Rapport - POO

Introduction

Ce rapport vise à détailler et justifier l'architecture logicielle du projet, qui est l'implémentation d'une version simplifiée du jeu vidéo *Blue Prince* en Python. L'objectif principal de la conception était d'appliquer rigoureusement les concepts de la programmation orientée objet afin de séparer la logique métier (les règles du jeu) de l'interface utilisateur (le dessin et l'affichage avec Pygame), garantissant ainsi la modularité et la facilité de maintenance du code.

Implementation des Classes et Justification de la strucutre

Le concept de POO a été segmenté afin que chaque classe ait une responsabilité unique. Cette structure permet de s'assurer qu'une modification des règles de l'inventaire ne nécessite pas de toucher au code de dessin de la grille.

Le controleur Global et l'Interface (main.py)

Le fichier *main.py* a été conçu pour être le contrôleur central. Sa structure combine l'initialisation des variables de Pygame avec l'orchestration des objets POO, en utilisant des fonctions globales pour gérer l'état du jeu (la grille, le joueur et l'affichage).

Bloc d'Implémentation	**Fonctions/Variables Clés**	**Justification du Rôle**
Initialisation & Configuration	`pygame.init()`, `SCREEN_WIDTH`, `TILE_SIZE`, `screen`, `clock`.	Définit l'environnement d'exécution (Pygame) et établit toutes les **constantes de dimension et de couleur** (`BLUEPRINT_BG`, `TILE_SIZE`, `PIECE_COLORS`).

Initialisation de l'État	`player_pos`, `grid`, `inventaire`, `generateur_alea`, `reset_game()`.	Le `reset_game()` est l'unique point qui crée les instances des classes POO (Inventaire, GenerateurAlea) et place les salles initiales (EntranceHall, Antichambre) dans la grille globale. Ceci garantit que toutes les parties du jeu travaillent avec des objets cohérents.
Boucle Principale	`while True`, `pygame.event.get()`.	Le moteur d'exécution. Il gère le flux entre les **événements d'entrée** (clavier/souris) et l'**état d'affichage** (`menu_active`, `game_over`), garantissant un déroulement du jeu fluide et réactif.
Fonctions `draw_...`	`draw_grid()`, `draw_inventory()`, `draw_start_screen()`, etc.	**Interface Utilisateur (UI)** : Ces fonctions dessinent les éléments à l'écran en lisant l'état des variables globales (`grid`, `inventaire.pas`) et des objets POO. Elles définissent l'aspect visuel Blueprint sans stocker la logique du jeu.
Fonctions `handle_move()`	Logique de mouvement et d'ouverture de porte.	**Point de Décision Métier** : Cette fonction implémente les règles de déplacement (vérification des murs/portes, niveau de verrouillage, et consommation de pas), déclenchant le tirage de pièce (`tirer_pieces`) ou la mise à jour de la position du joueur.

Deux fonctions principales illustrent l'intégration des algorithmes complexes dans le flux de jeu :

1. `handle_move()` : Cette fonction implémente la règle de l'énoncé selon laquelle l'ouverture d'une porte vers une nouvelle pièce consomme un pas et nécessite une vérification des ressources (`Clés` ou `Kit de Crochetage`) par rapport au `niveau_verrouillage` tiré par le `GenerateurAlea`. Elle gère également le remboursement du pas en cas d'échec (pas assez de clés ou de gemmes).
2. `place_selected_piece()` : Elle est appelée après le choix du joueur dans le menu de sélection. Elle applique la règle de la **Rotation Dynamique** pour aligner la porte de la nouvelle pièce à la grille, garantissant ainsi la connectivité du manoir. Elle gère ensuite la déduction du coût en gemmes.

Classes POO externes

L'architecture du jeu a été rigoureusement structurée pour séparer la logique métier (règles et données) de l'interface graphique (visuel). Cette séparation, bien que le code soit exécuté dans un seul fichier, garantit que les règles du jeu peuvent être modifiées sans affecter l'affichage, et inversement. Le cœur de cette structure repose sur trois entités principales : l'Inventaire, la Salle et le Générateur Aléatoire.

La Gestion de l'État du Joueur : La Classe `Inventaire`

La classe `Inventaire` est essentielle pour encapsuler l'état et les capacités du joueur. Au lieu de manipuler directement des variables globales pour les ressources, le jeu interagit avec l'inventaire via des méthodes spécifiques. Cette encapsulation permet d'appliquer les règles de consommation de manière sécurisée et centralisée. Par exemple, lorsqu'une porte de niveau 1 est rencontrée, le code de déverrouillage utilise la méthode de l'inventaire pour vérifier la présence d'une **clé** ou du **kit de crochetage**. Ce choix de conception isole la logique d'obtention et de dépense des ressources (Pas, Gemmes, Clés).

