

PROJET Arcade

ETUDIANTS: PORTOIS Loïc
CHEKHAR Islam

ENSEIGNANTS: DESPLANCHE Charles-Henri
LEGRAND Gabin



ETABLISSEMENT HABILITE PAR LA COMMISSION DES TITRES D'INGENIEUR
ET MEMBRE DE LA CONFERENCE DES GRANDES ECOLES



14 quai de la Somme – BP 10100
80082 AMIENS CEDEX 2

Tél. : 03.22.66.20.00 – Fax : 03.22.66.20.10
<https://amiens.unilasalle.fr/>



1 Table des matières

I. Introduction

- 1.1. Tetris : Genèse d'un monument vidéoludique
- 1.2. L'impact mondial : D'un jeu de puzzle à un phénomène de société
- 1.3. Cadre du projet : Le défi de la programmation bas niveau
- 1.4. Ambition du projet : Vers une expérience "Arcade Perfect"

II. Architecture Logicielle et Environnement de Développement

- 2.1. Choix des outils et technologies (C, SDL2, VSCode)
- 2.2. Le système de build Cross-Platform : Pourquoi CMake ?
- 2.3. Organisation modulaire du code source
- 2.4. Gestion des ressources (Assets, Mémoire et Pointeurs)

III. Conception Algorithmique et Logique de Jeu

- 3.1. Structure de données : La matrice de jeu
- 3.2. La Boucle Principale (Game Loop) *[Algorigramme]*
- 3.3. Gestion des collisions et déplacements *[Algorigramme]*
- 3.4. Algorithmes avancés : *
- 3.4.1. Générateur aléatoire équitable ("7-Bag Randomizer")
- 3.4.2. Système de rotation avancé (SRS et Wall Kicks)
- 3.5. Gestion des lignes et calcul du score *[Algorigramme]*

IV. Implémentation Graphique et Sonore (SDL2)

- 4.1. Gestion de l'affichage et résolutions dynamiques
- 4.2. Système de textures et packs graphiques
- 4.3. Gestion événementielle (Clavier, DAS & ARR)
- 4.4. Moteur audio : Ambiance et effets sonores (SDL_Mixer)

V. Fonctionnalités Avancées et Persistance des Données

- 5.1. Système de sauvegarde binaire (Sérialisation)

- 5.2. Architecture des menus et machine à états
- 5.3. Paramétrage utilisateur (Remapping des touches, Audio)

VI. Résultats et Démonstration

- 6.1. Présentation de l'interface (Menu, Options)
- 6.2. Déroulement d'une partie (Gameplay, Ghost Piece, Hold)
- 6.3. Gestion de la défaite et High Scores

VII. Bilan et Conclusion

- 7.1. Difficultés rencontrées et solutions techniques
- 7.2. Compétences acquises
- 7.3. Axes d'amélioration (IA, Multijoueur)

I. Introduction

Ce premier chapitre contextualise le projet en revenant sur l'histoire emblématique de Tetris, de sa création soviétique à son impact culturel mondial. Il définit ensuite le cadre pédagogique de ce développement, centré sur la programmation bas niveau, et expose l'ambition technique de recréer une expérience fidèle aux standards modernes (Arcade Perfect).

1.1 Tetris : Genèse d'un monument vidéoludique

Plus qu'un simple jeu vidéo, *Tetris* est une icône culturelle universelle. Conçu en 1984 par l'ingénieur soviétique Alexey Pajitnov sur un ordinateur *Electronika 60*, le jeu tire son nom de la contraction entre "tétromino" (une figure géométrique composée de quatre carrés) et "tennis", le sport favori de son créateur.

Le concept repose sur une prémisse d'une simplicité désarmante : organiser des formes géométriques tombant du haut de l'écran pour former des lignes complètes, qui disparaissent alors pour libérer de l'espace. Pourtant, derrière cette apparente simplicité se cache une complexité algorithmique et psychologique fascinante, souvent qualifiée d'état de "flow" ou d'hypnose cognitive, connue sous le nom d'Effet Tetris.

