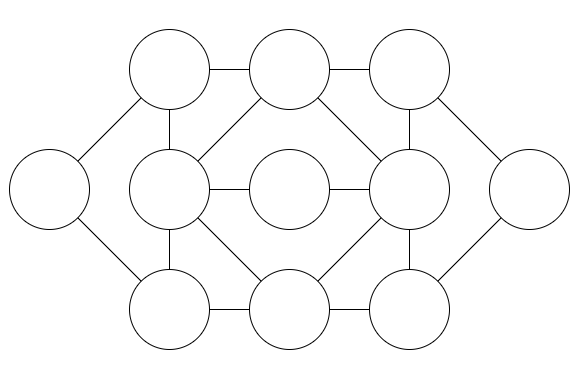
Everglades: Game Development for Reinforcement Learning

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 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 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 as seen in Fig. 2.

A screenshot of a cell phone

Description automatically generated

Fig. 2. An example of a class and associated variables from the Unreal client codebase.

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

### IV. Stochasically seeded wind effects

### V. Improved drone death animation

### VI. New drone unit

### vII. Additional work

### ViII. conclusion