

## Compte Rendu Projet IPS GROUPE 2

Avant de présenter notre travail et de justifier nos choix d'implémentation tout le long du rapport, il est primordial de revoir certaine base concernant le jeu :



Figure 1 : Grille du jeu.

Ce jeu se nomme « Blue Prince », il se joue comme suit :

Visuellement : l'interface principale est un rectangle avec juste deux cases que l'on peut voir au début : le bas milieu est le commencement : L'Entrance Hall, et le Haut du milieu est la pièce que l'on souhaite atteindre : L'Antichambre. Une autre partie du jeu représente l'inventaire du joueur, et enfin, une case pour voir les différentes pièces proposées.

Tout d'abord, le joueur en début de partie se trouve dans l'Entrance Hall, qui est la pièce d'entrée de jeu à chaque partie. Le but du jeu est d'avancer en utilisant ZQSD, pour bouger et changer de salle afin d'atteindre l'Antichambre.

La suite de ce rapport va donc justifier l'implémentation du jeu en trois parties :

- La création d'un fichier classe ;
- La création d'un code principale réunissant les différentes fonctions et la boucle principale ;
- L'implémentation de l'aléatoire tout le long du code principal.

## Introduction :

Lors de la réalisation du projet, nous avons adopté la méthode POO (Programmation Orientée Objet).

Pour rappel, la Programmation Orientée Objet (POO) consiste à organiser le code non pas autour de fonctions, mais autour d'entités appelées objets, qui sont des instances de classes. Ce modèle vise à rendre le code plus structuré.

Nous avons appliqué deux principes fondamentaux de la POO : l'Encapsulation et l'Abstraction.

L'Encapsulation permet de regrouper les données (attributs) et les comportements (méthodes) au sein d'une même classe (à l'aide d'un constructeur pour l'initialisation du joueur par exemple).

À titre d'exemple, la classe Joueur gère en interne son propre inventaire et ses mécanismes (perdre\_pas() par exemple). Le reste du programme n'a donc plus besoin de connaître les détails de cette gestion, il lui suffit d'appeler la méthode.

L'abstraction nous a permis de nous concentrer uniquement sur ce qui est important pour chaque élément du jeu (le Joueur, la Salle, les Objets) dans le code principal, sans nous préoccuper des détails complexes de leur fonctionnement interne.

Nous avons tout d'abord créé un fichier ne contenant que les définitions des classes (les "modules" des objets du jeu). À part, nous avons élaboré notre code contenant le fonctionnement principal du jeu. Par ailleurs, nous avons inséré toutes les fonctions et catalogues du jeu dans le code principal.

Dans le dossier principal nous avons ensuite fait appel au fichier classe (en utilisant import).

Ce choix nous est apparu comme le plus efficace car il évite un fichier principal trop long, séparant la définition des objets (les classes). Nous n'avons pas séparé les fonctions car leurs implémentations et l'implémentation du main nécessitent des lignes de code intégrées au code principal.

Concernant le dépôt GitHub, nous avons rencontré une difficulté initiale avec l'espace de travail créé.

→ Des problèmes d'autorisation ou de configuration (malgré la création en mode "public") ont empêché certains membres du groupe déposer leurs documents à tour de rôle. Ce problème technique a été la cause du retard du premier dépôt.

Nous avons depuis résolu cela par la création d'un nouveau repository fonctionnel pour garantir que toutes les contributions soient bien enregistrées, conformément aux exigences du projet.

## I. Création d'un fichier classe :

Le fichier `classes.py` modélise les différentes entités dynamiques du jeu. Nous avons ainsi défini quatre classes principales dans ce fichier.

En premier lieu, la classe `Joueur` est la création du personnage principal du jeu. Elle encapsule l'état complet du personnage en temps réel (position, inventaire, nom).

Elle contient aussi des méthodes (sous forme de fonctions au sein de la classe ; par exemple : `deplacer` et `perdre_pas`).

Celles-ci, constituent l'interface permettant au reste du programme d'interagir avec l'état du joueur (son inventaire).

