Rapport projet de Programmation

Sabrina Moulahcene

21204151

Elif Kaya

21213105

M1 Syscom 2025/2026

Introduction

Ce projet consiste à implémenter en Python une version simplifiée du jeu *Blue Prince*, un jeu de construction procédurale de manoir dans lequel le joueur doit progresser pièce par pièce jusqu'à atteindre l'antichambre située en haut d'une grille de dimension 5×9. Le manoir n'est pas généré à l'avance : ses pièces apparaissent dynamiquement lorsque le joueur ouvre une porte, et il doit alors choisir parmi trois salles tirées aléatoirement celle qu'il souhaite ajouter à la structure du bâtiment. Chaque pièce possède un coût en gemmes, des portes dans certaines directions, un niveau de verrouillage, des objets potentiels et parfois un effet particulier.

L'objectif principal de ce projet est de mettre en pratique les principes de la programmation orientée objet afin de structurer clairement les différentes entités du jeu : pièces, portes, joueur, inventaire, manoir et interface graphique. Le programme intègre également une part importante d'aléatoire, afin que chaque partie soit unique tout en respectant les contraintes imposées par l'énoncé (rareté des pièces, niveaux de verrouillage dépendant de la position, tirage garantissant au moins une pièce gratuite, etc.).

Afin de rendre le jeu accessible et visuellement clair, nous avons utilisé la bibliothèque `Pygame` pour représenter la grille du manoir, gérer les interactions clavier et afficher les différents menus de sélection. Le résultat final permet au joueur de se déplacer, de gérer ses ressources, de découvrir de nouvelles salles et de tenter d'atteindre l'objectif tout en évitant d'épuiser ses pas ou de se retrouver bloqué.

1 Structure générale du programme

Le projet est organisé en plusieurs fichiers Python, chacun ayant une responsabilité clairement définie. Ce découpage modulaire permet de séparer la logique métier, la gestion du manoir, l'interface graphique et les paramètres globaux.

1.1 `main.py` – Point d'entrée du programme

Le fichier `main.py` :
— instancie un objet `Game`,
— lance la boucle principale du jeu.
Cette structure garantit que la logique centrale est déléguée aux autres modules.

1.2 `game.py` – Gestion du déroulement du jeu

Ce fichier contient la classe `Game`, responsable de :
— la boucle principale,
— l'affichage via Pygame,
— la gestion des entrées clavier (ZQSD + `ESPACE`),
— la direction sélectionnée,
— les conditions de victoire et de défaite,
— l'appel aux méthodes de la classe `Maison` pour les déplacements,
— l'interface de sélection de salle (`RoomPicker`) lorsque le joueur ouvre une nouvelle porte.

1.3 `maison.py` – Modélisation du manoir

Ce module définit la classe `Maison`, qui gère :
— la grille du manoir (dimensions 5×9),
— l'initialisation des salles *Start* et *Goal*,
— la logique de déplacement (`can_move`, `move`),

— l'ouverture des portes et la création de nouvelles salles,
— l'affichage de la grille complète.

1.4 `models.py` – Classes métiers principales

Ce fichier regroupe les entités fondamentales du jeu :
— **Room** : salles du manoir (couleur, coût, rareté, loot, boutique, portes),
— **Door** : direction + niveau de verrouillage,
— **Inventory** : gestion de toutes les ressources du joueur,
— **Player** : position et déplacement du joueur dans la grille.

1.5 `rooms_catalog.py` – Catalogue des pièces et tirages aléatoires

Ce module contient :
— la liste des salles disponibles (catalogue),
— le filtrage des salles compatibles avec la direction d'entrée,
— l'élimination des salles ayant des portes sortant du manoir,
— la gestion de la rareté (pondération des tirages),
— la génération des niveaux de verrouillage en fonction de la hauteur dans le manoir,
— la garantie d'obtenir au moins une salle gratuite parmi les trois proposées.

1.6 `room_picker.py` – Interface de sélection des salles

Cette classe gère l'écran de choix des salles :
— affichage des trois salles tirées,
— navigation avec les flèches gauche/droite,
— validation du choix avec `Entrée`,
— relance du tirage si le joueur dispose d'un dé (*reroll*),
— paiement automatique des gemmes.

1.7 `settings.py` – Paramètres globaux

Fichier centralisant les constantes :
— dimensions de la fenêtre,
— taille des tuiles,
— palette de couleurs,
— variables globales du jeu.

Bilan

Ce découpage permet une séparation nette entre :
— **logique de jeu** : `Maison`, `Room`, `Door`, `Inventory`, `rooms_catalog`,
— **interface graphique** : `Game`, `RoomPicker`, Pygame,
— **paramètres globaux** : `settings`.
Cette organisation rend le code plus clair, modulaire, extensible et facile à maintenir.

2 Modélisation orientée objet

2.1 Représentation des entités du jeu

L'ensemble des entités principales du jeu ont été implémentées sous forme de classes dédiées, chacune remplissant une responsabilité bien définie.

