ALBA PASTOR Jaime
ANJOU Raphaël

Gameplay

- Élément Gameplay.I : Interface Graphique (Images à la fin du rapport)
    - **GraphicVariables(object)**: classe pour gérer tous les attributs et méthodes liées à l'interface graphique et donc à pygame.
        - **__init(self, hero)__**: charge toutes les images nécessaires pour l'interface graphique ainsi que quelques variables très utiles à la propreté du code (sinon les méthodes auraient beaucoup trop d'arguments)
        - **draw_gui(self, state)**: méthode qui dessine les informations du jeu (hp, expérience, inventaire, etc) ainsi que les messages.
        - **draw_map(self)**: méthode qui dessine uniquement la floor du jeu et le brouillard.
        - **draw_elements(self, monster_state)**: méthode qui dessine uniquement les éléments présents sur la map.
        - **draw_hero_move(self)**: méthode qui dessine les animations du hero en fonction de la direction dans laquelle il bouge.
        - **draw_game_screen()**: appelle *draw_map(), draw_elements() et draw_hero_move()* pour tout dessiner à la fois
        - **draw_message(self, time)**: méthode qui appelle *Game.readMessages()* et qui les affiche sur l'écran avec pygame pendant un temps (time).
        - **draw_menu(self, list_menu)**: méthode qui affiche la liste introduite pour pouvoir choisir différentes options. Ajoute '>' à l'option sélectionnée.
        - **draw_trader(self, list_objects)**: méthode qui appelle *draw_menu()* et qui ajoute les objets disponibles pour acheter.
        - **player_plays(self, event)**: méthode permettant de gérer les pygame events quand le joueur bouge ou choisit une action (*choose_action()*)
        - **choose_action(self, event)**: méthode qui gère les événements liés aux actions du joueur .
        - **choose_in_menu(self, event)**: méthode qui gère les événements lorsqu'on choisit une option dans le menu
        - **choose_in_inventory(self, event)**: méthode qui gère les événements lorsqu'on choisit une action dans l'inventaire
        - **change_hero_appearance(self, costume)**: méthode qui change l'apparence du hero par celle introduite
        - **select_from_inventory(self, item_chosen_class)**: méthode qui sélectionne l'item de l'inventaire qui correspond avec la classe introduite
        - **update_fog(self, actual_map)**: méthode qui actualise le brouillard en fonction de la position du hero
        - **play_next_song(self)**: méthode qui joue la chanson suivante de la liste. Appelée quand la chanson finit. 3 chansons disponibles

- Elément Gameplay.II : Étages
    - Ajout de l'attribut d'instance floor_number de la classe Map (correspond aussi avec le niveau de l'étage)
    - Ajout des attributs d'instance dans la classe Game:
        - level: niveau atteint
        - nb_floors: nombre d'étages total
        - floor_list: liste avec les différents étages
    - Le changement d'étage est géré par les escaliers (expliqués plus tard)

- Élément Gameplay.III : Point d'expérience
    - **Hero.gain_xp(self, xp_point)**: méthode permettant d'ajouter le nombre *creatureXP* à son niveau d'expérience. Vérifie ensuite si le héros gagne un niveau, le cas échéant elle appelle la méthode *Hero.gainLevel(self, nbOfLevel).*
    - **Hero.gain_level(self, nb_of_level)**: méthode permettant de donner un certain nombre de niveaux en plus au héros et de gérer les bonus donnés au héros.

- Élément Gameplay.IV : Inventaire limité
    - **Hero.delete_item(self, elem, throwing=False):** méthode pour supprimer l'élément *elem* de l'inventaire du héros.
    - **Hero.checkInventorySize(self):** méthode appelée à chaque début de tour pour vérifier que le héros a bien 10 objets maximum dans son inventaire. On peut spécifier la taille initiale de l'inventaire avec l'attribut *Hero.default_inventory_size*.

- Élément Gameplay.V: Déplacements Intelligents:
    - **Map.direction(self,c1,c2)**: renvoie une direction intelligente pour aller de la coordonnée c1 à la c2. Elle cherche la case la plus proche de c2 si la case et vide, sinon elle prend la deuxième la plus proche

- Élément Gameplay.VI et VI : Nuage de visibilité
    - Attribut d'instance de la classe Map:
        - graphic_map : liste dans une liste qui contient des couples: image et booléen. Lorsque le booléen est True la map s'affiche, sinon il y aura du brouillard
    - **GraphicVariables.update_fog()**: modifie les booléens des cases à proximité de l'héro pour que la map soit visible.
- Élément Gameplay.VIII : Diagonales
    - Ajout de touches de contrôles grâce à pygame.
    - **GraphicVariables**.**player_plays(self, event)**: méthode permettant de gérer les pygame events quand le joueur bouge ou choisit une action (*choose_action()*).

Diagonales disponibles avec les touches q,e,z et c en qwerty et a,e,w et c en azerty.

Actions :

- Élément Actions.I : Jet
    - **Hero.throw_item(self):** Permet de jeter l'item sélectionné ou alors l'arme équipée (cela dépend de l'input utilisateur). Comportements différents en fonction de si l'arme agit comme un boomerang ou non. Comportements différents en fonction des éléments rencontrés sur sa trajectoire. L'utilisateur utilise *Hero.choose_direction()* pour choisir la direction vers laquelle l'objet est lancé. Par défaut, les objets sont lancés à 5 cases du héros.

Objets :

- Élément Objets.I : Nourriture
    - **Hero.verifyStomach(self):** méthode pour vérifier que le héros n'a pas l'estomac vide. Cette fonction est appelée à tous les tours de jeu. De la nourriture est enlevée du héros tous les 20 tours dans la méthode *Game.play()*.
    - **Food item :** Ce sont des équipements avec l'usage *FeedEffect*.

