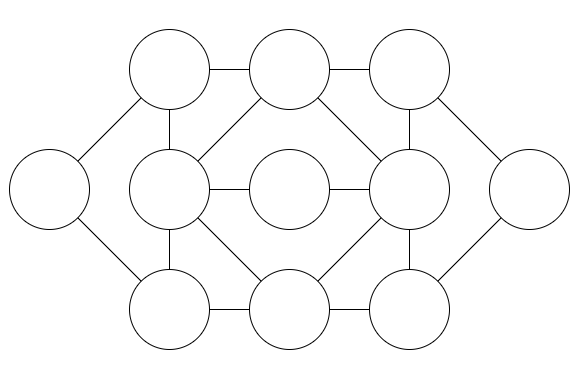
Everglades: Game Development for Reinforcement Learning

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.

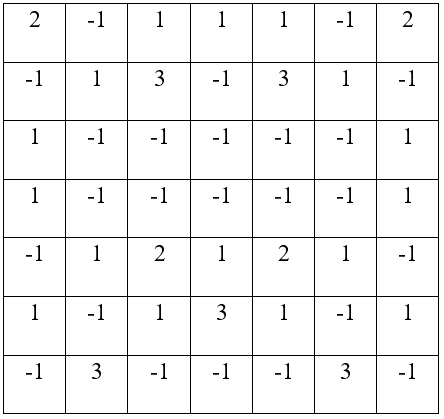
The Markdown files for the server’s methods differ slightly from the client’s method, event, event dispatcher, and macro files. Both provide descriptions, but the server’s file presents the syntax for calling the method as well as a table describing the parameters. The client files present two tables 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. 3.

Fig. 3. 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

One of the main objectives for improving the Everglades was to implement a stochastically seeded wind effect. Randomly generated game features are particularly important for training AI agents, as it provides them with a variety of different scenarios to learn from. The wind is represented as a vector field where each generated vector is the force that the wind produces. The vector field is created using Perlin noise to generate the angle of each vector. A 2D vector field was created because this application of wind can easily be scaled to an application that uses 2D movement, if future iterations decide to make that change.

Perlin noise was chosen for generating the field, because it creates a sequence of random numbers that is more natural and holistic. This is extremely important for something like wind generation, as you want it to be represented in a natural manner. Each vector corresponds to a generated node in the same position, which means the vector field is a parallel array to the generated game board. The stochastic aspect of the vector field is a seed which determines the offset of the generated Perlin noise values. This provides a large variety of possible wind values. The offset is a any number between 1 and 10000. The seed value is passed through the GameSetup.json file under the “Stochasticity” field and is read in the server.py file. The wind can also be mirrored as shown in Figure 4.  This ensures that there is an even playing field for each player in the game.

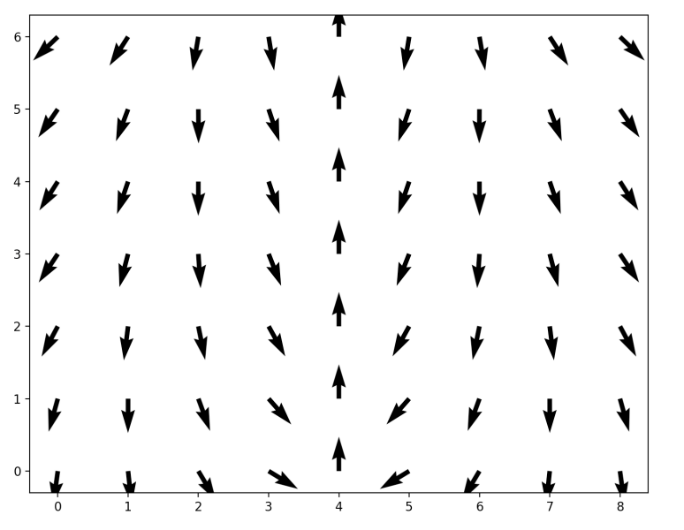


Fig. 4. Mirrored vector field that uses Perlin Noise

Movement in the Everglades game is one-dimensional where units can only move from one node to another through the connection between them. In order to accommodate this kind of movement, the wind affects a traveling unit through a scalar that is applied to it depending on which way the wind is facing. If the wind is blowing against the unit, it will slow down and vice versa if the wind is blowing with it. Each of these magnitudes is calculated by determining the magnitude of the projection of the wind vector onto the direction of the connection. A visual representation of this projection is shown in Figure 5. In the figure, F represents the force of the wind, U represents the direction of the connection, and A represents the result of the projection. Using the projection to determine the scalar value ensures that the value can be found no matter the direction of the connection.

