreal. Typically, game engines such as Unreal and Unity utilize Perforce for version control, which requires AWs. We are opting to use a student’s version of AWS Lambda via an AWS Educate account, which provides 1 million free requests per month. In the very unlikely scenario that we would use up all 1 million free requests in a month overflow, the creation of the AWS Educate account will secure us up to $100 of credit that can go to covering those costs.

Version Control Costs

As of April 2020, GitHub has added an option to allow for team repository management for free with unlimited collaborators. Provided we use GitHub only for the version control of the Python server and we do not push the repository size beyond the GitHub Teams Free size limit of 500 MB (of which we are starting at 10.7 MB), the use of GitHub will be completely free.

Unfortunately, we will not be using one GitHub repository for version control of all parts of our project. The total file size for our Unreal Engine assets at the beginning of our project already exceeds 40 GB. For this reason, we will be using Perforce for Unreal version control, which is most typically used for version control in game development engines.

Perforce requires an AWS EC2 instance to be set up, which with a student account can accommodate 1 million free requests, and with the amount of alterations we have made to the Unreal portion of Project Everglades, such a request should not be met.

Summary of Costs

Python and Unreal are free and thus have no development costs. Our students' AWS Educate account will likely incur no costs by using AWS Lambda to host our server but may cost some amount in proportion to how much we exceed the free amount of requests. Using GitHub with Git LFS to host our Unreal assets will cost us $5 monthly for $25 unless we exceed 50 GB of data, in which case from the month we do onward the monthly fee becomes double at $10.

Gantt Chart

The following are the two Gantt Charts used for Senior Design I and Senior Design II respectively. The information contained within is reflected by the Milestones section after this.

