ICHOU Aymane, N° 22423539
LEMAIRE Théo, N° 22419719

Université de Paris
M1 Informatique
Année 2024/2025
Rapport de Projet - Piece Out

1. Introduction

Ce rapport présente notre projet de C++ intitulé *Piece Out*.
Durant ce projet, nous avons pu manipulés les concepts suivants :

- Gestion des constructeurs, destructeurs, et copie.
- Utilisation de l'héritage et de la généricité.
- Manipulation des exceptions et des énumérations.
- Mise en œuvre de patrons de conception (Decorator, Visitor, etc.).
  Ce projet s'appuie sur la bibliothèque SFML pour le rendu graphique et propose une séparation claire entre le modèle et la vue.

2. Présentation Générale du Jeu

2.1. Description

Le jeu *Piece Out* est un casse-tête où le joueur manipule des pièces ressemblant à celles de Tetris.
L'objectif est d'atteindre une configuration cible en utilisant :

- **Rotations** : orientées dans le sens horaire ou anti-horaire.
- **Déplacements** : limités à certaines directions.
- **Symétries** : par rapport à un axe vertical ou horizontal.

Nous avons pu intégrer l'ensemble de ces fonctionnalités dans notre projet.

2.2. Bibliothèques utilisées

Nous avons utilisé la bibliothèque **SFML** pour le rendu graphique. Cette bibliothèque facilite la gestion des fenêtres, des événements utilisateur, et des éléments graphiques.
Nous avons utilisé la bibliothèque **Iostream** qui permet d'utiliser les flux d'entrée et de sortie comme std::cout, std::cin, et std::endl.
Nous avons utilisé la bibliothèque **Vector** qui permet l'utilisation de la structure vector.

3. Conception et Architecture

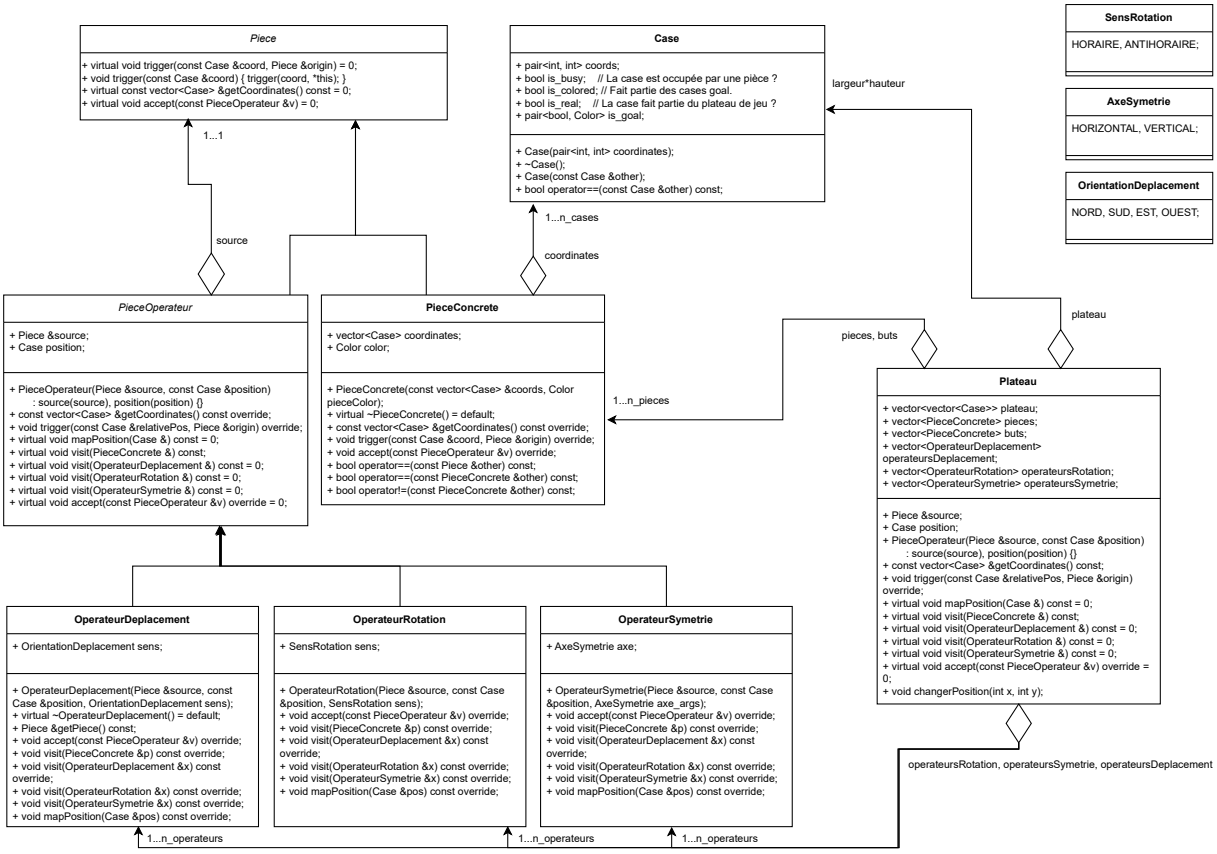
3.1. Structure Générale

Notre projet suit une architecture basée sur la séparation **modèle/vue** :

- Le modèle gère les règles du jeu, les mouvements, et la détection de victoire.
- La vue s’occupe du rendu graphique et des interactions utilisateur.

