Bilkent University

Department of Computer Engineering

Senior Design Project

Project Analysis Report

Project Name: Espionage

Project Group Members:

Özgür Öney	21101821
Ahmet Safa Kayhan	21001532
Selçuk Gülcan	21101231
Ateş Balcı	21202771
Fırat Özbay	21201683

Supervisor: Can Alkan

Jury Members: Öznur Taştan, Uğur Güdükbay

Project Website: espionage-game.github.io

1. Introduction

Video games that get involved our lives in pre 60's, from these day to nowadays confront us as not only a basic show business element but also software that is used by people from every age and worth millions and billions worth market share. Such software that plough through software world with the help of devices called game consoles at the very beginning became very popular just after personal computers proliferation such that there is at least one PC in every single house.

Espionage, basically is an two dimensional (2D) stealth-arcade game which has character left into a map to go from one point to another with the help of supplied inventory and where users directs this character. Map, that player will follow to finish the particular level is not constant structure of course; it will be designed as layered and will be looking like a lateral section of a building. In addition to its lateral section structure, it will contain elements that have different characteristics; such as boxes will be set down to hide behind them, stairs will be there to elevate between floors/layers, vent holes to pass through enemies without getting attention etc. Needless to say that, number of elements will differ from level to level with the help of machine learning algorithm described below to change difficulty.

On player's way to the end, computer-controlled units that have own adaptive intelligence attempts to fail him whereas player attempts to complete the game by reacting with reflexive and strategic moves. As character is controlled by player, enemy units that aim to fail player will be controlled by artificial intelligence algorithm, which means that they will also behave in natural ways to give reaction to player's character. For example, whenever character is seen by an enemy unit, such enemy unit will alert all other enemy as well as change their constant formation (such as patrol route or condition of being armed etc.). In addition to that, a machine learning algorithm will detect the player's style by gathering data like time spent, number of tools used or number of enemies neutralized. Such processed data will be used to determine the level of difficulty of course, hereby play a critical role on playability and enjoyment.

Most importantly, a genetic algorithm will be implemented for the game. As the name suggest, a spectrum basic artificial intelligences for every enemy unit will be implemented first and game will be opened to many players to play. As time goes and players play the game to finish, just enemy units that have adequate intelligence will be survived, while others were simply eliminated. To do this, of course a score mechanism for enemy units to detect ones those are able to survive more. At last, a combination of surviving ones will be created to construct a sound basis artificial intelligence.

1.1 Object Design Trade-Offs

High quality graphics vs. Performance

Espionagé game aims to satisfy the potential customer-gamers during gameplay. To do such, graphical content -including character that player control, non-playable enemy units and even objects to interact- of the game should be drawn in high

resolution. To satisfy this requirement, a professional help from a graphic designer has taken. However, GPU of a single computer is directly responsible from drawing of elements to the screen and due to high load on drawing, performance problem can occur. Hereby, balance of such trade-off should be achieved through a wise policy to sacrifice neither performance nor quality.

Cost vs. Scalability

Espionage game keep the data of the users in its community such that it records every single player's level passing time, number of gadgets used and enemies eliminated to optimize the difficulty level of the game. As number of players and number of levels played increases, machine learning algorithm's approximation for difficulty approaches to optimal value. However, keeping track of such attributes simply requires a working database on the background. As pre-defined numbers increase, since expenses are directly proportional, they are also increases. To cope with such trade-off, optimal number of players should be determined.

Complexity vs. Playability/Usability

Espionage game includes much more traits compared to a basic adventure game: enemy dynamics that uniquely affects level difficulty, gadgets that changes playability of the game from head to toe as well as environmental factors that stands here to determine level structure. As combination of these values are getting complex, playability of the game is dramatically decreases for the player because of the fact that people tend to understand basic combinations much more than complex ones. Hereby, as a trade-off, elements defined above should be spread over the game so that they do not make game difficult to understand and easy-to-play.