- Éléments Objets.II : Armes
    - **Class Weapon(Equipment):** Gestion des armes. Elle dérive de *Equipement* et peut donc avoir un *usage* qui servira pour définir l'utilité de chaque arme.
        - **__init__***(self, name, abbreviation="", price=1, damage=1, launching_damage=1, come_back=False)*
    - **Creature.hit(self, other):** Si la créature qui frappe à une arme, l'autre créature perd un nombre d'hp égal à l'attribut *damage* de l'arme.
    - **Creature.equip_weapon(self, weapon):** Permet d'équiper une arme dans son slot désigné, s'il y a une arme dans ce slot celle-ci est remise dans l'inventaire.
    - **Creature.remove_current_weapon(self):** Permet de retirer l'arme actuellement dans le slot d'armes.
    - **Creature.has_weapon(self):** Return un booléen en fonction de si la créature a une arme ou pas.
    - **Creature.current_weapon(self):** Return l'arme actuellement dans le slot de la créa ture.
- Éléments Objets.II : Armes de jet
    - Utilisation de la méthode **Hero.throw_item(self)** suite à l'appui d'une touche par l'utilisateur. Celle-ci lance l'arme du *weapon_slot* qui est ensuite gérée dans cette même méthode.

Salles :

- Éléments Salles.I: Gestion des salles
    - **RoomObjects(Element)**: classe pour les éléments de la salle (escaliers et marchand)
        - **meet(self, hero)**: méthode qui appelle l'usage de l'objet lorsqu'il rencontre le hero.
        - **go_upstair()**: méthode statique qui fait l'héro monter d'étage. Elle correspond à l'usage des escaliers vers le haut.
        - **go_downstair()**: méthode statique qui fait l'héro descendre d'étage. Elle correspond à l'usage des escaliers vers le bas.
        - **meet_trader()**: méthode statique qui appelle *GraphicVariables.draw_trader()* en insérant une liste d'équipements aléatoires. Elle correspond à l'usage du marchand.
    - Ajout de l'attribut d'instance special_objects (liste). Pour stocker les différentes salles spéciales on ajoute des salles prédéfinies dans un attribut de la classe Game de type dictionnaire avec les différents special_objects correspondants.
        - **Room.rand_empty_coord(self, map):** méthode qui renvoie une coordonnée vide d'éléments
        - **Room.rand_empty_middle_coord(self, map)**: méthode similaire à *rand_empty_coord()* mais la coordonnée renvoyée doit être entourée de cases vides d'éléments
        - **Room.decorate(self, map)**: méthode que décore la salle avec un équipement, un monstre et les roomObjects présents dans la liste specialObjects
- Éléments Salles.II: Boutique
    - Ajout du trader/marchand et appelle un menu avec les objets aléatoires disponibles à vendre
        - **meet_trader()**: méthode statique qui appelle *GraphicVariables.draw_trader()* en insérant une liste d'équipements aléatoires. Elle correspond à l'usage du marchand.
        - **GraphicVariables.draw_trader(list_objects)**: méthode qui appelle *draw_menu()* et qui ajoute les objets disponibles pour acheter.
        - **Hero.buy(objet)** ajoute l'objet à l'inventaire s' il y a de la place et si l'héro a de l'argent suffisant. Alors il soustrait le prix de l'objet de l'argent

Monstres :

- Éléments Objets.I : Poison
    - Les créatures ont un *powers_list* qui stocke les pouvoirs de la créature. Les monstres avec l'effet poison ont donc le *PoisonEffect* ajouté lors de leur définition.
    - Les effets de *powers_list* sont appliqués suite au **Creature.meet** qui appelle lui-même **Creature.hit** (les effets sont gérés dans cette méthode).

Effets et pouvoirs :

- Classe Effect(object)
    - **Effect.__init__(self, creature)**: Initialise : game, creature, value, info, duration, level, graphicOutput
    - **Effect.delete(self):** Supprime un élément de la liste des effets actifs actuellement sur n'importe quelle créature.
    - **Effect.update(self):** réaliste l'action d'un effet et lui enlève 1 de durée s'il en a.
    - **Effect.action(self):** ajoute le message d'action propre à chaque effet.
    - **Effect.add_effect(self):** ajoute le message d'ajout propre à chaque effet.
    - **Effect.activate(self):** ajoute le message d'activation propre à chaque effet et l'ajoute à la liste des effets actuellement en cours du jeu.
    - **Effect.deactivate(self):** ajoute le message de fin propre à chaque effet. et utilise *Effect.delete(self)*
    - **Effect.clear(unique):** enlève tous les effets du héros.
- Class EphemeralEffect(Effect)
    - **Effect.activate(self):** super().activate et gestion de la durée de l'effet.
    - **Effect.deactivate(self):** super().deactivate et gestion du message donné.
- *Class HealEffect(EphemeralEffect)* | *PoisonEffect(EphemeralEffect)* | *FeedEffect(EphemeralEffect)* | *HungerEffect(EphemeralEffect)* | *TeleportEffect(EphemeralEffect)* :
    - Toutes les classes citées ci-dessus ont un fonctionnement similaire. Il y a une définition de **action(self)** qui gère ce que fait l'effet
- ConstantEffect(Effect)
- *WeaknessEffect(ConstantEffect)* | *StrengthEffect(ConstantEffect)* :
    - Ces effets ne sont appliqués qu'une seule fois mais ont aussi une durée. Il y a une redéfinition par rapport à Effect et *ConstantEffect* de **activate(self)** et **deactivate(self).** Création des méthodes **action(self)** pour gérer l'action de l'effet.