Ensuite, nous avons conçu la classe `Salle` qui modélise chaque case du manoir.

La classe `Salle` modélise les cases (définies par la grille représentant le manoir) du plateau, gérant leur état (ouverte, verrouillée) et leur comportement (déclenchement d'effet pour les pièces contenant par exemple un coffre).

En effet, chaque instance (objet) de cette classe est responsable de son propre état, notamment si elle est découverte, son niveau de verrou potentiel, et si son effet\_declenche a déjà eu lieu ou pas.

Nous y avons intégré un attribut `effet_declenche` au niveau de l'objet lui-même pour permettre à chaque salle de gérer sa particularité (par exemple dans la cuisine, on peut gagner des aliments et ses aliments font gagner des pas au joueur).

Procéder ainsi, nous a permis de nous assurer que les bonus du jeu ne sont collectés qu'une seule fois selon les pièces.

La méthode `declencher_effet` illustre en fait l'interaction entre la salle et le joueur.

Elle agit directement sur l'état du joueur (son inventaire).

Enfin, la classe `ObjetCollectable` sert à représenter tous les objets que le joueur peut trouver dans le manoir.

Son constructeur reçoit quatre informations essentielles : le nom de l'objet (comme une pomme ou le kit de crochetage), la ressource de l'inventaire qu'il modifie (par exemple les pas ou les clés), le montant à ajouter à la ressource (pour la pomme, il gagne +2 pas par exemple), ainsi que le type de l'objet (s'il est consommable ou permanent).

Comme pour la classe `salle` la méthode `appli_effets()` applique directement l'effet de l'objet au joueur.

Elle met à jour les valeurs de son inventaire et, lorsqu'il s'agit d'un objet permanent, elle active aussi le drapeau (appelé le « flag » dans le code) associé dans `joueur.objets_permanents`.

Cela permet au jeu de gérer naturellement des capacités spéciales, comme l'ouverture de portes ou l'amélioration des chances de trouver des ressources.

Enfin, une autre structure, `PLACEMENT_OBJET`, est utilisée pour associer les divers objets collectables à une position aléatoire sur la grille du jeu au début de la partie.

On pourra ainsi plus facilement indiquer dans l'inventaire du joueur s'il a déjà été ramassé ou non.

## II. Architecture du 'pipeline' principal du jeu :

Afin de concevoir le fonctionnement principal du jeu avec toutes ses fonctionnalités, nous avons fondé notre code sur la bibliothèque Pygame, utilisée comme la fondation technique de l'interface graphique et de la gestion des événements du jeu. Au-delà de son rôle moteur de jeu, Pygame a été essentiel pour la phase d'initialisation du jeu. Ce module nous a notamment permis de créer la fenêtre du jeu, de définir le thème du jeu (couleurs, polices pour affichage de l'inventaire et des messages). De plus, nous avons pu préparer l'espace où sera dessiné le manoir, les salles et le joueur. Pygame nous permet de créer une grille avec des lignes colonnes avec des tailles que nous avons définies.

Cette première étape consiste donc à préparer le canevas de l'application avant d'entrer dans la boucle principale qui va régir la logique du jeu.

À ce stade, la programmation orientée objet prend le relais grâce à l'appel des classes définies dans les fichiers du projet.

Le fichier Projet-Python-Principale organise les étapes essentielles suivantes :