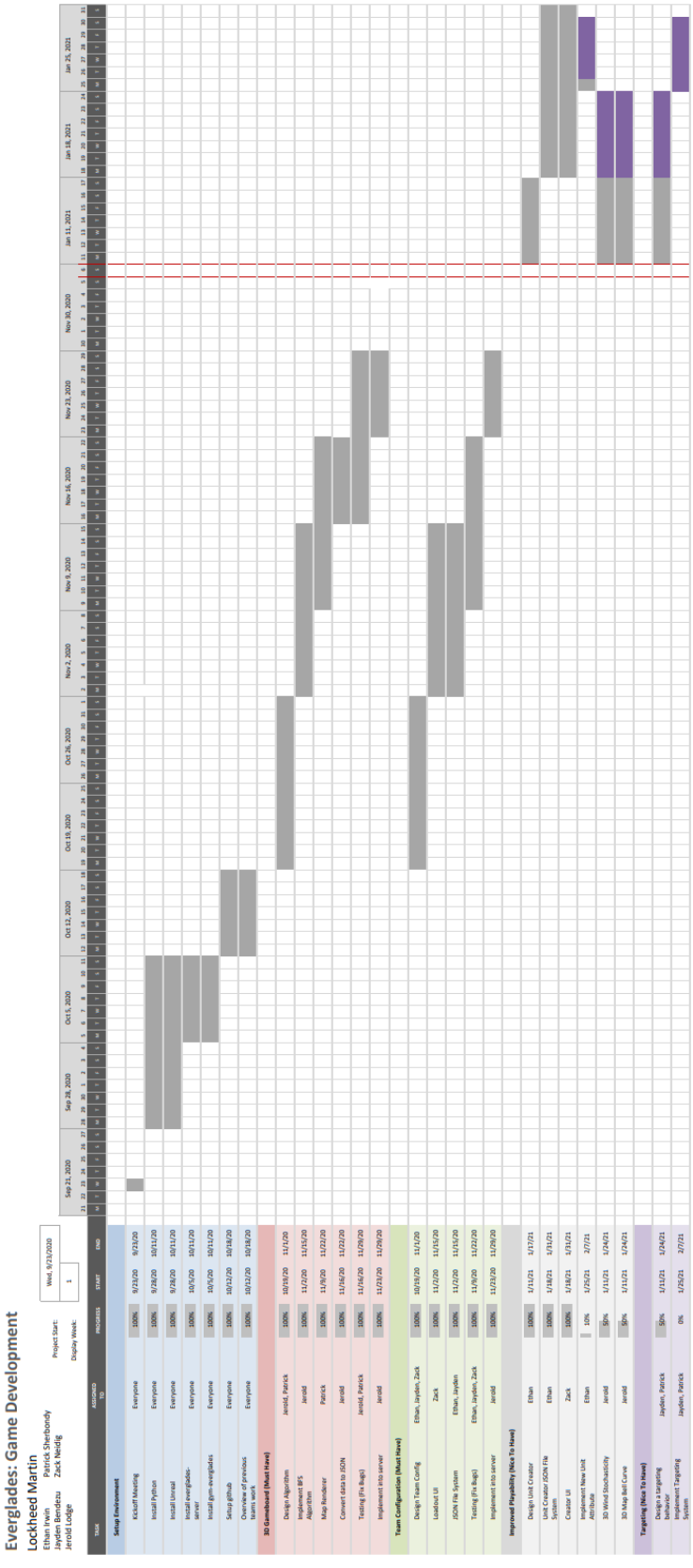


Figure 100: Gantt Chart

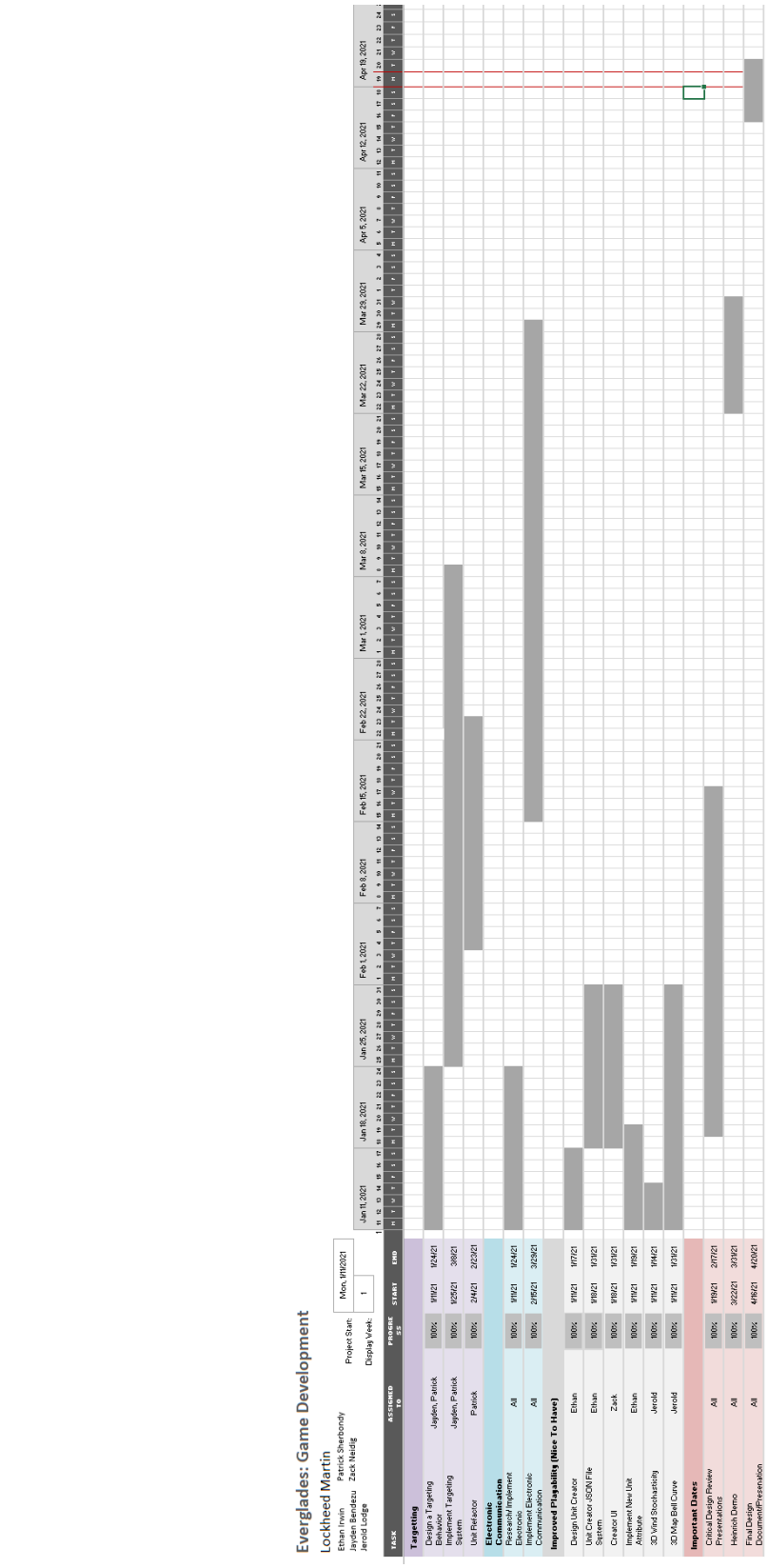


Figure 101: Gantt Chart for Senior Design II

Project Milestones

Senior Design I Milestones

Our primary goal with the first semester was to get fully acquainted with the existing codebase so that we could implement new changes as fast as our knowledge allowed. This is a large project with many moving parts, so failing to understand how each part interacts with one another would make it difficult to accomplish much.

Once all of the research into the project itself was finished, actual development could take place. The research and understanding of existing implementations took a while, as there was a lot to cover and fully understand. At this stage, we were ablet o get started on the unit creator, 3D gameboard generation, and some basic increase playability improvements in the form of crude UI mockups.

The 3D gameboard generation took a surprisingly short amount of time, as the implementation of the new system piggy-backed off of the previous team's existing solution. It was a simple matter of adding a third dimension to their gameboard generation algorithm. Although, a proper third-party renderer needed to be created to properly diagnose and repair issues with the generation algorithms. Additionally, the pre-existing wind stochasticity had to be refitted to work in 3D space, which required an extra dimension of complex mathematical calculations.

By far the requirement that took the longest to implement was the unit loadout requirement. This was because it directly required the modification of existing systems that had been hard coded in the project for some time now, so a lot of understanding and manipulation had to take place to fulfill the requirement.

