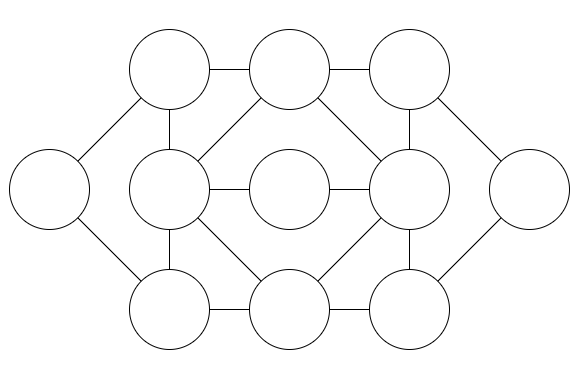
Everglades: Game Development for Reinforcement Learning

James Dwyer, David Gravett, and Michael Fielder

Department of Computer Science, University of Central Florida, Orlando, Florida, 32816-2450

***Abstract*** — **Project Everglades is a game developed by Lockheed Martin as a tool to help teach reinforcement learning for artificial intelligence. This game consists of a Python based game server and an Unreal Engine post processer to view the completed game. For our project, we were tasked with improving upon this game by adding more advanced features, such as a randomized game board and wind effects, as well as preparing the game for future improvements by characterizing the codebase and creating the base for new units.**

***Index Terms*** — **Computer science, object-oriented programming, machine learning, computer applications**

### I. Introduction

Project Everglades is a game developed by Lockheed Martin as a tool to help teach reinforcement learning for artificial intelligence. This game is played by two opposing AI Agents. Each agent has a starting base that they must defend while also attempting to capture the opponents base. The default map (Fig. 1) consists of a 3x3 layout of nodes between these two starting bases that the drones must travel between to reach to opposing base. Each of these inner nodes can also be captured which will award that player with points depending on which node was captured. Some of these inner nodes will have special bonuses for the player who controls the node. Some inner nodes will also be either a watchtower or a fortress. A watchtower node will allow the controlling player further vision across the map. A fortress node will give the controlling player’s defending drones a boost in their defense to help during combat. At the start of each game, both players will begin with 100 drones split into several different groups. The players can use these drone groups to traverse around the map to capture nodes and accumulate points. If drone groups from opposing players meet on the same node, they will battle for that node. There are 3 types of drone units these groups can contain: controller, striker, and tank. Each drone type has different properties allowing for complex strategy development. A striker has increased movement speed and damage but decreased armor, a controller has increased capture speed for nodes, and a tank has increased armor. The game consists of 150 turns where each player can move up to 7 drone groups per turn. There are 3 possible ways a player can win: the player captures the opposing player’s starting base, the player destroys all of the opposing player’s drones, or the player has more points than the opposing player at the end of the 150 turns.

This game consists of a Python based game server and an Unreal Engine post processor to view the completed game. Unlike other popular video games, this game executes completely before displaying any of the visuals to view the game in action. The game logic is completely processed by the Python game server, which outputs telemetry data to be used by the Unreal Engine game client to visualize the match. This telemetry data includes information such as the drone unit group compositions, drone group movements, and the player’s point totals after each turn.

Fig. 1. Default Map Layout. The leftmost and rightmost nodes are each player’s starting base.

For our project, we were tasked with improving upon this game by adding more advanced features, such as a randomized game board and stochastically seeded wind effects, as well as preparing the game for future improvements by characterizing the codebase and creating the base for new units. The three tasks that were required by our sponsor are the characterized codebase, randomized game board, and stochastically seeded wind effects. The other requested tasks were not required but were desired to be included. These other tasks included the new drone unit type and an improved drone unit death animation. In addition to these desired features, we all worked on some additional parts for this project. This included reworking the Unreal game client to function with the Python game server, fixing some bugs in the game server and the game client, and adding small quality of life features to the game client.

### II. CHARACTERIZING THE CODEBASE

Characterizing the codebase of the previous iteration of Everglades: Game Development helped us understand how the existing classes and procedures affect the overall flow of the game. Understanding the current structure of the game was critical to effectively integrate our improvements and additions. Providing documentation also allows future developers to better understand the design and purpose of the existing code.

The Everglades codebase consists of two distinct sections: the Python server and the Unreal client. For both the server and client, the codebase was manually researched. The server required walking through the code and making use of *Pyreverse* to confirm variables and data types. Tracing blueprints in Unreal was necessary to unravel its codebase. The layout provided an easier way to visualize the code most of the time but could prove confusing if the blueprint was large with many connections.

Markdown files were used to present this characterization. The raw text format was easy to create and edit while the HTML output looked both clean and professional. During development, the Visual Studio extension *Markdown Editor* provided a real-time display of the output as content was added to the files. After development, the files were pushed to our project’s GitHub repository, where they could easily be viewed as needed. A partial Markdown file can be seen in Fig 2.

A screenshot of a cell phone

Description automatically generated

Fig. 2. An example Markdown file showing a class and associated variables from the Unreal client codebase.

