Rapport Projet Python Orientée Objet

Sharaine MALARVIJY, Florian THOLIN, Amaury JACOB
Master SI

I/ Résumé des classes

player.py

Player : Représente le joueur dans le jeu.
Inventory : Gère les objets et consommables du joueur.
Room : Représente une salle sur la carte.
Map : Représente la carte et gère la disposition des salles.

navigation.py

NavHandler : Classe enfant de EventHandler, gère les entrées clavier pour contrôler le joueur.

Nav : Gère la logique globale de navigation, d'ouverture de salles, d'objets, et de conditions de jeu.
Méthodes principales :

- player_move : Gère la présence d'une salle avec porte ouverte et le lancement de l'écran de sélection des salles
- open_room : ajout d'une salle dans la map avec des objets et effets puis l'enlève de la pool.

Effect : Applique des effets spéciaux liés à certaines salles.

ui.py

UI : Gère l'interface utilisateur globale du jeu (explication plus détaillée dans la partie III)

Classes dans le fichier ui_lib

Window : Gère la création, la taille et l'affichage de la fenêtre pygame.

Screen : Classe abstraite pour tous les écrans de l'UI.

- LoadScreen : Écran de chargement et précharge toutes les ressources graphiques du jeu avant l'affichage du menu principal.
- MainScreen : Écran principal du jeu
- SelectRoom : Écran menu de sélection de pièces
- Explore : Écran d'exploration des salles pour afficher et sélectionner des items.
- Shop : Écran d'achat d'objets consommables.
- EndScreen : Écran de victoire et de défaite, propose d'abandonner ou de recommencer la partie

Image : Classe abstraite servant de boîte à outils pour manipuler les images (chargement, mise à l'échelle...).

- ImageSimple : Gère une image unique avec position et version mise à l'échelle.

- ImageReapeated : Gère une image répétée à plusieurs positions.
- ImageRoom : Gère les images des salles et pré-génère leurs 4 rotations pour un affichage optimisé.

grids.py : Grilles d'affichage des éléments de l'écran principal
Ces classes gèrent en interne l'affichage et le positionnement des images avec leurs textes associés (s'il y en a).

- Consumable_grid : Tableau des consommables (coins, dés, etc) avec leurs compteurs. Cette grille est réutilisée pour les écrans Shop et Explore.
- Permanent_grid : Grille de 5 colonnes des objets permanents équipés.
- Map_grid : Affiche la carte du manoir : les salles avec leurs rotations.
- door : Affichage de la position du joueur par rapport aux portes de la map

EventHandler : Classe parente des gestionnaires d'événements (handlers) pour les différents écrans de l'UI.

Lors de l'appel de l'event_listener (méthode de l'UI qui poll les events et appelle la méthode associée dans le gestionnaire dite 'handler' de l'event), le handler exécuté sera celui défini dans l'attribut screen.event_handler de l'UI. L'attribut screen de l'UI représente l'écran courant.
Ainsi screen.event_handler est un objet héritant de EventHandler qui est définie dans la classe Screen de l'écran actuel.

Grâce à cela l'event_listener va donc appeler par polymorphisme le gestionnaire d'évènement de l'écran actuel, et EventHandler étant parent définit les handlers par défaut.

II/ Plan d'organisation des classes

Le programme a d'abord été divisé en parties majeures qu'il a semblé être intéressant de créer pour découper le projet :

- Le fichier main : intentionnellement presque vide, ne comprenant que la boucle de jeu principale
- La classe de Navigation (appelée Nav) : Gère la logique globale du jeu.
- L'UI (User Interface) : l'interface Utilisateur faisant l'intermédiaire entre la Navigation et le User (player), il réalise l'écoute sur les inputs du clavier et toute la partie Graphique (GUI). Au vu de sa complexité il a été subdivisé en plusieurs sous-classes dans ui_lib appelées par la classe UI principale.
- Player : rassemblant toutes les structures de données (et méthodes) liées à la partie de jeu en cours. Il comprend la classe Inventory (objets consommables et permanents) et Map (grille des Rooms)
- Le fichier database : l'ensemble des données de départ comprenant les noms des objets, la liste et composition des rooms disponibles...

Cette séparation de l'UI et du contrôleur de jeu (Nav) permet de les rendre indépendants et modulaires.

Ainsi on peut tout à fait utiliser le contrôleur de jeu avec une autre Interface Utilisateur (graphismes et inputs) et inversement.

Le but était de pouvoir travailler indépendamment sur ces 2 grosses classes centrales du programme.

On aurait aussi pu utiliser une Interface Graphique minimale (dans le terminal) pour développer Nav, mais cela n'a en fait pas été utilisé car l'UI était déjà suffisamment développée.

L'une des conséquences est que la gestion des inputs des menus (choix des rooms, des items dans shop, explore, ...) est réalisée intégralement dans l'UI, contrairement à l'écran principal. En effet si on était dans le terminal on demanderait un choix avec des numéros (choix 1, choix 2, …), alors que l'écran principal nécessite un traitement plus direct des inputs pour déplacer le joueur, réalisé ainsi dans Nav.

III/ Développement de UI

L’un des objectifs définis dès le début du projet était de rendre l’interface graphique « responsive », c’est-à-dire capable de s’adapter dynamiquement aux redimensionnements de la fenêtre.

Cela a engendré la nécessité de faire tous les positionnements en relatif à la taille de l'écran, de dissocier les fonctions 'build' et 'blit', respectivement servant à construire l'écran graphique et à l'afficher (blitter sur le buffer pour être exact, l'affichage étant réalisé par flip dans la boucle de jeu).

L’UI a beaucoup évolué au cours du développement du programme, c’est la première partie du projet par laquelle on a commencé. Elle a subi plusieurs changements de stratégies et de structures de données, résultant à plusieurs restructurations et réécritures de certaines fonctions. Nous sommes actuellement à la 3ème, presque 4ème version de l’UI.

Parmi les changements les plus notables se trouvent la gestion des évènements.

A l'origine, dans la version 2 de l'UI, on affectait les 'event handlers' dynamiquement, de façon similaire aux microcontrôleurs.

Ce qui nécessitait de mémoriser les modes du jeu (écran principal, sélection des rooms,...).

De plus il fallait mémoriser la fonction à utiliser pour reconstruire et afficher l'écran actuel en cas de redimensionnement.

A la version 3 de l'UI, les motifs du code ont fait apparaître une classe : la classe Screen.

Elle permet de changer très facilement les handlers d'évènement grâce une classe héritant de Event_Handler qui définit les handlers par défaut.

Et de passer tout aussi facilement (par polymorphisme également) la fonction que l'UI doit appeler lors d'un redimensionnement d'écran.

Il ne suffit alors que d'avoir un objet 'screen' dans l'UI représentant l'écran actuel.

La classe Screen est donc le résultat de ce défi réussi de faire une interface responsive.

Il est également intéressant de remarquer que toutes les ressources du jeu sont chargées au début. Un choix discutable qui permet d’éviter des latences dans l’expérience de jeu.
Un soin a même été apporté pour essayer de rendre l’écran de chargement « responsive » (on peut quitter, redimensionner l’écran), par des appels de l’event_listener, et simuler un comportement asynchrone.