As stated previously, some work was started with improved playability by creating custom UI screens to better interface with the program. All of the program had to be run and controlled by dragging and dropping files into proper directories manually. Thus, the main purpose of the desktop UI from the outset was to allow programmers and laymen alike to run the game using a simple interface.

With many requirements being accomplished in a timely fashion, it was possible to get started on extra requirements far earlier than once expected. While it was initially relegated to Senior Design II, the quick progress of this semester allowed us to get started on the improved targeting behavior requirement while also perfecting the basic requirements we had implemented.

Senior Design I Milestones

No.	Task	Planned	Completed
1	Team Formation	September 16, 2020	September 16, 2020
2	Project Kickoff	September 23, 2020	September 23, 2020
3	TA Meeting	September 25, 2020	September 25, 2020
4	Sponsor Meeting	September 28, 2020	September 28, 2020
5	Setup Design Document (15 Pages)	September 30, 2020	September 30, 2020
6	Get Familiar with the Previous Codebase and Document	October 2, 2020	October 2, 2020
7	Complete Setup/Implement Loadout UI, Reimplement Stochastic Wind	October 9, 2020	November 25, 2020
8	Meeting with Prof. Heinrich, Research 3D Gameboard and Drone Loadout Customization	October 12, 2020	October 12, 2020
9	Finish up research of Gameboard and Loadouts	October 19, 2020	October 19, 2020
10	Begin / finish designing implementation of Gameboard and Loadouts	October 26, 2020	November 25, 2020
11	Meeting With TA, begin implementation of Gameboard and Loadouts	November 2, 2020	November 2, 2020
12	Finish implementations of Gameboard and Loadouts	November 23, 2020	November 6, 2020
13	Improved Targeting Requirement Research, Continue Iterating UI design	November 30, 2020	November 30, 2020

