Luiz Henrique Gariglio dos Santos
Maria Clara Oliveira Domingos Ruas

PROJET INFORMATIQUE
TETRIS

PALAISEAU
2026

TABLE DES MATIÈRES

1 TETRIS

Ce travail vise à présenter la structure du projet développé pour la discipline de Programmation Orientée Objet. Le projet consiste en un jeu Tetris avec les options de jeu solo, multijoueur local et multijoueur en ligne pour jusqu'à deux joueurs.

Le jeu a été développé en suivant les directives de la programmation orientée objet. Ainsi, différentes classes ont été définies, correspondant aux parties constituantes du programme. Dans la Figure A, nous pouvons voir les classes utilisées et les relations entre elles.

Le jeu dispose d'une bande-son et de sons pour la rotation de la pièce et le nettoyage de ligne. Chaque pièce, en plus de sa forme, possède une couleur différente pour une meilleure identification. Toute l'interface graphique a été développée en utilisant la bibliothèque *raylib*.

Le multijoueur en ligne a été développé en utilisant l'architecture client-serveur. Nous avons utilisé un serveur en ligne (Firebase) qui sert de base de données, où les informations des parties sont conservées. Chaque instance du jeu en ligne agit comme client, envoyant les données de la partie vers la base de données de manière systématique et lisant les informations de la partie de l'autre joueur pour une mise à jour locale. Le code de fonctionnement du client est basé sur la bibliothèque *winhttp*, et les messages échangés avec le serveur sont au format *JSON*.

Pour le bon fonctionnement du multijoueur en ligne, l'un des joueurs doit sélectionner la touche *O*, tandis que l'autre doit appuyer sur *Shift+O*.

Lien vers github: Tetris

Commande de compilation:

```
g++ main.cpp grid.cpp block.cpp blocks.cpp colors.cpp position.cpp
game.cpp client.cpp -I ./nlohmann/ -L lib/ -lraylib -lwinhttp
-lgdi32 -lwinmm -lws2_32 -o tetris.exe

./tetris.exe
```

2 CLASSES

2.1 Game

La classe *Game* contient les méthodes et paramètres nécessaires pour assurer le fonctionnement du jeu, tels que *score*, *level* et *game over*. Parmi les méthodes figurent celles pour mettre à jour le score, choisir, déplacer et faire pivoter les blocs, ainsi que pour contrôler les effets sonores et la musique du jeu. La classe possède également un objet *grid*, qui est l'espace où se déroule le jeu. En résumé, une instance de la classe *Game* est créée pour chaque partie.

2.2 Grid

La classe *Grid* est le plateau de jeu. Elle consiste en une matrice $20x10$, initialement remplie uniquement de zéros. Chaque bloc est représenté par un nombre à une position de la matrice. Chaque type de bloc possède un numéro spécifique, également lié à sa couleur. Ici se trouvent aussi les fonctions pour nettoyer et déplacer les lignes, ainsi que pour obtenir le niveau du joueur.

2.3 Block

La classe Block représente les pièces du Tetris. Elle contrôle leurs rotations et mouvements. Les classes IBlock, JBlock, LBlock, OBlock, SBlock, TBlock et ZBlock sont des spécifications de la classe Block pour chaque type différent. Elles contiennent les informations de forme et de couleur, en plus des caractéristiques héritées de la classe Block.

2.4 Position

Cette classe ne contient que les informations de ligne et de colonne de chaque bloc.

2.5 Client

La classe Client est responsable de contrôler la communication en ligne. Elle contient principalement les méthodes pour échanger des messages avec le serveur.

3 MÉTHODE

C'est dans le fichier main.cpp que le jeu est réellement implémenté. Dans la boucle principale, chaque état est défini (MENU, SINGLEPLAYER, MULTILOCAL, MULTIONLINE et GAMEOVER). Le jeu commence par un écran de menu(1) où il est possible de sélectionner le type de jeu souhaité à partir du clavier, avec les options S pour solo (2), L pour multijoueur local (3) et O pour multijoueur en ligne (5). Dans le multijoueur local, l'autre joueur utilise le WASD. En cas de défaite, le joueur est redirigé vers l'écran de game over, et pour revenir au menu, il suffit d'appuyer sur n'importe quelle touche. Si le multijoueur en ligne est sélectionné, un écran de chargement (4) apparaît jusqu'à ce que l'autre joueur se connecte.