Propulsé au rang de phénomène mondial grâce à son inclusion avec la Game Boy de Nintendo en 1989, Tetris a traversé les décennies sans jamais vieillir. Il reste aujourd'hui le cas d'école par excellence pour tout développeur : il exige une gestion rigoureuse de l'espace mémoire, une logique de collision précise et une boucle de jeu (Game Loop) parfaitement huilée. C'est dans cette lignée historique et technique que s'inscrit ce projet.

1.2 Tetris un monument du jeu vidéo

Si Tetris est devenu le jeu vidéo le plus vendu de tous les temps (toutes plateformes confondues), ce n'est pas un hasard. Son ascension fulgurante symbolise à elle seule la démocratisation du jeu vidéo.

Le catalyseur du jeu portable Le véritable point de bascule survient en 1989. Nintendo, préparant le lancement de sa révolutionnaire Game Boy, cherche le titre parfait pour accompagner la console. Henk Rogers, entrepreneur visionnaire, convainc Nintendo d'inclure Tetris plutôt qu'un jeu d'action classique, avec cet argument désormais célèbre : *"Si vous incluez Mario, la console sera pour les enfants. Si vous incluez Tetris, elle sera pour tout le monde."* Le pari est gagnant : Tetris devient la première "Killer App" de l'histoire du jeu portable, se vendant à plus de 35 millions d'exemplaires sur ce support et transformant le jeu vidéo en un loisir grand public, accessible aux adultes comme aux enfants.

Une universalité abstraite La force de Tetris réside dans son abstraction totale. N'ayant ni personnages, ni scénario, ni violence, ni références culturelles spécifiques (hormis sa musique folk russe *Korobeiniki*), le jeu brise la barrière de la langue. Il repose sur une boucle de gameplay pure : la satisfaction cognitive de mettre de l'ordre dans le chaos. Ce concept

de "tâche incomplète" qui doit être résolue exploite une faille psychologique humaine (l'effet Zeigarnik), rendant le jeu intrinsèquement addictif et rejouable à l'infini.

Un sujet d'étude scientifique L'importance de Tetris dépasse le cadre du divertissement pour toucher aux sciences cognitives. Le jeu a donné son nom à un phénomène scientifique reconnu : "L'Effet Tetris" (ou *Tetris Effect*), où le cerveau commence à percevoir des motifs géométriques et des opportunités d'emboîtement dans la vie réelle après de longues sessions de jeu. Plus récemment, des études en psychologie clinique ont même démontré que jouer à Tetris peu après un traumatisme pouvait aider à réduire les symptômes de stress post-traumatique (PTSD), en occupant les circuits visuo-spatiaux du cerveau et en limitant la formation de souvenirs intrusifs.

Ainsi, programmer un Tetris aujourd'hui, c'est toucher à un morceau d'histoire qui a non seulement défini l'industrie du logiciel, mais qui a également servi de pont culturel inattendu entre l'Est et l'Ouest en pleine Guerre Froide.

1.3 Cadre du projet : Le défi de la programmation bas niveau

Ce projet a été réalisé dans le cadre du module "Algorithmique et Programmation" à UniLaSalle. L'objectif pédagogique était double : consolider la maîtrise du langage C et appréhender la programmation événementielle et graphique via la librairie SDL2 (Simple DirectMedia Layer).

Contrairement aux moteurs de jeu modernes (comme Unity ou Unreal5) qui automatisent la physique et l'affichage, programmer un Tetris en C et SDL2 impose de revenir aux fondamentaux de l'informatique :

- Gestion manuelle de la mémoire : Allocation et libération dynamique (malloc/free) pour éviter les fuites de mémoire.
- Manipulation de pointeurs : Pour gérer les structures de données complexes (grille, pièces, textures).
- Mathématiques appliquées : Calculs matriciels pour les rotations et la détection de collisions.

1.4 Ambition du projet : Une approche "Arcade Perfect" et Cross-Platform

Au-delà du cahier des charges initial (une grille et des pièces qui tombent), j'ai souhaité orienter ce projet vers une réalisation semi-professionnelle, visant une expérience utilisateur fluide et moderne, souvent qualifiée de "Modern Tetris" par la communauté.

Notre implémentation ne se contente pas de déplacer des blocs, elle intègre les standards officiels des jeux de compétition actuels :

1. Mécaniques Avancées (SRS) : Implémentation du *Super Rotation System*, permettant aux pièces de "glisser" ou de se recaler intelligemment contre les

murs (Wall Kicks) lors des rotations, rendant le jeu moins frustrant et plus dynamique.

