



Programmation Objet Python

Projet : Conception et Implémentation d'un Jeu de Tactique
Tour par Tour en 2D

League On Budget

M1 Ingénierie des Systèmes Intelligents (ISI1-1)

BOUHAI Yasser 21200183
CHELFAT Aya 21416394
HANAFI Ghozlene 21216169

16/12/2024

Sorbonne Université

Encadrants : Louis Annabi, Florence Ossart

Table des matières

1 R ´partition des Tâches entre les Membres du Groupe	2
2 Fonctions de Base et Supplémentaires	2
2.1 Fonctionnalités de Base	2
2.2 Fonctionnalités Supplémentaires	3
3 Explication des Relations Utilisées dans les Diagrammes UML	3
4 Classe Abstraite Abilities	3

1 RÉPARTITION DES TÂCHES ENTRE LES MEMBRES DU GROUPE

Class	Contributor(s)	Methods
Unit	Aya, Ghozlene, Yasser	Aya : <code>create_units</code> ; Ghozlene : <code>in_range</code> , <code>move</code> ; Yasser : <code>react_to_attack</code> , <code>attack</code> , <code>update_buffs_and_debuffs</code> , <code>revert_buff</code> , <code>revert_debuff</code> , <code>draw</code>
MonsterUnit & NexusUnit	Yasser	<code>react_to_attack</code> (overwritten in NexusUnit), <code>draw</code> (overwritten in MonsterUnit)
Sounds	Ghozlene	<code>play</code> , <code>set_volume</code> , <code>stop</code>
Abilities	Aya, Yasser	<code>use</code> , <code>get_targets_in_aoe</code> , <code>apply_effect</code> , <code>reduce_cooldown</code>
BuffAbility & DebuffAbility	Aya	<code>use</code>
DamageHeal Ability	Yasser	<code>use</code>
Tile	Yasser	<code>draw_tile</code>
Pickup	Yasser	<code>initialize</code> , <code>update</code> , <code>spawn_single_pickup</code> , <code>draw_pickups</code> , <code>picked_used</code> , <code>remove_pickup</code> , <code>get_random_spawn_location</code>
Grid	Ghozlene, Yasser	<code>create_grid</code> (by both), <code>draw</code> (Yasser)
Highlight	Yasser	<code>highlight_range</code> , <code>update_fog_visibility</code> , <code>draw_fog</code> , <code>show_buff_animation</code>
Game	Aya, Ghozlene, Yasser	Aya : <code>draw_abilities_bar</code> ; Ghozlene : <code>log_event</code> , <code>draw_info_panel</code> ; Yasser : <code>draw_units</code> , <code>basic_attack</code> , <code>advance_to_next_unit</code> , <code>handle_turn</code> , <code>get_respawn_location</code> , <code>manage_keys</code> , <code>draw_key_counts</code> , <code>main_menu</code> , <code>show_menu</code> , <code>check_game_over</code> , <code>game_over_screen</code> , <code>run</code>

2 FONCTIONS DE BASE ET SUPPLÉMENTAIRES

Dans cette section, nous décrivons les fonctions de base nécessaires au fonctionnement du jeu ainsi que les fonctionnalités supplémentaires implémentées pour enrichir l'expérience utilisateur.

2.1 Fonctionnalités de Base

- **Unités** : Implémentation de 5 types d'unités(joueurs) et 2 autres unités supplémentaires(Monster et Base) avec des caractéristiques distinctes : points de vie(HP), attaque, défense.
- **Compétences** : Trois types de compétences(abilities) définies avec des caractéristiques comme puissance, portée, zone d'effet et précision. Ces compétences sont de type attaque(attack) et soin(heal).
- **Cases** : Le terrain est composé de ces types de cases : eau,herbe,pierre,buisson,base
- **Déplacement** : Les unités peuvent se déplacer selon leur périmètre de déplacement, parcourant un nombre de cases déterminé et ou il a été ajouté que l'eau diminue le périmètre de mouvement.
- **Portée des compétences** : Les compétences ont des portées variées.
- **Zone d'effet** : Une compétence qui agit sur une zone entière infligeant des dégâts à toutes les unités présentes.