1.3 Engineering Standards

UML: Unified Modeling Language is a standard that is intended to provide a way to visualize the design of a system.

OpenGL: Basically a standard maintained by the OpenGL Architecture Review Board (ARB) for rendering 2D and 3D vector graphics.

IEEE Xplore Standard: Naming conventions from web service to machine learning algorithm is covered by this standard.

1.4 Definitions, Acronyms and Abbreviations

UML: Unified Modeling Language.
SQL: Structured Query Language.
DB: Database
AI: Artificial Intelligence
GPU: Graphical Processing Unit, a part of computer which is responsible for rendering graphical objects.
Python: Object-oriented extensive programming language
ML: Machine Learning GUI: Graphical User Interface

2D: Two dimensional. Designed in a way such that its only possible to see width and length of a single object.

2.0 Packages

2.1 Artificial Intelligence (AI) package

Package that is responsible for every single action of non-controllable enemies. Explanation of classes in this package is given below.

Node: Grids defined in each frame of a level which is used in calculations of possible paths that non-playable enemy unit can follow.

Graph: Class that manages interactions contain nodes, such as choosing a route from current location to destination or selecting a place to hide.

ReactionAI: Class that keeps algorithms responsible for reactions of non-playable characters.

State: Non-playable characters have their unique states depending on the player's interaction with them, such as one enemy should be in alert state whenever our player-controlled character is seen by him. This class herein is responsible for hosting algorithms that determines states of non-playable characters.

ApproachAction: Class responsible for guardians'/enemies actions whenever they meet our player controlled character such as going towards source of a unidentified sound.

ChaseAction: Class responsible for guardians/enemies whenever they meet a player who is escaping from them.

FireAction: Class responsible for guardians/enemies whenever they are forced to use their weapons. In a scenario when player escapes from shoots, non-playable units will be managed in a way that they will shoot in the direction where player escapes to.

ActionManager: Class that contains a priority queue for the events so that event mechanism is arranged in a logical order. For example, whenever a series of events occur, their respective drawings and triggers should be materialized in a chronological order.