A single Markdown file serves as the README for the repository’s *Codebase* folder and has links to the server and client sections, each of which has their own respective file containing alphabetized links to classes, variables, and functions. The Unreal client file also contains links to events, event dispatchers, and macros. All of these have their own respective Markdown files except for variables, which are listed in a table in the relevant class’ file.

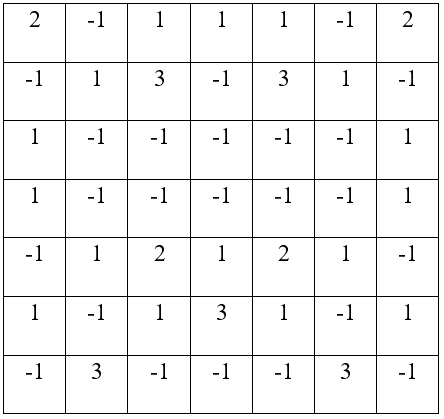
The Markdown files for the server’s methods differ slightly from the client’s method, event, event dispatcher, and macro files. Both provide descriptions, but the server’s file presents the syntax for calling the method as well as a table describing the parameters as seen. The client files present two tables of inputs and outputs, mimicking the structure of blueprint nodes in Unreal.

### III. Randomized game board

Previously, Project Everglades was only able to simulate matches on the default 3x3 game board (Fig. 1). For our project, we were tasked with adding a feature to generate random game boards to be used for each simulation of the game. The implementation of a random game board will add more complexity for the AI agents to overcome by now needing to adapt for any possible game board layout.

The procedurally generated game board is restricted to under 50 total nodes as any bigger would be too large for the 150 turns of the game. This allows a square board with dimensions up to 7x7 or a maximum of 49 nodes. Using Breadth First Search, a randomized layout of nodes constrained in a 7x7 area can be generated to make up the game board. To ensure the board is fair for both players, only half the board will be generated and then will be mirrored to the opposite side. This will give both players an equal number of nodes that can be reached in the 150 turns.

The game also contains territory bonuses for certain nodes. Some nodes are also Fortresses, which grant an additional defense bonus for any allied units in the area during combat, and some are Watchtowers, which extend a player’s scouting range to adjacent territories. During the generation of the board there will be an additional check to see whether the generated node contains a Watchtower, Fortress, or neither. These territory bonuses are included in the mirroring of the game board to ensure a fair field for both players. The frequency of these structures can be changed by altering the weight that is used to check if the node will contain a territory bonus.

The method chosen for our implementation is a weighted version of Breadth First Search. This process begins by first creating a 2d array of the desired map size, 7x7 for example, and initializing all the values to 0. In this array, a 0 marks a node location that has not been tested yet, a 1 marks a successful node, and a -1 marks an unsuccessful node. The generation starts by adding the initial node to a queue and marking that location with a 1 in the 2d array. This first node is pulled out of the queue to test for possible connections. A node has a maximum possible of 8 connections 1 unit apart by connecting horizontally, vertically, or diagonally. To randomize these connections, an initial weight is assigned to the pulled node equal to 1 divided by the total possible connections for that node. For example, if a node has 5 possible connections, the initial weight will be set to 1/5 or 0.2. To test if a possible connection will become a node, a random value between 0 and 1 will be generated. If this value is less than the current node weight, a new node will be created and added to the queue. An additional test will then be done to determine if this new node will be a fortress, watchtower, or neither. After this secondary test, the new node’s location will be marked in the 2d array with a 2 if it is a fortress, a 3 if it is a watchtower, or a 1 if it is neither. If the node creation test value is greater than the current node weight, the node weight will increase by the same amount as the initial weight. For example, if the first test fails, the node weight will increase to 2/5 or 0.4. If all tests fail up to the last test, the weight will have increased by 4/5 or 0.8, bringing the node weight up to a value of 1. This final test will now have a 100% chance of creating a new node. This is used to ensure that all nodes have at least 1 new connection to guarantee the creation of a valid game board.

Once the queue has emptied, half the game board will have been generated. Since the players earn points based on how many nodes they have captured, the board is mirrored to ensure a fair match between the two players. If the board size is an odd number, the center line will still need to be generated. This center line is generated using the same weighted system but instead of using the weight to ensure each node has at least 1 connection, it is used to ensure a certain amount of center connecting nodes. A valid game board will be determined by how many center connecting nodes were generated. If the minimum required is set to 2 nodes, only game boards with at least 2 center nodes will be considered valid. If the game board is invalid, the generation process will execute again to generate a new valid game board. This process will always result in a valid game board, an example of which can be seen in Fig. 2.

Fig. 2. Randomly generated game board example. A 1 marks a regular node, a 2 marks a fortress, a 3 marks a watchtower, and a -1 marks an unsuccessful node creation.

Matches using randomized game boards can also be viewed in the Unreal game post processing. The nodes used in the random board for that match will be stored in the telemetry output data for the match. When the Unreal game client loads in the telemetry data for the desired match, the nodes included in that match will then be rendered to match the layout of the random game board.

### IV. Stochasically seeded wind effects

### V. Improved drone death animation

### VI. New drone unit

### vII. Additional work

### ViII. conclusion