Figura 1 – Menu

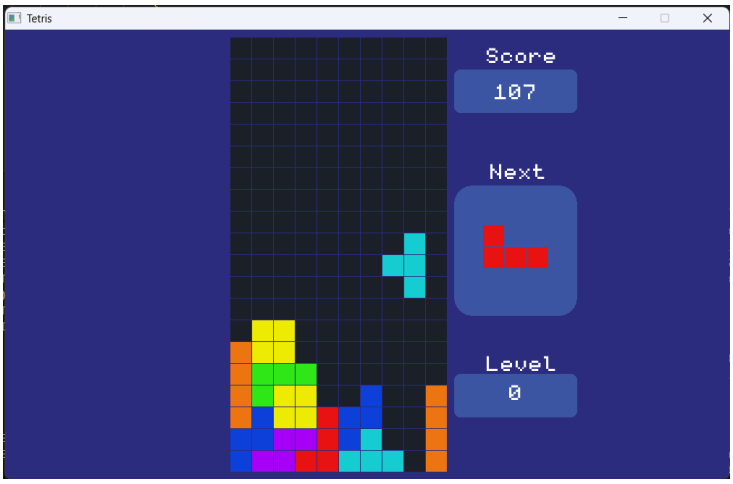

Figura 2 – Singleplayer

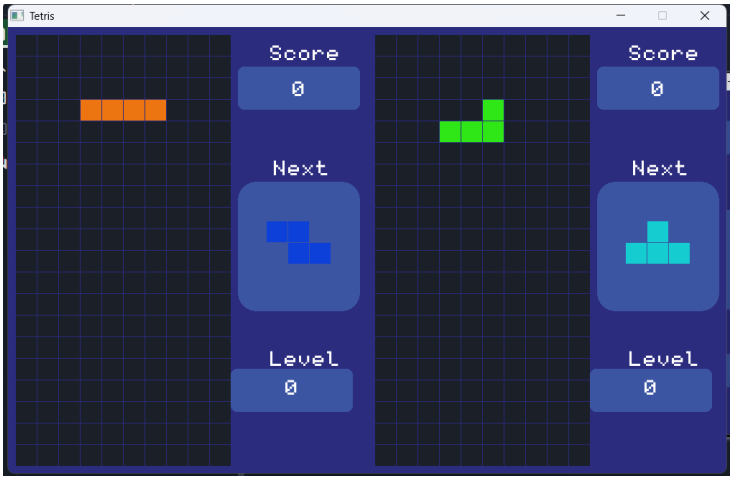

Figura 3 – Multiplayer local

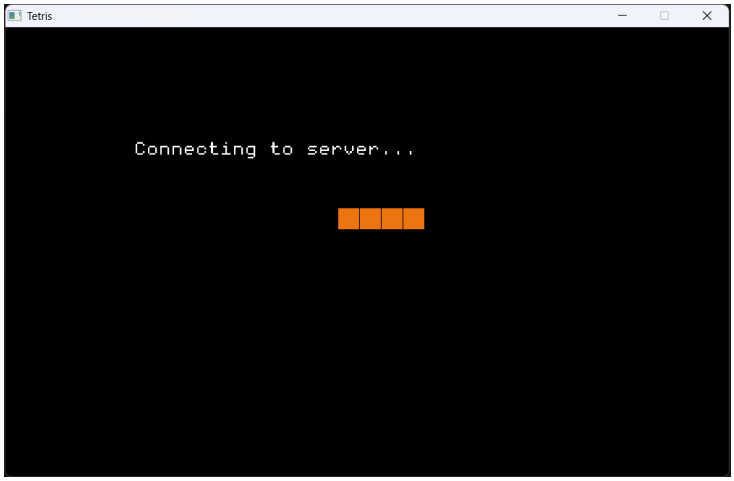

Figura 4 – Loading

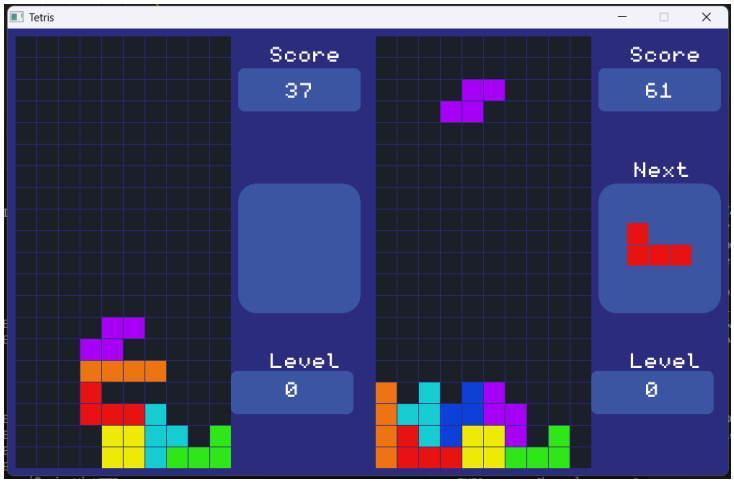

Figura 5 – Multiplayer en ligne

4 ANALYSE DE COMPLEXITÉ

L'analyse de complexité du jeu considère que toute son exécution se déroule à l'intérieur d'une boucle principale while(WindowShouldClose() == false), responsable du maintien du jeu en fonctionnement tant que la fenêtre reste ouverte. Cette boucle représente le cycle de mise à jour continue du jeu, au cours duquel sont effectuées des opérations telles que la gestion des événements, la mise à jour de l'état des pièces et le rendu graphique.

Étant donné que tous les autres paramètres du jeu, tels que la taille du plateau, le nombre maximal de pièces, les structures de données utilisées et les opérations exécutées à chaque itération, sont fixes dans le temps, le coût computationnel de chaque itération de la boucle principale demeure constant. Ainsi, chaque cycle du while exécute un ensemble limité et prévisible d'opérations.

Du point de vue de la complexité temporelle, chaque itération de la boucle possède une complexité O(1), puisqu'elle ne dépend pas de variables qui évoluent dynamiquement au cours de l'exécution. Par conséquent, le temps total d'exécution du programme est proportionnel uniquement à la durée pendant laquelle le joueur maintient la fenêtre ouverte, et non à la croissance des données ou des structures internes.

En ce qui concerne la complexité spatiale, comme toutes les structures utilisées ont une taille fixe et préalablement définie, la consommation de mémoire reste également constante, caractérisant une complexité en espace O(1). Cette approche garantit des performances stables et prévisibles, ce qui est particulièrement adapté aux applications en temps réel telles que les jeux, où l'efficacité et la réactivité sont essentielles.