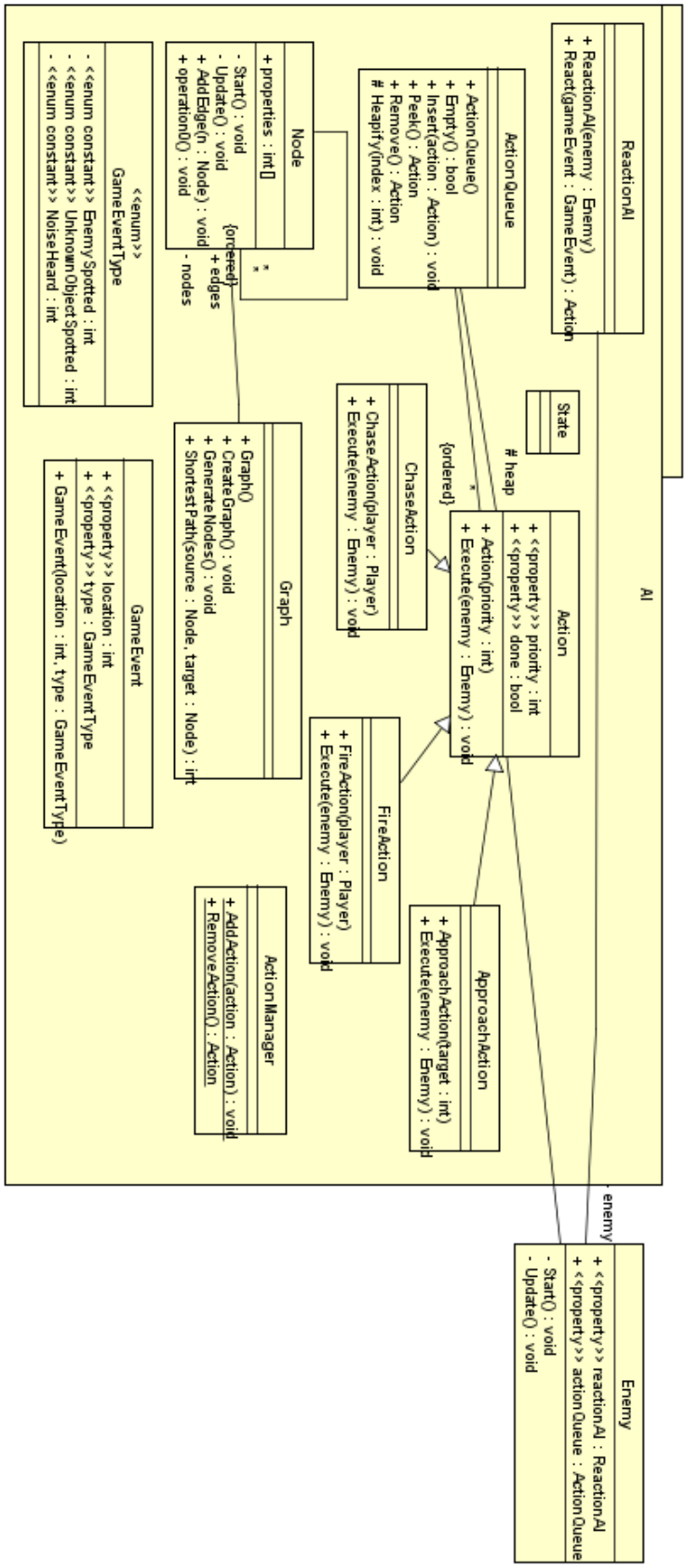


Figure 1: Detailed component diagram of AI package of the game.

2.2 Machine Learning (ML) package

This package contains classes that are responsible for every single Machine learning action. Explanation of classes in this package is given below
MLCommunicator: Class required to communicate with machine learning module.

MLConfig: Class that contains basic configuration information needed to communicate with ML module.

MLLevelStats: Class keeping data of level statistics like time spent, number of enemies eliminated and number of gadgets used during a single level.

MLLogger: Class that keeps the log information which will be later used to update MLLevelStats' information.

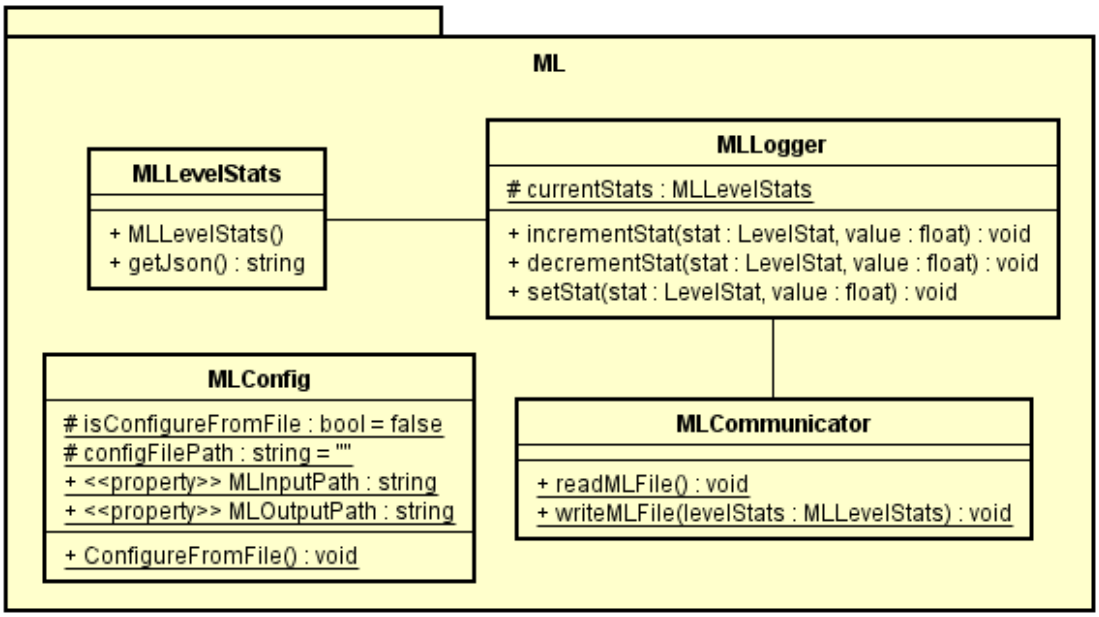


Figure 2: Detailed component diagram of ML package of the game.