- Initialisation du jeu : cette étape est la base visuelle et technique nécessaire avant toute interaction, commençant par la création de la fenêtre avec `pygame.display`, titré 'Blue Prince' et la définition de la taille de la grille pour préparer l'espace de jeu.
- Chargement des images des pièces : nous avons mis en place un dictionnaire `Chemin_images` qui regroupant les ressources graphiques, puis chargement des visuels pour une gestion simplifiée et structurée des ressources.
- Catalogue des pièces : afin de gérer les interactions du joueur et d'assurer la cohérence des règles, nous avons implémenté le dictionnaire catalogue de pièces afin de centraliser toutes les informations des salles et leurs caractéristiques (coût, effet, état de verrouillage).
- Fonctions principales d'affichage et de gestion : ce fichier est composé d'un ensemble de fonctions telles que : `afficher_inventaire`, `afficher_choix_pieces`, `afficher_defaite` et `afficher_victoire`, elles assurent à la fois la visualisation de l'inventaire, la présentation des choix possibles et la gestion des écrans pour signaler la fin de la partie (défaite ou victoire).
- Boucle principale du jeu : la fonction principal gère l'ensemble des mécaniques déplacements du joueur via les touches ZQSD, tirage des pièces, mise à jour de l'inventaire et les conditions de victoire ou de défaite.

Cette boucle constitue le cœur du pipeline. Elle relie les classes (Joueur, Salle, Objets) aux fonctions d'affichage et illustre la continuité méthodologique.

### III. Implémentation de l'aléatoire :

Afin d'obtenir un jeu permettant de rejouer sans obtenir les mêmes objets et une même configuration du manoir, nous avons implémenter sur le fichier principal des fonctions, et bout de code permettant de réaliser ceci :

- Une fonction permet d'obtenir différent niveau de verrou : 0, 1, et 2 de manière totalement aléatoire, ces niveaux sont ensuite assignés aux salles présente dans le catalogue présenté plus tôt. Afin de ne pas avoir de difficulté dès le début du jeu, étant donné que le nombre des clés et des permettant d'ouvrir ou de tirer à nouveau trois salles est initialiser à 0 et que le joueur à peu de chance de trouver immédiatement un kit de crochetage. Le degré de difficulté du jeu sera donc accru si nous ne considérons pas un autre critère lors du choix des niveaux : la probabilité de trouvé un niveau élevé augmente au fur et à mesure des étages du manoir, sur les premiers étages on ne trouve que des niveaux = 0, puis un mélange avant d'arriver aux derniers étages près de l'Antichambre, où les niveaux seront = 2.
- On a ajouté une extraction de notre catalogue de manière aléatoire pour obtenir à chaque fois un set différent de salle, sachant que chaque salle à son degré de rareté, ce degré permet de calculer la probabilité d'obtenir une pièce au tirage aléatoire. Bien entendu, afin que le joueur ne soit pas bloqué, on s'est assuré qu'au moins une salle sur les trois ne demandé pas de gemmes pour choisir la salle.
- Nous avons aussi implémenté une classe et un bout de code, qui permet de gérer l'aspect aléatoire de la dispersion d'objet que l'on peut collecter : tout ce qui est nourriture, tous les objets permanents, les clés et les gemmes. Pour cela, nous avons assigné aléatoirement les objets dans le manoir, en s'assurant qu'ils ne soient jamais placés dans les cases dédiées à l'Entrance Hall et l'Antichambre. Cette implémentation permet de diversifier l'expérience du joueur à chaque partie.

→ L'ajout de l'aléatoire (qui se fait avec le module 'random') est un élément essentiel à tout jeu, même un simple jeu de carte doit présenter une pioche différente à chaque partie, c'est ce que l'on souhaite obtenir dans ce jeu « Blue Prince ».

### Conclusion :

Nous avons implémenté la plupart des fonctionnalités demandé dans la grille d'évaluation principale, il ne manque que l'ajout des effets de probabilité du détecteur de métaux et patte de lapin, ils sont bien présents dans le jeu en tant qu'objet permanent, mais n'ont pas encore d'effet sur la chance de trouver des clés ou certains autre objet dans le manoir.

Mais le code présenté réalise bien le reste des fonctionnalités de la grille.

En outre, ce projet nous a permis de nous exercer sur l'utilisation de git en joignant notre dépôt local au dépôt distant commun afin d'ajouter, de corriger et de modifier le code et les différents fichiers de manière continus, malheureusement, comme expliqué plus haut, nous avons eu quelque problème de configuration qui a ralentie notre progression et nous a contraint à initialiser un nouveau répertoire public pour déposer nos commits respectifs.