SORBONNE UNIVERSITÉ

M1 IPS SMR

Année universitaire 2024-2025

Programmation Objet Python

Compte-Rendu de Projet de Python

GRICOURT Axel

ZOUAK HAMZAOUI Ahmed-Amine

Introduction

Notre jeu est inspiré de Rainbow Six Siege. Nous l'avons adapté afin qu'il réponde aux attentes de l'énoncé : c'est une version 2D tour par tour : chaque joueur peut choisir une unité par tour, et ensuite elle peut soit se déplacer et utiliser une attaque spéciale soit se déplacer et utiliser une attaque normale. Le jeu se déroule en 3 étapes : Lors de la première phase, le joueur 2 doit placer ses défenseurs, ses barricades et ses barricades blindées. Lors de la deuxième phase, les deux équipes s'affrontent, pendant que l'équipe des attaquants essaie d'atteindre l'otage sans se faire éliminer, ou de directement anéantir l'équipe des défenseurs, les défenseurs doivent les empêcher. Lors de la troisième phase, qui se déclenche si une unité attaquante atteint l'otage, le joueur 1 doit déplacer son unité jusqu'à un des quatre points d'extraction (qui clignotent en rouge sur l'écran). Le joueur 2 doit faire tout son possible pour éliminer l'unité qui détient l'otage, sinon il perds la partie.

Choix Techniques et Concepts de Programmation

Pygame a été exploité pour gérer les aspects graphiques (affichage des unités, dessin et affichage des cartes et boutons), ainsi que les événements interactifs (clavier, souris). Cela permet un rendu fluide et une interface utilisateur intuitive.
L'héritage simplifie l'organisation et la maintenance du code. Et la composition permet de modéliser des relations fortes entre les composants du jeu.
Voici les fonctions clés de notre code :

- **Déplacement des unités** :
  - `get_movement_range` : Calcule les cases accessibles en fonction de la vitesse des unités et des obstacles du terrain(Breadth-First Search).
  - `move_unit` : Gère le déplacement et les interactions (kits de soin, otage, points d'extraction).
- **Portées et ligne de vue** :
  - `get_attack_range` : Détermine les cases accessibles pour une attaque.
  - `is_line_of_sight_clear` : Vérifie si une ligne de vue est dégagée (algorithme de Bresenham).
- **Interactions clés** :
  - `handle_healthkit_interaction` : Permet à une unité de se soigner avec un kit.
  - `check_hostage_interaction` : Capture et transport de l'otage.
- **Affichage** :
  - textttflip_display : Permet de mettre a jour l'affichage de l'écran
- **Attaques et capacités spéciales** :
  - `handle_attack` : Gère les attaques normales des unités selon leurs dégâts et défense
  - `handle_special_attack` : Gère les attaques spéciales des unités qui en possèdent
- **Carte et placement** :
  - `adapt_logical_map` : Ajuste la carte à la taille de l'écran.
  - `initial_placement_phase` : Gère le placement des unités et des barricades.
- **Victoire et affichage** :
  - `display_winner` : Affiche le gagnant et termine la partie.
  - `flip_display` : Affiche les zones accessibles et les points d'extraction.
  - `check_extraction_or_death` : Conditions de victoire liées à l'otage.
- **Clignotement des points d'extraction** :
  - `draw_extraction_points` : Gère l'effet visuel des points d'extraction.

Fonctionnalités Implémentées

Nous avons respecté le cahier des charges en implémentant 10 unités, 5 pour chaque équipe. Elles ont toutes des points de vie, des dégâts, une stat de défense et une stat de vitesse.

Parmi les 5 unités, 3 ont des capacités spéciales :

— `Médecin` : Soigne les unités de son équipe.
— `Démolisseur` : Casse les barricades blindées
— `Bombardier` : Lance des grenades et cause des dégâts sur un carré de taille 3x3

Les deux autres unités ont soit plus de points de vie et de défense (Tank) soit une vitesse de déplacement supérieure (Scout).

Nous avons 5 types de cases différentes :

— `Case simple` : Les unités peuvent s'y déplacer librement.
— `Mur` : Infranchissable et indestructible, les unités ne peuvent pas le traverser ni utiliser des attaques simples a travers.
— `Barricade normale` : Même principe que un mur infranchissable, mais elles peuvent être cassés avec une attaque normale par n'importe quelle unité.
— `Barricade blindée` : Même principe que une barricade normale, mais peut être cassée uniquement par les démolisseurs des deux équipes via leur attaque spéciale.
— `Point d'extraction` : Apparaît si l'otage a été sécurisé, permet aux attaquants de gagner en l'atteignant avec l'unité qui escorte l'otage.

Nous avons aussi la implémenté deux fonctions de calcul de portée, que ce soit pour le déplacement ou l'attaque.
De plus, les dégâts causés par les attaques dépend de la défense des unités qui les reçoivent :
Dégats = max(10,Attaque - Défense x 0.3)
Cela veut dire que une unité peut recevoir un minimum de 10 dégats, et le reste des dégats est calculé en enlevant la valeur de la défense multipliée par un facteur 0.3 a l'attaque de l'unité qui la lance.