2.3 Enemy package

Enemy package can be seen as a bridge between AI package and game itself. Decisions taken by AI package are implemented/drawn/applied through Enemy package. Therefore, respective classes to connect AI & game each other will be implemented for each AI package class.

IApproachable: Interface responsible for executing commands taken from PathingAI class defined above.

IFirable: Interface responsible for executing commands taken from ReactionAI class defined above.

ISpotable: Interface responsible for executing commands taken from ChaseAction & FireAction classes defined above.
Enemy: Class responsible from initializing & updating enemy attributes.
Turret: Subclass responsible for actions of turrets, which are unshifted enemies that attacks whenever it sees player controlled unit.
Guardian: Subclass responsible for actions of shifting enemies that not only attacks whenever they see player controlled unit but also chase him down and warn any other guardian.

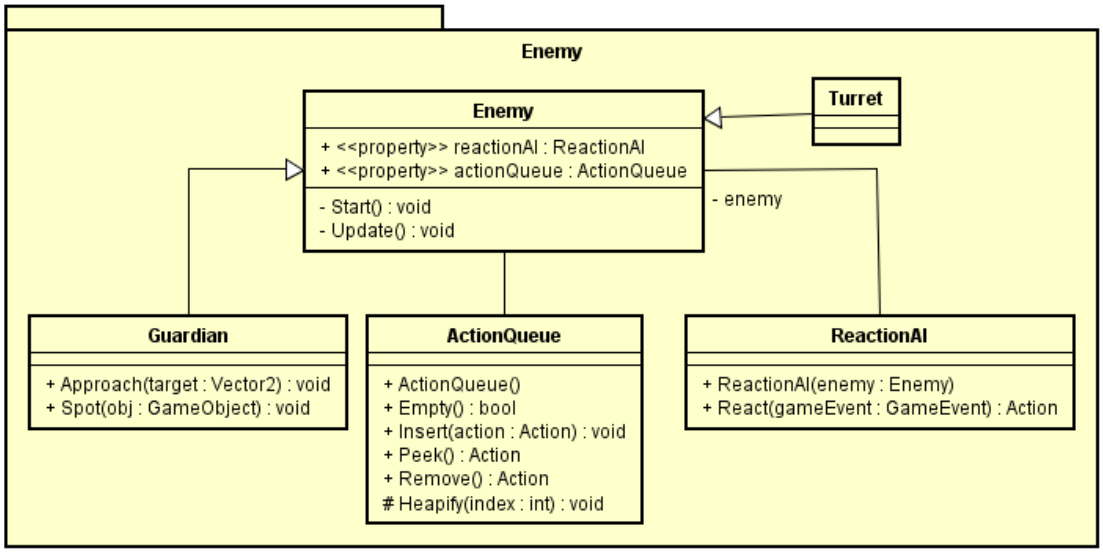
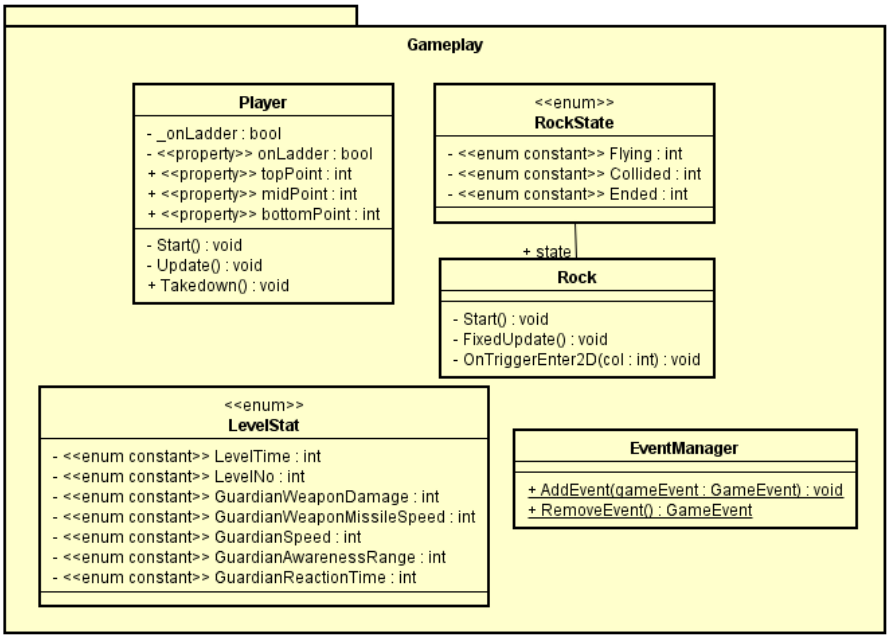