Player (joueur) La classe `Player` stocke :
— la position actuelle du joueur (`row`, `col`),
— une méthode `move(dr, dc)` permettant de modifier cette position.

Inventory (inventaire) La classe `Inventory` regroupe en un seul endroit toutes les ressources du joueur :
— **consommables** : `steps`, `keys`, `gems`, `coins`, `dice`,
— **permanents** : `lockpick`, `metal_detector`, `rabbit_foot`.
Ce regroupement simplifie :
— les vérifications d'ouverture de porte,
— les achats en boutique,
— le traitement des objets trouvés dans les salles.

Room (salle) Une salle est représentée par la classe `Room`, contenant :
— un nom,
— une couleur,
— un coût en gemmes,
— une rareté,
— une liste de portes,
— un effet d'entrée `on_enter`,
— une gestion particulière pour les salles de type *Shop*.
Elle possède également une méthode `draw(screen, x, y)` pour son affichage sur la grille.

Door (porte) La classe `Door` encapsule :
— une direction,
— un niveau de verrouillage (`UNLOCKED`, `LOCKED`, `DOUBLE_LOCKED`),
— la logique d'ouverture via les méthodes `can_open` et `open`.
Ce choix évite de répéter la logique dans `Game` ou `Maison`.

Maison (grille du manoir) La classe `Maison` gère :
— la grille complète des salles,
— les conditions de déplacement,
— l'ouverture effective des portes,
— la détection des nouvelles salles,
— l'affichage de la carte.

Conclusion : chaque classe possède une responsabilité unique, respectant le principe **S (Single Responsibility)** du modèle SOLID.

2.2 Types pour les directions et les verrouillages

Les types `Direction` et `LockState`, définis dans le dossier `utils`, remplacent les valeurs magiques dans le code. Ils permettent :
— d'éviter l'usage d'entiers arbitraires,
— d'empêcher les comparaisons ambiguës,
— de réduire le risque d'erreurs de manipulation.
Ils améliorent ainsi fortement la lisibilité et la robustesse des conditions de déplacement et d'ouverture dans la classe `Maison`.

3 Gestion du manoir et des déplacements

La classe `Maison` utilise une grille 2D de dimensions `MAP_ROWS` × `MAP_COLS`. Chaque case de cette grille contient soit :

— une instance de `Room`,
— soit `None` si la salle n'a pas encore été générée.

3.1 Salles de départ et d'arrivée

Deux salles sont placées dès l'initialisation :

— **Start** (salle de départ) : positionnée en bas au centre de la grille, avec une porte ouverte vers le haut.
— **Goal** (Antichamber) : située tout en haut au centre, avec une porte *doublement verrouillée* vers le bas.

Ces emplacements respectent rigoureusement les contraintes imposées par l'énoncé.

3.2 Méthodes fondamentales

La classe `Maison` implémente trois méthodes essentielles à la logique du jeu.

`can_move()` Cette méthode vérifie successivement :
— si la position cible appartient à la grille,
— si la salle courante contient une porte dans cette direction,
— si cette porte peut être ouverte via `Door.can_open()` en utilisant l'inventaire.

`move()` Cette méthode réalise le déplacement effectif :
— Si la salle cible existe déjà :
  — la porte est ouverte,
  — le joueur est déplacé,
  — un pas est consommé.
— Si la salle cible n'existe pas encore :
  — la méthode renvoie `("NEW_ROOM", r2, c2)`,
  — ce qui déclenche l'appel au `RoomPicker` afin de générer une nouvelle salle.

3.3 Intégration dans la classe `Game`

Dans `Game`, la direction est sélectionnée à l'aide des touches `ZQSD`. La validation du mouvement s'effectue avec la touche `ESPACE`.

Le déplacement est ensuite réalisé via :

`try_move()` → `Maison.move()`

Conclusion : Le découplage est net :
— `Game` gère l'**interface** et les entrées utilisateur,
— `Maison` gère les **règles internes du manoir** et la cohérence des déplacements.

4 Gestion de l'inventaire et des objets

L'inventaire est centralisé dans la classe `Inventory`. Il regroupe l'ensemble des ressources du joueur :

— **consommables** : `steps`, `keys`, `gems`, `dice`, `coins`,
— **permanents** : `lockpick`, `metal_detector`, `rabbit_foot`.

Ce choix de conception simplifie considérablement toutes les interactions du jeu, car les salles, les portes et le `RoomPicker` n'ont besoin que d'un seul objet pour gérer les ressources.

4.1 Système de loot dans `Room.on_enter()`

L'implémentation respecte fidèlement les règles de l'énoncé :
— la probabilité de loot dépend de la **couleur** de la salle,
— un **bonus** est appliqué si le joueur possède la patte de lapin,
— le détecteur de métaux augmente la probabilité d'obtenir gemmes et clés,
— les objets possibles incluent : gemmes, clés, dés, nourriture, pièces, et objets permanents,
— les valeurs de nourriture varient selon l'objet (pomme, banane, gâteau, sandwich, repas complet).