3.2. Diagrammes UML

Diagramme de Classes



3.3. Détails des Classes Principales

- **`PieceConcrete`** : Représente une pièce du jeu, composée de plusieurs cases. Elle gère les transformations possibles (rotation, symétrie, déplacement).
- **`Plateau`** : Gère l’espace de jeu, les pièces, et leur interaction. Il vérifie les conditions de victoire.
- **Décorations et Opérateurs** : Implémentés à l’aide des patrons **Decorator** et **Visitor**, ces classes gèrent les mouvements et les transformations des pièces.

4. Développement

4.1. Méthodologie de travail en binôme

- Pair programming, sur une même machine.
- Réunion régulière sur des supports distants ou bien dans des lieux physiques.
- Gestion des versions avec Git pour un suivi clair des modifications.

4.2. Problèmes Rencontrés et Solutions

- **Problème** : Inversion des coordonnées lors des clics souris.
  Solution : Alignement des coordonnées entre SFML et le modèle en ajustant la gestion des offsets.
- **Problème** : Détection des collisions entre pièces.
  Solution : Ajout d'une fonction de vérification dans la classe `Plateau`.
- **Problème** : Symétrie, parfois effectué plusieurs fois.
  Solution : Ajout d'un set, recueillant les opérateurs déjà traités.
- **Problème** : L'application d'une rotation pouvait permettre de faire sortir la pièce hors du plateau.
  Solution : Ajout de vérification et de tests supplémentaires.

5. Résultats

5.1. Fonctionnalités Implémentées

- Gestion complète des mouvements (rotations, translations, symétries).
- Gestion des collisions, et de l'espace de jeu.
- Ajout d'un menu.
- Intégrations de plusieurs niveaux.
- Détection des conditions de victoire.
- Interface graphique fonctionnelle avec SFML.

6. Prise en Main du Projet

6.1. Compilation

Pour compiler le projet :

```
make pieceOut
```

6.2. Exécution

Pour lancer le programme :

```
./pieceOut
```

7. Détails des fonctions.

```
Plateau(int rows, int cols, vector<vector<int>> calque,
vector<PieceConcrete> pieces, vector<PieceConcrete> buts,
vector<OperateurDeplacement> operateursDeplacement,
vector<OperateurRotation> operateursRotation, vector<OperateurSymetrie>
operateursSymetrie);
```

Construction du plateau, et création de l'espace de jeu.

```
Color getColorOfCase(int x, int y);
```

Fonction qui renvoie la couleur d'une case

```
void drawGrid(RenderWindow &window, int caseSize, int borderSize);
```

Fonction de dessin qui va venir dessiner l'ensemble du plateau, les pièces, les opérateurs, la surbrillance d'une pièce.

```
void displayBusyCases() const;
```

Fonction de débuggage affichant textuellement les cases occupés.

```
bool isMovePossible(int pieceIndex, OrientationDeplacement direction);
```

Cette fonction possède le même rôle que isRotationPossible et isSymetriePossible, à quelques différences près.
Ces fonctions vont effectuer un mouvement donné sur une pièce donnée de manière abstraite, et renvoyer si le mouvement est possible ou non. Elles vérifient la collision, la limite de bordure et l'existence des cases résultantes après mouvement, ainsi que l'existence de la pièce.

```
void movePiece(int pieceIndex, OrientationDeplacement sens);
```

Cette fonction possède le même rôle que rotatePiece et symetriePiece, à quelques différences près. Elles vont appeler les fonctions isPossible, puis vérifier si un opérateur existe en fonction de la pièce donnée et le sens donné grâce à `isOperateur…..Valid.`

Elles libèrent ensuite les cases, puis appelle la fonction `ApplyTransformationToOperateur`

Enfin, on vient trigger la pièce cible pour appliquer le mouvement.

```
OperateurDeplacement *isOperateurDeplacementValid(int pieceIndex,
OrientationDeplacement direction);
```

Ces fonctions isOperateur….Valid renvoie une référence d'un opérateur si il existe, sinon un pointeur null. Cela permet de pouvoir tester l'existence d'un opérateur dans notre tableau d'opérateur.

```
void ApplyTransformationToOperateur(PieceConcrete &concretePiece,
PieceOperateur *operateur_a_appliquer);
```

Fonction générique, qui prend en paramètre n'importe quel type d'opérateur, et applique la translation/rotation/symetrie présent sur une pièce donnée.

```
bool isGameFinished();
bool IsPieceInGoal(PieceConcrete p1, PieceConcrete p2);
```

Fonctions complémentaires pour tester le gain. IsGameFinished vérifie chacune des pièces en vérifiant que les coordonnées des pièces correspondent aux coordonnées gagnantes grâce à IsPieceInGoal.

8. Conclusion et Perspectives

Le projet *Piece Out* nous a permis d’explorer et d’appliquer les concepts avancés en C++ tout en créant un jeu complet et fonctionnel.

Améliorations envisagés, mais non réalisées.

- Effet visuel indiquant la victoire.
- Ajout d’une touche ou un bouton pour revenir sur un coup joué.