Figure 3: Detailed component diagram of Enemy package of the game.

2.4 Gameplay Package

Package that contains necessary classes for precise player-game interaction.

Player: Class that keeps the information about player's visual attributes such as width & length as well as necessary methods/operations to start visualizing.
Rock: One of the many gadgets we plan on implementing for the player.
RockState: State of the rock wheter it hit something or still airborne
EventManager: A class which interfaces between gameplay and AI, game adds events, and those events are sent to the AI to be evaluated.
LevelStat: An enumeration which holds level specific parameters.



3.0 Class Interfaces

3.1 AI package

Node

extends MonoBehaviour

Represent a node in pathing graph of a level. These nodes constitute vertices of level graph. This class is a MonoBehaviour(a predefined Unity class) class, which means Node object will appear on the Unity scene screen.

Fields
Inherited fields:
enabled, gameObject, tag, transform, GameObject, hideFlags, name

Methods
Inherited methods:
BroadcastMessage, CompareTag, GetComponent, GetComponentInChildren, GameObject, GetComponentInParent, GetComponents, GetComponentsInChildren, GetComponentsInParent, SendMessage, SendMessageUpwards, GetInstanceID, ToString

Graph

Represent AI pathing graph of a level. Based on this graph AI object decide where to go. Graph nodes exist on the scene, this class retrieves these nodes and creates a graph. Some pathing utility methods also exist in this class.

Fields
List<Node> nodes;

Stores nodes in the level. Positions of nodes creates a graph and these positions define weights of edges.

Dictionary edges;
Stores edges in the graph. These edges are directed.

Constructors
public Graph()
Creates graph and initializes field members. To do so, it retrieves all node object on the current scene and by using their positions and links, it creates a directed weighted graph.

Methods
public void CreateGraph()
Retrieves nodes and creates graph.

public List<Node> ShortestPath(Node source, Node target)
A dijkstra shortest path function. Given source and target nodes, this method tries to find shortest path between them. If such path is found, it returns as list of nodes. Null otherwise.
Parameters: source, source node; target, target node.
Return: List of nodes if a path is found, Null otherwise.

ReactionAI

AI class representing reaction rules of an enemy. These reaction rules depend on state of enemy, events and of course AI constant parameters given by machine learning module.

Fields
private Enemy enemy
References to an enemy object.

Constructors
public ReactionAI(Enemy enemy)
Initializes object by setting enemy references to constructor paramter.

Methods
public Action React(Event gameEvent)
This method define how enemy AI object will react when an event occurs. According to given event, enemy state and other AI parameter an Action is produced.

State

Defines the state of an enemy object.

