Documentation du projet IN204

Création d'un jeu Tétris

Réalisé par :

Cyriac SALIGNAT
cyriac.salignat@ensta.fr

Théotime GALMICHE
theotime.galmiche@ensta.fr

Encadrants :

Bruno MONSUEZ

Valentin PERELLE

Année universitaire 2025–2026

Table des matières

1 Introduction

Cette documentation fait suite à la réalisation du jeu Tétris dans le cadre du cours IN204, et intervient après l'analyse faite en amont du projet.
Le but de ce document est de fournir les instructions à suivre pour pouvoir compiler et exécuter le projet, et de détailler les usages des concepts objets.

2 Instructions à suivre

Les instructions seront données pour des ordinateurs tournant sur Ubuntu/Debian.

2.1 Installations des dépendances

```
# Mettre à jour les paquets
    sudo apt update

# Installer les outils de compilation
    sudo apt install build-essential cmake git

# Installer SFML
    sudo apt install libsfml-dev

# Vérifier l'installation
    cmake --version
    g++ --version
```

2.2 Récupération du projet

```
git clone https://github.com/Cyriac20/IN204.git

cd IN204/
```

La structure du dossier devrait alors être la suivante :

```
IN204/
| src/
|   |----main.cpp
|   |---- grille.cpp
|   |---- grille.hpp
|   |---- pieces.cpp
|   |---- pieces.hpp
```

```
|   |---- score.cpp
|   |---- score.hpp
|   |---- menu.cpp
|   |---- menu.hpp
|   |---- horloge.cpp
|   |---- horloge.hpp
|   |---- res/
|---- build/    # Optionnel
|---- CMakeLists.txt
|---- README.md
```

2.3 Création du dossier build et configuration avec CMake

```
# Si un dossier ../IN204/build/ existe déjà
    rm -rf build/

# Puis dans tous les cas
    mkdir build/
    cd build/

    cmake ..
    cmake --build .
```

2.4 Exécution du jeu

```
# Toujours dans le fichier /build/
    ./bin/main

# Le jeu devrait alors se lancer
```

3 Fonctionnalités

Le projet implémente un jeu de **Tetris** avec les fonctionnalités suivantes :

- **Menu principal** : Permet de naviguer entre les options *Jouer*, *Commandes* et *Quitter*.
- **Affichage graphique de la grille** : La grille est représentée à l'écran avec ses cases et ses lignes.
- **Gestion des pièces** : Les pièces (I, O, T, L, J, S, Z) sont affichées, peuvent se déplacer à gauche/droite, descendre, et tourner.
- **Affichage de la prochaine pièce** : Une zone au-dessus de la grille montre la pièce qui apparaîtra après la chute de la pièce actuelle.

— **Score et niveau** : Affichage du score actuel, du niveau et d'un chronomètre en temps réel.
— **Écran Game Over** : Affiche un message de fin de partie lorsque la grille est remplie et que le joueur a perdu.

4 Utilisation des notions vues en cours

Parmi les concepts objets vus en cours, voici ceux que nous avons utilisé, ainsi qu'un exemple de leur utilisation systématiquement :

— **Des classes, des constructeurs, des destructeurs, des contrôles d'accès** : en se basant sur pieces.hpp, nous avons bien des classes, constructeurs (PieceI(), PieceO() ...), des destructeurs (~Piece() = default) ainsi que des contrôles d'accès (public et protected ; nous avons également du private dans horloge.hpp par exemple).
— **Hiérarchies de classes** : la classe mère Piece ne sert qu'à définir les bases pour les différentes classes qui en héritent (PieceI, PieceO ...).
— **Conteneurs STL** : nous utilisons des conteneurs pour la définition des des positions des pièces par exemple (std : :array<std : :array<int,2>,4> position ;).
— **Polymorphisme** : ce concept est utilisé sur la rotation des pièces, dont l'existence est commune à toutes les pièces, mais les spécificités propres à chaque pièce :

```
// Méthode virtuelle pure dans Piece
    virtual void rotation(grille& matrice) = 0;

// Implémentations différentes dans chaque classe fille
    void PieceI::rotation(grille& matrice) override { ... }
```

— **Pointeurs intelligents** : nous en utilisons dans le main :

```
std::unique_ptr<Piece> piece_aleatoire();
std::unique_ptr<Piece> piece = piece_aleatoire();
```

— **Itérateurs** : dans pieces.cpp, nous avons :

```
for (std::array<int,2> coord : position){  // Range-based for = itér
    matrice.set(coord[0], coord[1], id);
}
```

— **Contrats** : dans pieces.cpp, nous avons un contrat sur le taille des blocs :

```
void Piece::apparition(grille& matrice){
    assert(position.size() == 4);  // Contrat: toujours 4 blocs
    for (std::array<int,2> coord : position){
       matrice.set(coord[0], coord[1], id);
    }
}
```

5 Difficultées rencontrées et choix faits

Une première difficultée a été l'affichage de la grille. Nous avions initialement réalisé une boucle avec un draw pour chaque case, ce qui augmentait fortement le temps d'exécution. Il a ainsi fallu changer de méthode pour gagner en efficacité d'affichage.

Un autre élément a été la définition de la rotation, qui fait appel à un pivot et permet ainsi d'avoir une fonction relativement similaire selon les types de pièces.

Ensuite, un grand enjeu a résidé dans la création d'un affichage clair et non surchargé (et esthétique). La manipulation des affichages, dont nous n'étions pas expérimenté précédemment, a nécessité un temps d'apprentissage certain.

Enfin, la gestion des resets a pendant un temps posé problème puisque le "Game Over" n'effectuait pas un reset profond de toute la grille et des paramètres, ce qui faisait crasher le jeu systématiquement lorsque nous voulions relancer une partie.

La structure actuelle du code est fortement centrée sur un seul joueur, avec la gestion de la grille, des pièces et des entrées clavier imbriquées. Cette architecture ne permettait pas une séparation claire des instances de jeu ni une synchronisation facile entre plusieurs joueurs. En conséquence, l'implémentation d'un mode multijoueur aurait nécessité une réécriture importante de l'architecture. C'est pourquoi nous avons opté pour l'implémentation d'autres fonctionnalités (comme l'affichage des prochaines pièces) et sur le visuel du jeu dans le temps qui nous restait lors du développement.