5 STRATÉGIES DE ROBUSTESSE

Dans la construction des classes, tous les attributs sont privés afin d'assurer un meilleur contrôle et une plus grande sécurité des informations accessibles dans le code. Les informations pertinentes sont accessibles au moyen de fonctions get, qui retournent les valeurs souhaitées, ou à l'intérieur de fonctions appartenant à des classes amies.

Les principales fonctionnalités du jeu Tetris sont implémentées de manière interne afin de renforcer la sécurité globale du système et de garantir un encapsulation efficace. Cette approche contribue également à une meilleure robustesse du code, en réduisant les dépendances externes et en limitant les accès non autorisés aux données sensibles.

Le choix des structures de listes chaînées pour le stockage des pièces permet une rotation plus efficace et flexible. L'ensemble de l'encapsulation est conçu en tenant compte de la séquence finale du jeu, ce qui améliore la stabilité, la cohérence du fonctionnement et la fiabilité globale de l'application.

6 ANALYSE EXPÉRIMENTALE

L'analyse expérimentale du jeu a été réalisée principalement en se concentrant sur le temps d'exécution et l'utilisation du processeur (CPU). Étant donné qu'il s'agit d'un jeu relativement simple, avec des graphismes peu exigeants, il est possible d'observer que l'utilisation du CPU reste faible. La majeure partie du temps, le processeur demeure en mode idle, comme illustré à la Figure 6, notamment parce que le code adopte un temps discret pour les mises à jour, en utilisant la fonction EventTrigger. Cette approche permet de limiter les calculs inutiles et de mieux contrôler le cycle d'exécution du jeu.

Comme il s'agit d'un code séquentiel, aucun mécanisme de parallélisme n'a été implémenté, puisqu'il n'apporterait que peu d'avantages dans ce contexte. En effet, le jeu présente déjà de bonnes performances et l'ensemble des actions dépend directement des interactions du joueur, ce qui réduit l'intérêt d'une exécution concurrente.