- **Calcul des dégâts :** Les dégâts infligés sont calculés à partir de la puissance de la compétence, de la statistique d'attaque et de la statistique de défense de l'unité ciblée avec une formule spécifique.

2.2 Fonctionnalités Supplémentaires

- **Objets :** Les unités peuvent ramasser des potions sur le terrain qui sont mises aléatoirement à chaque tour. Une fois ramassés, chaque potion donne un effet spécifique à l'unité.
- **Visibilité des ennemis :** Les unités d'équipes adverses ne sont pas visibles les unes pour les autres sauf si elles sont présentes dans le champs de vision d'une unité de son équipe (Fog).
- **Animations :** Implémentation d'animations spécifiques pour marquer la défaite des monstres.
- **Réincarnation :** Les unités peuvent revenir à la vie après leur mort, avec un mécanisme de réincarnation.
- **Système de clés :** Ajout d'un système de collecte de clés, nécessaire pour abattre les bases et remporter la victoire : La base ennemie ne peut pas être attaquée tant que trois clés n'ont pas été collectées par les unités du joueur.
- **Système d'assassinat :** Quand un joueur entre dans un buisson il devient invisible même si il est dans le champs de vision de l'ennemi ,et si un ennemi y entre elle sera tuée automatiquement
- **Types de dégats et de défense :** On a introduit des dégâts physiques et dégâts magiques.

3 EXPLICATION DES RELATIONS UTILISÉES DANS LES DIAGRAMMES UML

Les diagrammes UML illustrent les relations entre les classes principales du projet. Nous utilisons principalement les relations suivantes :

- **Relation d'héritage**

MonsterUnit → Unit et BaseUnit → Unit : Les deux classes héritent de **Unit**. Elle surchargent les méthodes `react_to_attack` et `draw` respectivement.

BuffAbility & DebuffAbility & DamageHealAbility → Abilities : Les classes **BuffAbility** et **DebuffAbility** spécialisent **Abilities** en redéfinissant la méthode `use` pour appliquer respectivement des buffs ou des debuffs, tout en conservant les attributs communs comme le coût en mana et le `cooldown`.

- **Composition**

Unit → Abilities : Une unité possède une ou plusieurs capacités (**Abilities**), comme attaquer ou soigner. Cela permet à l'unité d'agir et d'interagir avec d'autres unités de manière flexible..

Grid → Tile : Chaque **Grid** contient des instances de **Tile**, qui forment la structure de la grille. Les **Tile** dépendent fortement de **Grid**, et si celle-ci est détruite, les **Tile** seront également détruites.

Game → Pickup : La relation entre **Game** et **Pickup** est une composition, car **Game** crée, gère et contrôle entièrement la logique et l'affichage des **Pickup**, qui ne peuvent pas exister indépendamment de **Game**.

Game → Sounds : **Sounds** est utilisé par **Game** mais reste indépendant. **Game** appelle des méthodes de **Sounds** (ex. `play`, `stop`) mais ne crée pas directement l'objet. Si **Game** est supprimé, **Sounds** peut encore exister et être utilisé ailleurs.

- **Agrégation :**

Game → Highlight : La classe **Game** utilise **Highlight** pour gérer les zones visibles ou interactives, comme les portées d'attaque ou de déplacement. **Highlight** est indépendant et peut exister sans **Game**, mais il est essentiel pour l'affichage interactif.

Game → Unit : La classe **Game** gère les unités (**Unit**), qui représentent les personnages ou ennemis du jeu. Les unités sont indépendantes de **Game**, mais elles n'ont un rôle que dans le contexte du jeu, où leurs actions sont orchestrées.

- **Association :**

Game → Grid : La classe **Game** utilise **Grid** pour gérer les déplacements et actions sur le terrain, mais **Grid** reste indépendant et réutilisable dans d'autres contextes, comme un éditeur de cartes..

4 CLASSE ABSTRAITE ABILITIES

La classe **Abilities** a été conçue comme une classe abstraite pour établir une structure commune à toutes les compétences (abilities) du jeu, tout en offrant une grande flexibilité dans leur implémentation.

Elle définit la méthode abstraite `use`, qui doit être redéfinie par toutes les classes qui en héritent, afin de permettre des comportements spécifiques pour chaque type de compétence.