Fields
int currentState
Represents current state of enemy. It takes one of predetermined integer values.

Constructors

public State()
Initializes object. Field members are initialized to default values.

public State(int currentState)
Initializes object and field members are initialized by given paramters.

Methods
Getter methods

public evaluateState(Event event)
According to event, this method determines the current state. For example, an spot event has occurred, this method may evaluate that event and results in changing currentState value from idle to hostile.

Action

A virtual class representing base of any <<Capabilility>>Action object such as ApproachAction, FireAction, ChaseAction etc.

Fields
public int priority
This field represents the priority of action. Higher this value is higher priority of the action. Depending on it, execution order of actions may change.

public bool done
Boolean value representing completeness of action.

Constructors
public Action(int priority = 0)
Initializes action with given priority.

Methods
Getter and setter methods.

public virtual void Execute
Executes the action. This method is generally overridden by child classes to define their action so implementation of this method highly relies on child classes.

ApproachAction

extends Action

Defines approach capability of enemy objects.

Fields
inherited fields: priority, done

private Vector2 target

This field stores reference to target point. Point is defined as a Vector2 struct. Since Gameplay package does not have any information about parameters of an action. This information should be transmitted via actions.

private IMovable source
Stores reference to moving object. It could be Guardian or any other enemy implementing IMovable interface. Since Gameplay package does not have any information about parameters of an action. This information should be transmitted via actions.

Constructors
public ApproachAction()
Initializes an empty ApproachAction object. field variables should be set via setter methods in order for this object to function properly. If field members are known before creating the object, other constructors should be preferred.

public ApproachAction(Vector2 target, IMovable source)
Initializes a ApproachAction object according to parameters. It set player and source references to field members.

Methods
Setter and Getter Methods

public override void Execute()
Executes defines action, in this case, source object tries to move towards target player. This execution may depend on state of enemy. For example, an enemy with hostile state may run whereas in other states, the enemy may walk.

ChaseAction
extends Action

Defines chase capability of enemy objects.

Fields
inherited fields: priority, done

private Player target
This field stores reference to chase target. Since Gameplay package does not have any information about parameters of an action. This information should be transmitted via actions.

private IMovable source
Stores reference to moving object. It could be Guardian or any other enemy implementing IMovable interface. Since Gameplay package does not have any information about parameters of an action. This information should be transmitted via actions.

Constructors

public ChaseAction()
Initializes an empty ChaseAction object. field variables should be set via setter methods in order for this object to function properly. If field members are known before creating the object, other constructors should be preferred.

public ChaseAction(Player target, IMovable source)
Initializes a ChaseAction object according to parameters. It set target and source references to field members.

Methods
Setter and Getter Methods

public override void Execute()
Executes defines action, in this case, source object tries to move towards target player. This execution may depend on state of enemy. For example, an enemy with hostile state may run whereas in other states, the enemy may walk.

FireAction
extends Action

Defines fire capability of enemy objects.

Fields
inherited fields: priority, done

private Player target
This field stores reference to fire target. Since Gameplay package does not have any information about parameters of an action. This information should be transmitted via actions.

private IFirable source
Stores reference to fire source. It could be Turret, Guardian or any other enemy implementing IFirable interface. Since Gameplay package does not have any information about parameters of an action. This information should be transmitted via actions.

Constructors
public FireAction()
Initializes an empty FireAction object. field variables should be set via setter methods in order for this object to function properly. If field members are known before creating the object, other constructors should be preferred.

public FireAction(Player player, IFirable source)
Initializes a FireAction object according to parameters. It set player and source references to field members.

Methods
Setter and Getter Methods

public override void Execute()
Executes defines action, in this case, source object fires at target player. This execution may depend on state of enemy.

ActionManager

It is an interface between Gameplay and AI classes. ActionManager class consists of some static members and functions. The purpose of this class is to construct link between AI and gameplay packages. General data flow of this class is as follows: AI classes add action(s), Gameplay takes action(s) and executes them.