Senior Design II Milestones

We were in a fortunate position at the start of the second semester, as we had finished most of our basic requirements in the previous one. This allowed us to not only get started on some of the more extreme requirements, but also focus our attention on these requirements throughout the entirety of the second semester.

As we began delving into the systems that would directly interact with the improved targeting requirement, we realized that the systems set in place by the previous team were too convoluted and poorly documented to allow for the new requirement. We petitioned Lockheed Martin to allow us to refactor the code, which was approved and started within the second month of the semester.

As the refactor took place, work continued on the main desktop UI, the secondary loadout UIs, and the backend of the unit creator. As this ran smoothly, we were able to find some spare time to begin working on the Unreal portion of Project Everglades on our own volition, despite it not being asked of us. This consisted of refactoring most of the Unreal blueprints to accept the new telemetry data, while also fixes to improve the overall performance.

With the middle of the semester coming around, we were able to finish up the unit class refactor in its entirety and perform regression testing to ensure the outputs generated by the refactor matched that of the previous iteration. With the approval of our sponsors, the refactor was complete and thus a full green light on starting the implementation of the targeting requirement.

Towards the end of the semester, we were able to sit down with our sponsors and do a mini showcase of our work at that stage, which was met with a very positive reception.

With the end of the semester coming around, we could focus solely on debugging and testing the code we had, while merging it together and preparing the software worth showing off during the final presentation.

Senior Design II Milestones

No.	Task	Planned	Completed
1	Research Drone Targeting and Electronic Communications	January 11, 2021	January 11, 2021
2	Continue work on Unit Creator, Continue iterating on UI, Continue drone targeting	January 18, 2021	January 18, 2021
3	Begin implementing targeting improvements, submit proposal for unit class refactor, start prep for CDR	February 1, 2021	February 1, 2021
4	Continue preparing for CDR, continue work on targeting systems, continue iterating UI, start unit class refactor	February 8, 2021	February 8, 2021
5	Present CDR, continue working on targeting, continue unit class refactor, work on electronic communications	February 15, 2021	February 15, 2021
6	Continue working on targeting, continue unit class refactor, work on electronic communications, continue iterating UI, continue unit creator, run regression testing on refactor	February 22, 2021	February 22, 2021
7	Continue working on targeting, continue unit class refactor, work on electronic communications, continue iterating UI, continue unit creator, refactor Unreal	March 1, 2021	March 1, 2021
8	Sponsor progress meeting, continue iterating UI, continue targeting work, add icons to Unreal	March 15, 2021	March 15, 2021
9	Begin testing and debugging, run targeting statistics	March 29, 2021	March 29, 2021
10	Continue testing and debugging, add small, viable features, integrate unit attribute code in necessary areas, merge all branches	April 1, 2021	April 1, 2021
11	Final Project Presentation	April 20, 2021	April 20, 2021

Project Summary and Conclusion

Project Summary

Project Everglades is a Windows PC application created by Lockheed Martin with the primary goal of simulating drone-based warfare in a video game setting. Its overall purpose is as a training tool for other universities as well as train fresh employees at Lockheed Martin to better understand artificial intelligence systems and machine learning, and how they work.

The goal of this turn-based strategy game is simple: capture as many nodes as possible before the max turn limit is reached, or completely defeat all of the enemy's units before the turn limit is reached. Players use robotic flying drones equipped with guns to capture and defend nodes on the gameboard. The amount of these drones that a player starts with cannot be increased throughout the game, as there is no way to produce new drones. Human players can play against an AI player or simulate a battle between two AI players.