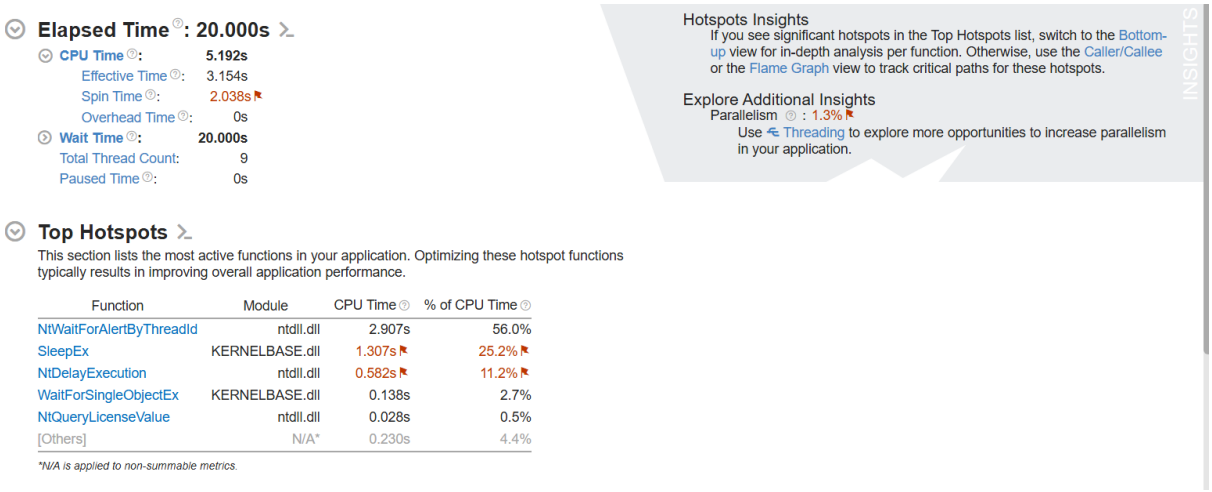

Figura 6 – Utilisation de la CPU

En ce qui concerne les fonctions, il est possible d'observer à la Figure 7 que lors des transitions entre les différentes interfaces ou écrans possibles du jeu, un plus grand nombre de fonctions internes est sollicité. Cela indique une adaptation interne de la logique du jeu en fonction de son état. Une fois cette phase de transition terminée, le jeu revient à un fonctionnement continu et stable, en utilisant principalement les fonctions HandleInput et Draw, qui assurent respectivement la communication avec le joueur et l'affichage visuel.

Enfin, les concepts de la programmation orientée objet jouent un rôle fondamental dans l'implémentation du jeu. Ils permettent de structurer le code de manière claire et cohérente, facilitant non seulement la compréhension du problème, mais contribuant également à l'amélioration des performances, à l'encapsulation des données et au maintien d'un modèle d'exécution interne uniforme et robuste.

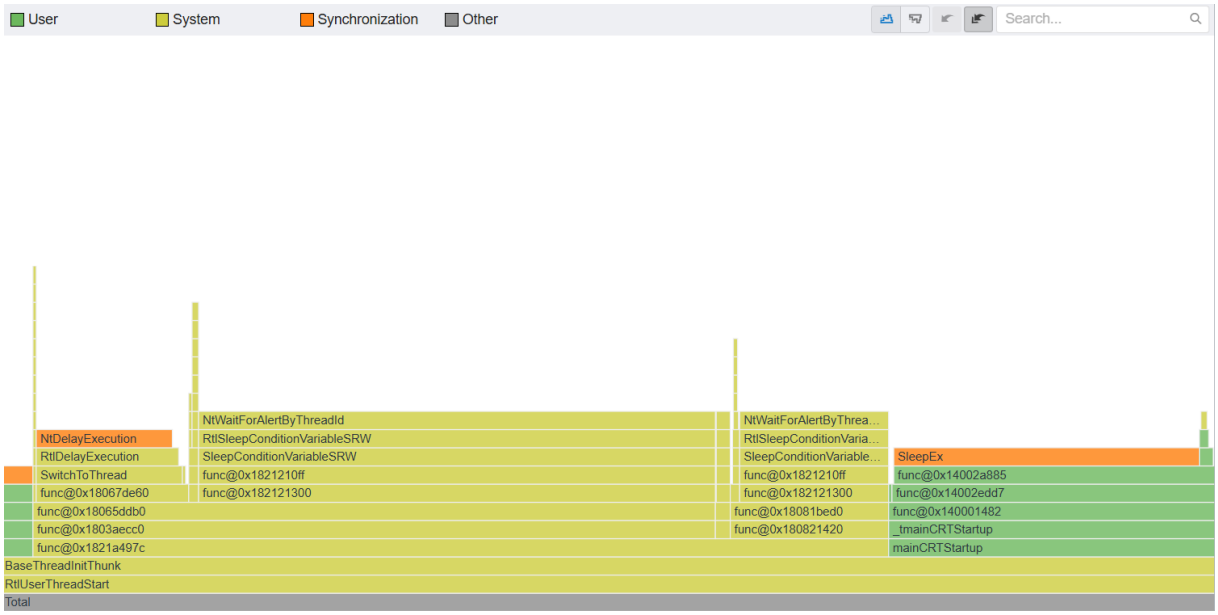

Figura 7 – Flame Graph de les fonctions

A DIAGRAME DE CLASSES