La Structure du Manoir : La Classe “Salle”

La classe “Salle” sert de modèle fondamental pour chaque pièce du manoir. Elle stocke les propriétés essentielles et statiques, telles que le nom, la couleur (qui définit souvent l'effet de la pièce), la rareté, le coût en gemmes, et la configuration de ses **portes**. L'utilisation de l'**héritage** (polymorphisme) dans ce modèle est justifiée par la création de salles spéciales comme “EntranceHall” (salle de départ) et “Antechamber” (salle de victoire). Ces classes spécialisées héritent des attributs de base tout en imposant des conditions et des effets uniques (ex: l'Antichambre déclenche la victoire).

L'Orchestration du Hasard : La Classe “GenerateurAlea”

La classe “GenerateurAlea” est dédiée à la gestion de l'imprévisibilité et des contraintes du jeu. Sa création a pour but d'isoler toute la logique mathématique complexe du reste du code. Cette classe implémente deux algorithmes cruciaux :

1. **Le Tirage Pondéré :** L'algorithme de tirage utilise une pondération rigoureuse où chaque niveau de **rareté** d'une pièce divise sa probabilité d'apparition par

trois. L'utilisation de “numpy.random.choice” garantit la performance de ce tirage et assure, par contrainte de l'énoncé, qu'au moins une pièce sans coût en gemmes est toujours proposée pour éviter le blocage du jeu.

2. **Le Verrouillage des Portes :** Elle est responsable de tirer aléatoirement le **niveau de verrouillage** des nouvelles portes (Niveau 0, 1 ou 2), en respectant la règle selon laquelle la probabilité de niveaux élevés augmente à mesure que le joueur avance vers le haut de la grille.

Le Déverrouillage : La Classe “Porte”

Bien que la grille ne stocke pas directement des instances de `Porte` pour chaque liaison, la classe `Porte` existe pour encapsuler la **règle complexe d'ouverture**. Elle contient la logique conditionnelle suivante : si la porte est de Niveau 1, elle peut être ouverte soit par une clé, soit par le kit de crochetage ; si elle est de Niveau 2, seule une clé permet l'ouverture. Cette classe est cruciale car elle permet d'isoler le contrôle des ressources du joueur.

Algorithmes Cruciaux Intégrés

Le succès du jeu repose sur l'intégration de deux algorithmes complexes qui gèrent l'équilibrage et la cohérence de l'environnement : le Tirage Pondéré et la Rotation Dynamique.

L'algorithme de tirage des pièces est central pour l'équilibrage. Il utilise le module `numpy.random.choice` pour effectuer un tirage pondéré, garantissant ainsi que la probabilité d'obtenir une pièce est inversement proportionnelle à son niveau de **rareté** (probabilité proportionnelle à 1/3^rareté). Cette implémentation algorithmique assure que le défi croît : les pièces rares et avantageuses sont statistiquement moins accessibles. De plus, pour respecter l'une des contraintes fondamentales du jeu, l'algorithme vérifie toujours que la sélection proposée au joueur contient **au moins une pièce gratuite** (coût de **0 gemme**), empêchant ainsi le joueur d'être complètement bloqué par manque de ressources.
L'algorithme de **Rotation Dynamique** est vital pour l'intégrité structurelle du manoir. Lorsqu'une nouvelle pièce est sélectionnée par le joueur, cet algorithme calcule l'angle de rotation précis nécessaire pour que la porte par défaut de la nouvelle pièce s'aligne et se connecte parfaitement à la porte de la pièce précédente. Ce processus s'appuie sur la classification des portes (N, S, E, O) et

assure que le joueur ne se retrouve jamais face à un mur dans une zone où une connexion est attendue. La rotation dynamique garantit ainsi la **cohérence du gameplay** et le respect des contraintes spatiales du manoir.

Conclusion

La structure modulaire adoptée, bien que centralisée dans le fichier `main.py`, a permis d'isoler efficacement la **logique métier** (gérée par des classes dédiées comme `Inventaire`, `Salle`, et `GenerateurAlea`) de l'**interface graphique** (gérée par les fonctions de dessin Pygame). Cette séparation garantit que le code est propre, lisible et hautement maintenable.
L'intelligence du jeu réside dans l'intégration réussie d'algorithmes complexes :
1. **L'Équilibrage Statistique :** Le **tirage pondéré** via NumPy a permis d'appliquer la contrainte de rareté avec précision, assurant un défi progressif.
2. **La Cohérence :** La **Rotation Dynamique** des pièces a résolu le problème crucial de la connexion des salles, garantissant l'intégrité structurelle du manoir à chaque nouvelle découverte.