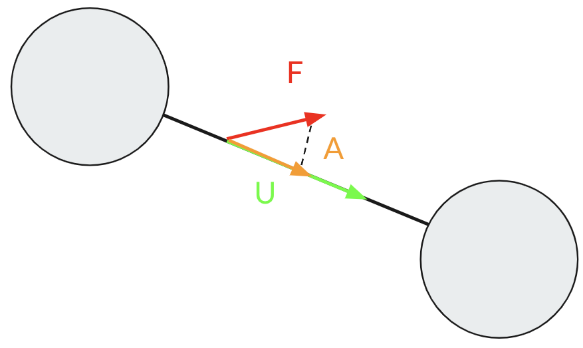


Fig. 5. Representation of wind vector projection on a connection

Each magnitude value is stored in an a hash map where the keys are tuples of the to and from node numbers and the value for a key is the scalar value. This hash map is then used in the movement function to affect the speed of a unit. The scalar values can be any number between -.2 and .2 meaning that the modifier is at most a 20% benefit or hinderance. This results in unit that can move at any value between 80% to 120% of its original speed.

### VI. New drone unit

The “homerun” requirement from Lockheed Martin was to develop a new unit that utilized some type of optical sensor. The team decided to simulate an infrared sensor for the project.

The purpose of the sensor was twofold. It needed to function on the Python server to participate in the current Everglades game. Additionally, it should work effectively in the Unreal client just as the existing units. However, additional sensor functionality should be present in Unreal, making use of visual data. Since the client is currently just a playback of the completed match, this won’t affect games but could be used during future iterations of the Everglades project.

The recon unit has the properties of mode, range, and wavelength which can be configured by the AI agent or use a default if none were found. Each group can have a unique sensor configuration, but all recon units within the group share that configuration. The properties have different uses depending on which part of the Everglades game the unit is operating.

*A. Python Server*

The new unit, called recon, contains similar statistics to the other units that are in the game, such as health, damage, and speed. Their settings are the minimum except for speed, which is 3, making them the fastest unit type. They also needed a unique capability. Since there is no visual data on the server, recon units needed some way to “sense” other units. The game already contains a “watchtower” to sense enemy units at adjacent nodes, so it was decided the sensor should detect enemy swarms travelling between nodes.

The sensor data is reported back to the agent through their observations which required adjusting the observation space in *everglades\_env.py*. The sensor state is reported every turn for each player through the *sensor\_state()* function in *server.*py and is a *numpy* array. The first index is the turn number, second is the source node the enemy units are travelling from, third is their destination, fourth is the number of *controller* units sensed, fifth is the number of *striker* units sensed, sixth is the number of *tank* units sensed, and seventh is the number of *recon* units sensed. Indices two through seven are repeated for the number of groups initially possessed by the enemy player. A telemetry file is also generated to allow an easier visual inspection of the data, but this file is not utilized by any Everglades component. An example of this output can be seen in Fig. 4.

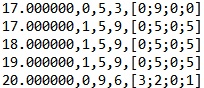


Fig. 5. Snippet of a SENSOR\_ServerData telemetry file. The elements are turn number, player number, enemy’s source node, enemy’s destination node, and the numbers of sensed units by type.

How detection is handled depends on the configuration specified to the unit. *Active* and *passive* are the two possible “mode” properties. *Passive*, which is the default mode, can only sense *controller* and *striker* units while *active* can sense all unit types. The consequence for using *active* mode is that the unit becomes visible to enemy recon units either travelling between the same nodes (no matter the distance) or at either end node.

The “range” property is an integer that specifies the distance that the recon unit’s sensor can reach, with a minimum and default of 1 and a maximum of 3. However, increasing the sensor range proportionally decreases the speed of the unit, with a speed of 1 when the range is maximized.

The “wavelength” property is a float that is not used on the server but is instead passed to the Unreal client.