Unlike most conventional video game, Project Everglades has multiple distinct components to it: an AI agent, a Python server, and the Unreal Engine. AI player actions are generated in the AI agent and passed to the python server, which takes those actions and performs various calculations before outputting telemetry data. This telemetry data is passed to the Unreal Engine turn by turn, where it is processed graphically.

This project is not on its first iteration, as there were at least two previous iterations – from Senior Design or otherwise - that comprised the life of Project Everglades. The previous Senior Design group's efforts were primarily focused on generating random 2D gameboards, adding stochastic wind, and creating electronic vision for the drone units.

A separate group from the Florida Interactive Entertainment Academy worked on creating a facet of the program that visually displayed telemetry data from the server in the Unreal Engine. Their iteration did not play off the previously mentioned group's progress, however, with their project being limited to a pre-determined gameboard. Most of their work was centered around creating the graphical assets necessary to display the telemetry data.

The focus of this iteration of the project was on expanding the gameboard in three dimensions, improving the customization and creation of drone loadouts, improving drone targeting, and devising a system for realistic drone communication.

Below is a comprehensive list of all of these goals and requirements:

- Necessary changes:
    - Generating a 3D Gameboard
        - Implementing completely new code to generate the 3D gameboard

        - Revising or replacing all the previously implemented solutions that considered strictly 2D gameboards and cannot support 3D
    - Drone Squad Customization
        - Un-hardcoding squad structure
        - Code new logic for squads of more than one type of drone
        - Create interfaces to customize squad structure
- Nice-to-have changes:
    - Increased Playability
        - Create various user interfaces to allow programmers and laymen alike to interface with the program for testing or other purposes.
        - Allow for the creation of custom groups, units, and attributes through this UI.
    - Drone Targeting Improvements
        - Create a variety of targeting behaviors that allow drones to prioritize targets autonomously with differing behaviors and priorities
        - Refactor unit classes and server files to allow targeting to be implemented.
- Home Run Goals:
    - Custom Drone Creator
        - Allows for the AIs (or player) to create new drone types that are not included with the game, including a point system to prevent either from making balance-breaking powerful drones
        - Create functionality for the AI to access the unit creator and make decisions
        - Create UI elements to allow players to access the unit creator
    - Electronic Communications
        - Implement electronic communications that affect the speed of enemy units.

For Future Groups

This section will discuss various foundations and loose ends that can be taken up by future groups that receive our iteration of the project.

Observation Space Refactor

The current observation space for unit types is currently setup as an integer value depicting the composition of units in each squad. For example, a striker will be defined as an integer value of 1 and a tank will be defined as an integer value of 2. Say a squad contains all strikers then the integer representation of the unit composition would be “1”. And if the squad containers strikers and tanks then the integer representation would be “12” or “21”. This is problematic for a couple of reason. First is that once there are for than 10 units it would be impossible to decode the integer representation accurately since some units will have a defined value with 2 digits. Second is that the AI may interpret some squad compositions as “greater” than other since they are integers. The observation space below is something we designed to combat these issues by changing the squad composition from an integer representation to a Boolean array where each index is a certain unit.

```
Observation_space = Dict({
   'board_state' = Box(),
   'player_state' = Dict({
      'groups' = Box(),
      'unit_types' = MultiBinary()
   }),
   'sensor_state' = Box()
})
```

Figure 102: Proposed observation space refactor

New Unit Attributes

With the unit creator comes the ability to add in attributes that apply various bonuses which can effect strategy. With each new attribute, the total amount of possible combinations to create new units increases. The addition of new attributes by future groups would be a desirable way of both adding to the game and tackling new avenues of other goals like improving targeting or electronic communications. New units relaying information with special attributes can fulfil electronic communication roles, and units that can change how targeting works could both be ways of expanding the game.