Nous avons aussi ajouté des fonctionnalités supplémentaires :

— `Démolisseurs + Compétences pouvant modeler le terrain` : Les barricades blindées peuvent être détruites uniquement par ce type d'unité.
— `Objets que l'on peut ramasser sur le terrain` : Les kits de soin soignent jusqu'à 20pv a n'importe quelle unité qui passe par dessus, temps qu'elle ait moins de ses pv max (on ne peut pas dépasser les pv max).
— `Victoire par extraction` : Le fait de pouvoir gagner en libérant l'otage en le ramenant a un endroit spécifique

Pour rendre le jeu plus agréable, nous avons crée un menu d'accueil, un menu pause, un bouton "Help" pour un didacticiel et une image de fin lors de la victoire. Nous avons aussi intégré des sons pour les différents menu ainsi que des sons pour les actions. Nous avons affiché une carte d'information qui se mets a jour en temps réel et qui affiche le nom, le rôle, les dégâts, la défense, les points de vie et une image du personnage pour pouvoir le sélectionner et choisir quelle action effectuer, mais aussi mis une "Text Box" qui affiche un historique des actions effectuées pendant la partie.

Justification du Diagramme UML

Unit est la classe abstraite. Elle est utilisée comme base pour toutes les unités, mais elle n'a pas vocation à être instanciée directement. Elle fournit des attributs et des méthodes génériques que les sous-classes spécialisées implémentent ou étendent.

Les classes de personnages héritent de Unit pour réutiliser les mécanismes de base tout en ajoutant leurs spécialités (soins, attaques spéciales).Cela illustre la flexibilité de l'héritage pour modéliser des personnages variés sans réécrire le code commun.

En relations supplémentaires nous avons l'association simple : Card –> Unit. La classe Card utilise une instance de Unit pour afficher les détails du personnage. Cette relation est pertinente car elle représente un lien entre l'interface utilisateur (carte) et le modèle (unité). Nous avons aussi une composition : Game –> Hostage car Hostage est une partie intégrante du Game. Si le jeu est détruit, l'otage n'a plus de sens d'exister.

Le diagramme UML figure en page 5, mais nous l'avons aussi publié sur le git pour une meilleure lisibilité.

Conclusion et répartition

Le projet illustre l'application des concepts de POO à un jeu vidéo tactique, avec une architecture claire et extensible. Les choix de conception permettent une grande modularité et facilitent l'ajout de nouvelles fonctionnalités. Les améliorations possibles incluent l'ajout d'une intelligence artificielle pour les ennemis et des animations pour les actions des personnages. Pour la répartition des tâches, nous avons essayé de faire ça de la façon la plus équilibrée possible, et nous nous sommes mis d'accord sur tout le projet, que ce soit l'inspiration de Rainbow Six Siege, les personnages, leurs capacités, le visuel et les fonctionnalités. Nous établissions des objectifs clairs pour organiser notre travail en partant de la structure générale jusqu'aux petits détails. Si l'un ne réussissais pas la tâche, l'autre la faisais.


Game
- screen
- textbox : TextBox
- cell_size : int
- grid_size_x : int
- grid_size_y : int
- logical_map : list
- player_units : list
- enemy_units : list
- hostage : Hostage
+ handle_turn(active_units, opponents, pause_button, color) : void
+ move_unit(unit, opponents, pause_button, movement_range) : void
+ handle_attack(unit, opponents) : void
+ handle_special_attack(unit, opponents, special_action_range) : void
+ flip_display(...) : void
+ display_winner(winner : str) : void
Card
- unit : Unit
- screen
- width : int
- height : int
- background_color : tuple
- text_color : tuple
- selected_action : str
- attack_range : any
__init__(unit, screen) : None
+ get_role() : str
+ get_name() : str
+ update(unit=None, selected_action=None, attack_range=None) : None
+ draw(x, y) : None
Button
- image
- x_pos : int
- y_pos : int
- text_input : str
- font
- base_color
- hovering_color
+ update(screen) : void
+ checkForInput(position) : bool
+ changeColor(position) : void
TextBox
- screen
- width : int
- height : int
- font
- x : int
- y : int
- bg_color : tuple
- text_color : tuple
- messages : list
+ add_message(message : str) : void
+ draw() : void
Hostage
- x : int
- y : int
- controlled_by
- control_turns : int
- image_path : str
+ draw(screen) : void
Unit
- x : int
- y : int
- health : int
- max_health : int
- attack_power : int
- defense : int
- speed : int
- team : str
- role : str
- image_path : str
+ move(dx, dy, grid_width, grid_height) : void
+ attack(target) : void
+ take_damage(amount) : bool
+ draw(screen) : void
Doc
+ special_ability(game_map, target_unit) : tuple
Caveira
+ special_ability(game_map, target_unit) : tuple
Fuze
+ special_ability(game_map, target_x, target_y) : tuple
Montagne
+ special_ability() : void
Thermite
+ special_ability(game_map, target_x, target_y) : tuple
+ get_special_target_range(game_map) : list
Jackal
+ special_ability(game_map, target_x, target_y) : tuple
Jaeger
+ special_ability() : void
Smoke
+ special_ability() : void
Glaz
+ special_ability() : void