*B. Unreal Client*

Since the client is the visual component to the Everglades game, the recon unit needed a visual representation that was unique. This was accomplished by exporting an existing asset from the game to Blender, an open-source 3D computer graphics program. There, the asset was edited, then imported back to Unreal as a new asset, and applied to the recon unit.

To simulate thermal data, several data tables were created. One table represents thermal data and contains rows for each type of object in the game world and a respective temperature value in Celsius. These values can be edited as desired. The remaining tables each represent infrared spectral reflectance data for each object in the game. The rows represent a specific electromagnetic wavelength and contain a value representing the reflectance of the material at that wavelength. This data was retrieved from the United States Geographic Survey Spectral Library 7. Note that some assumptions were made about the materials that make up these objects. Future developers can replace an existing table with one they imported themselves, making sure it has the same name.

To sense the properties of objects in the world, the recon units each make a call to Unreal’s *MultiSphereTraceByChannel* function. A flag is checked to make this call only once every turn; however, the check can be removed to make this call occur every tick. This function projects an invisible sphere outward from the front of the unit to the specified range and reports any objects with which it has collided. Then, thermal data for each object is retrieved from the data table if in “passive” mode, or reflectance data if in “active” mode. Reflectance data reported is within a range of the unit’s wavelength parameter. This is because there are 2,151 rows in the table so a range of data will be needed to be any kind of use. Recon units ignore units from their own team.

A telemetry file is generated to present the infrared data that was collected. This is currently only to demonstrate the functionality and has no effect on gameplay. Each line shows the current turn, the player number, the name of the recon unit that made a detection, a list of temperatures if in passive mode (empty otherwise), and a list of reflectance values if in active mode (empty otherwise). An example can be seen in Fig. 6.

A close up of a logo

Description automatically generated

Fig. 6. Snippet of an infrared data telemetry file. The elements are turn number, player number, sensing unit, sensed temperatures, and sensed reflectance.

An additional feature is the simulation of thermal vision by pressing the ‘2’ key. This makes different types of objects appear as different colors along a color gradient based on their temperature values.

This is achieved with a post-process volume that uses a post-process material instance. First, each type of object in the world is flagged for a custom depth pass and assigned a stencil number in the Unreal editor. Stencils allow objects with custom depth to be differentiated. Then, in the material blueprint, temperature values are converted to a range between 0 and 1 based on a temperature minimum and maximum. These converted values serve as a uv coordinate to choose a color from the color gradient. A second, cooler color is also chosen from the gradient by multiplying the uv by a float between 0 and 1. These color values are used along with the scene normals to create a Fresnel material. A panning noise texture completes the effect.

The material is unable to get data from the thermal table that was created, so the matching data must be manually entered for each object. Fortunately, the material instance applied to the post-process volume was created from this material. It allows the developer to easily change temperature values, the minimum and maximum temperature, the Fresnel exponent, and the color gradient used for the thermal image.

### vII. Additional work

Prior to this project, Everglades did not support mixed-unit or custom configurations, instead providing a single configuration for any and all players. The ability for an agent to specify their own unit configurations for their groups would allow much more customization and strategy. Agents can now specify the configuration of their groups by including a unit configuration dictionary. The key is the group ID number and the value is a list of tuples. Each tuple contains the name of the unit type in quotes followed by the number of those units. Asserts are used to verify the correctness of the format and to ensure the configuration conforms to the rules of the game. If the agent does not supply a configuration, the server will provide a default.

Mixing units caused issues in a few parts of the game that needed to be corrected. Most were remedied by adding loops to cycle through the different unit types for each group. Player state, combat, and movement were a little more involved.

Player state needed an integer representation of the units in the group. A single-digit integer represents each unit type (e.g. 0: controller, 1: striker). These digits are placed in decreasing order to preserve the 0 digit and used as a single integer. For example, a group consisting of the four types of units would be represented by the number, 3210.

Movement needed a creative solution. Since units must travel in a group, it was decided that the speed of the group must be the speed of the slowest unit in the group. A sorted list of unit speeds is used for each group, ensuring the slowest speed is at index 0.

Combat required the use of a parallel list to track the types of units targeted. This allowed the proper stats to be used during combat as well as correctly removing the correct unit should it be destroyed. Destroying all units of a certain type also remove that unit’s speed from the group’s speed list.

### ViII. conclusion