As stated in the chapter on new unit ideas for future groups, there are many unimplemented ideas that can be later added in for interesting effects. By leveraging this

system of attributes, it is hoped that there will be sufficient foundation for a large variety of future changes.

New Targeting Functions

The improved targeting requirement has been completed in whole, with four pre-made functions and a fifth function which allows for custom targeting functions to be made by the teams responsible for the AI portion of Project Everglades.

The current targeting functions that are implemented include one which selects targets at random, one which selects targets by lowest health first, one which selects targets by highest health first, and one which selects targets both by highest health and highest damage first. This is highly scalable and not limited to just these pre-made functions, which are by no means perfect. A project could be undertaken which devises new targeting functions that prove strategically useful in various ways.

Additionally, the custom targeting function exhibits only a small degree of input validation, which is meant to prevent the AI teams from cheating. This can be extended to handle a variety of cases to positively ensure that the AI teams cannot find creative ways to cheat the game.

Extend Electronic Communications and Warfare

The development undertaken on electronic warfare this semester was purely foundational, due to issues with getting a sufficient explanation of a way to model electronic communications within the Python server.

Currently, electronic warfare only results in the change of the speed of various units who are engaged in electronic warfare. The foundations are in place, however, for more to be done with this system, allowing a future team to start development on electronic warfare that realistically models the real-world.

Extend the Unreal Visualizer

The Unreal portion of the game existed before we began this iteration of the project, however it was only set up to support two-dimensional maps and had incredibly poor performance that dipped into 20 frames per second on the high-end desktops tested. On some systems, it outright would not run.

For our iteration, we fixed the performance issues, enabled 3D maps for the Unreal visualizer, and fully hooked it up with our telemetry data. Future groups may want to extend the functionality of the Unreal portion by re-implementing 2D maps, display more telemetry data graphically, un-hardcode the telemetry folder directory, fix various control issues, and overall extend the visualizer’s functionality with the python server.

References

[1] "1200px-Breadth-first-tree.svg" Wikimedia https://upload.wikimedia.org/wikipedia/commons/thumb/3/33/Breadth-first-tree.svg/1200px-Breadth-first-tree.svg.png

[2] "spherical_coordinates" Math Insight https://mathinsight.org/media/image/image/spherical_coordinates.png

[3] "Cb_splash" Codeblocks Wiki http://wiki.codeblocks.org/images/c/c7/Cb_splash.png

[4] "header-logo" wxWidgets https://wxwidgets.org/assets/img/header-logo.png

[5] "logo02" Ratfactor http://ratfactor.com/misc/mingw64/logo02.png

[6] "ATOYX Mini Drone for Kids and Beginners" Amazon.com [Online] Available at: https://www.amazon.com/ATOYX-Quadcopter-Helicopter-Batteries-Beginners/dp/B07GTCB7G7/ref=sr_1_2?dchild=1&keywords=drone&qid=1607205809&refinements=p_n_feature_seven_browse-bin%3A7029321011%2Cp_n_feature_three_browse-bin%3A15697663011&rnid=15697662011&s=toys-and-games&sr=1-2&th=1 on 2 December 2020

[7] "Freefly Alta X Drone with FPV System" adorama.com [Online] Available at: https://www.adorama.com/fr950100vu.html on 2 December 2020

[8] "Apex Legends Crypto guide [Season 7]: abilities, hitbox, tips and tricks" rockpapershotgun.com [Online] Available at: https://www.rockpapershotgun.com/2020/11/10/apex-legends-crypto-guide-abilities-hitbox-tips-tricks/ on 2 December 2020

[9] "Parallel Array" geeksforgeeks.org Available At: https://www.geeksforgeeks.org/parallel-array/ on 30 March 2021

[10] "What is Regression Testing? Definition, Test Cases (Example" guru99.org Available At: https://www.guru99.com/regression-testing.html on 30 March 2021