Fields
static Queue actionQueue
First in first out queue object required to store actions. actionQueue is a System.Collections.Generic Queue class representing queue data structure for action objects.

Constructors
No constructor

Methods
public static void Init()
Initialize functions that initializes all static variables of the class. It must be called at just after level creation to make sure that any part of project does not call other methods which results in NullReference exception.

public static void AddAction(Action action)
This function adds an action (which is created by other AI package objects) to actionQueue. Added action stored here until it is retrieved by RemoveAction function. Parameters: action, Action to be added.

public static Action RemoveAction()
This function retrieves and removes an action in the actionQueue. Since this class is an interface between AI and Gameplay packages, this function constitutes Gameplay end of this communication. It means Gameplay packages use this function to get an action in the ActionManager. Returns: Action to be returned.

3.2 ML package

MLCommunicator

Methods
readMLFile(), writeMLFile(MLLevelStats levelStats)
These methods are for reading and writing the parameter tuning data file generated by the machine learning module

MLLogger

Fields
protected static MLLevelStats currentStats

This field references to the current status object of the player as MLLevelStats type which has the related statistical data about player's gameplay.

Methods
incrementStat(LevelStat stat, float value), decrementStat(LevelStat stat, float value)
These methods are to increment and decrement the statistical data with respect to gameplay of a particular player

setStat(LevelStat stat, float value)
This method sets the parameters of the level according to how player progress

MLLevelStats

Fields
private Dictionary<LevelStat, float> stats
Keeps the information about a particular level's statistical data including LevelTime, LevelNo, GuardianWeaponDamage, GuardianWeaponMissileSpeed, GuardianSpeed, GuardianAwarenessRange, GuardianReactionTime

Constructors
MLLevelStats()
Initializes an empty level statistics object.

Methods
getJson()
This method first converts the MLLevelStats object to the JSON data, then return that as a string

MLConfig

Fields:
protected static readonly bool isConfigureFromFile
This field is used for indicating whether the level configuration data is to be loaded externally or not and is set to False initially.
protected static readonly string configFilePath
This field indicates the configuration file path belonging to a specific player.

Methods:
ConfigureFromFile()
This method configures the ML module by setting specific parameters depending on a player with retrieving data from a file.

3.3 Enemy package

Enemy

reactionAI: A property type of ReactionAI which will used by nonplayable units later on.

actionQueue: A property type of ActionQueue to be used in queing the events to reflect them in chronologic order.

Start(): Method initializes a logical components of an enemy unit.
Update(): Function to update the pre-defined logical properties of a single enemy unit.

Guardian

Approach(target : Vector2) : Function that creates a logical approaching route for a subtype guardian.
Spot(GameObject) : Function to determine what to do by enemy units whenever they see our character.

ReactionAI

Fields
private Enemy enemy
References to an enemy object.

Constructors
public ReactionAI(Enemy enemy)
Initializes object by setting enemy references to constructor paramter.

Methods
public Action React(Event gameEvent)
This method define how enemy AI object will react when an event occurs. According to given event, enemy state and other AI parameter an Action is produced.

3.4 Gameplay Package

Player

onLadder: A boolean property which tells if the player is holding on to a ladder.

Start(): MonoBehaviour function which handles initializations.

Update(): A function called by the game engine every update interval so that the class can determine if it is latched on to a ladder, or midair as well as taking control inputs from the user and executing them. Practically handles the entire functionality of the class.

Takedown: Performs a takedown if an enemy is in range to neutralize an enemy on a given level.

Rock

Start(): Initialization stage function.

OnTriggerEnter2D(): Executed by the game engine when the rock collides with something, as this function handles the consequences of the event depending on what it hit.

state: State of the rock depending if it is airborne, collided or ended.

FixedUpdate(): Uses the physics update instead of Update for state updates since

EventManager

AddEvent(GameEvent): Adds an event to the game to be interpreted in the AI layer of the application.

RemoveEvent(GameEvent): Cancels the interpretation of a given event.