Tout est encapsulé dans la classe `Room`, ce qui rend le système extensible et facile à maintenir.

4.2 Salles de type Shop

Une fonctionnalité supplémentaire a été ajoutée : les salles de type `Shop`. Elles permettent :
— d'échanger **3 pièces** contre **1 clé**,
— d'échanger **2 pièces** contre **10 pas**,
— des achats automatiques répétés jusqu'à épuisement des pièces.

La gestion est entièrement intégrée à `Room.on_enter()`, tout en restant indépendante du reste du code. L'implémentation est propre et conforme à l'esprit du jeu.

5 Tirage des salles (`rooms_catalog`)

Le tirage des pièces respecte l'ensemble des contraintes imposées par l'énoncé.

5.1 Filtrage par direction

Seules les salles possédant une porte correspondant à la direction d'entrée sont retenues. Cela garantit la cohérence structurelle du manoir.

5.2 Filtrage par bords de la grille

Toutes les salles dont une porte conduirait en dehors du manoir sont éliminées. Cette étape empêche de générer des portes invalides.

5.3 Gestion de la rareté

La probabilité de tirage d'une salle de rareté r est proportionnelle à :

$$\frac{1}{3^r}.$$

Ainsi, les salles rares apparaissent beaucoup moins fréquemment, conformément aux règles du jeu.

5.4 Niveaux de verrouillage dépendants de la hauteur

Les niveaux de verrouillage augmentent progressivement en montant dans la grille :
— en bas : portes généralement ouvertes,
— au centre : mélange équilibré,
— en haut : portes systématiquement **doublement verrouillées**.

Ce système renforce la difficulté à mesure que le joueur se rapproche de l'Antichamber.

5.5 Garantie d'au moins une salle gratuite

Lors du tirage des trois salles :
— si aucune pièce ne coûte 0 gemme,
— alors un **nouveau tirage** est automatiquement effectué.
Cela évite au joueur de se retrouver bloqué par manque de gemmes et assure la jouabilité.

6 Interface graphique et interactions

L'interface du jeu repose entièrement sur la bibliothèque `Pygame`, permettant une représentation visuelle simple mais claire de la progression du joueur dans le manoir.

6.1 Affichage géré dans `Game`

La classe `Game` prend en charge :
— la création de la fenêtre principale,
— l'affichage de la grille du manoir,
— la représentation du joueur (un cercle rouge),
— l'affichage de l'inventaire sur la partie inférieure de l'écran,
— l'indication visuelle de la direction sélectionnée via une flèche.
Cette structure permet de séparer clairement l'affichage de la logique interne du manoir.

6.2 Contrôles utilisateur

Les interactions du joueur sont gérées par un ensemble de touches simples :
— **Z, Q, S, D** : sélection de la direction (haut, gauche, bas, droite),
— **ESPACE** : validation du déplacement dans la direction sélectionnée,
— **Flèche gauche / droite** : navigation entre les trois salles proposées lors du tirage,
— **R** : relance du tirage si le joueur possède un dé,
— **Entrée** : validation du choix de salle.
Ces contrôles respectent les conventions habituelles du clavier et rendent la navigation intuitive.

6.3 Composant `RoomPicker`

Le `RoomPicker` constitue l'écran de sélection des nouvelles salles. Il affiche les trois salles tirées sous forme de "cartes" avec :
— leur nom,
— leur coût en gemmes,
— leur niveau de rareté,
— des bordures et effets visuels permettant de mettre en valeur la salle sélectionnée.
Cette interface permet au joueur de faire un choix clair et informé lors de l'expansion du manoir.

7 Conclusion

La conception du projet repose sur une architecture claire et modulaire permettant une séparation nette entre les différentes responsabilités du programme. D'une part, la logique interne du jeu (classes `Maison`, `Room`, `Door`, etc.) est isolée de l'interface graphique gérée par la classe `Game`. D'autre part, la gestion de l'aléatoire suit fidèlement les contraintes imposées par l'énoncé, notamment en ce qui concerne la rareté des salles, la compatibilité des portes et la garantie d'obtenir toujours au moins une salle gratuite.

La modélisation orientée objet adoptée est à la fois propre, intuitive et extensible : l'ajout de nouveaux types de salles, de mécaniques supplémentaires ou d'objets spéciaux peut se faire sans modifier le cœur du programme.

Au final, l'ensemble du projet propose une expérience de jeu complète et fonctionnelle, intégrant :

— les déplacements du joueur,
— le tirage et la sélection de nouvelles salles,
— la gestion détaillée des ressources,
— le système de loot et les objets spéciaux,
— les magasins et conversions (clé, pas, etc.),
— la rareté des pièces et les niveaux de verrouillage,
— les conditions de victoire et de défaite.

Ce travail démontre l'efficacité d'une structure logicielle bien pensée et l'importance d'un découpage cohérent pour garantir lisibilité, maintenabilité et évolutivité du code.