2. Équité Algorithmique (7-Bag Randomizer) : Utilisation d'un générateur aléatoire par "sacs" de 7 pièces, garantissant au joueur de recevoir toutes les formes régulièrement, éliminant ainsi les séquences de malchance pure.
3. Architecture Cross-Platform (CMake) : Une attention particulière a été portée à l'ingénierie logicielle. Grâce à l'utilisation de CMake, le projet est compilable et exécutable indifféremment sur Windows, Linux et macOS, sans modification du code source.

II. Architecture Logicielle et Environnement de Développement

La réussite d'un projet de développement repose avant tout sur un socle technique solide. Cette partie justifie le choix technologique du couple C/SDL2, détaille l'environnement de compilation cross-platform via CMake, et décrit l'architecture modulaire adoptée pour garantir la maintenabilité du code ainsi qu'une gestion rigoureuse des ressources système.

2.1. Choix des outils et technologies (C, SDL2, VSCode)

La réalisation d'un projet de cette nature nécessite une sélection rigoureuse d'outils permettant un contrôle total sur l'exécution du programme.

- Le Langage C : Choisi pour sa proximité avec le matériel et sa gestion explicite de la mémoire. Contrairement aux langages de haut niveau (Python, Java) qui masquent la complexité via un Garbage Collector, le C impose une rigueur absolue. Cela permet une optimisation fine des performances, cruciale pour maintenir une fluidité constante (60 FPS) nécessaire au gameplay rapide de Tetris.
- La librairie SDL2 (Simple DirectMedia Layer) : Plutôt qu'un moteur de jeu "clé en main" (comme Unity), l'utilisation de la SDL2 s'est imposée. C'est une bibliothèque de bas niveau qui fournit une abstraction légère sur le matériel graphique (OpenGL/Direct3D), l'audio et les périphériques d'entrée. Dans le fichier draw.c, elle permet de manipuler directement des textures (SDL_Texture) et des rectangles de rendu, offrant une compréhension profonde du pipeline graphique.
- VSCode et Environnement : Visual Studio Code a été utilisé comme éditeur principal pour sa légèreté et son écosystème d'extensions C/C++, facilitant le débogage et l'autocomplétion (IntelliSense) lors de la manipulation des nombreuses structures du projet.

2.2. Le système de build Cross-Platform : Pourquoi CMake ?

La compilation d'un projet C impliquant des dépendances externes comme SDL2, SDL_Image, SDL_TTF et SDL_Mixer peut rapidement devenir complexe. Pour pallier les différences entre les systèmes d'exploitation (Windows gérant les chemins et les bibliothèques différemment de Linux ou macOS), l'utilisation d'un système de méta-build comme CMake est essentielle.

CMake permet de générer automatiquement les fichiers de construction (Makefiles sous Linux/Mac ou Solutions Visual Studio sous Windows) adaptés à la machine de l'utilisateur. Il automatise la détection des fichiers d'en-tête et l'édition de liens (linking) des librairies dynamiques, garantissant que le projet est portable et compilable sur n'importe quelle plateforme sans modification du code source.

2.3. Organisation modulaire du code source

Afin de garantir la maintenabilité et l'évolutivité du projet, le code a été structuré selon une approche modulaire stricte, s'inspirant du patron de conception MVC (Modèle-Vue-Contrôleur), bien que non natif en C procédural. L'architecture sépare clairement les responsabilités :

- Le Modèle (Logique Pure) - `logic.c / logic.h` : Ce module contient toute l'intelligence du jeu. Il gère la matrice de jeu (`int board[20][10]`), les règles de collision, le système de rotation SRS et le calcul des scores. Il est totalement indépendant de l'affichage : il ne "sait" pas que la SDL existe.
- La Vue (Affichage) - `draw.c / draw.h` : Ce module est le seul autorisé à communiquer avec la SDL. Il reçoit l'état du jeu et se charge de dessiner les textures des blocs, le texte (scores, menus) et de jouer les sons.
- Le Contrôleur (Entrées) - `input.c / input.h` : Il intercepte les événements clavier bruts (via `SDL_PollEvent` dans `main.c`) et les traduit en actions sémantiques compréhensibles par le jeu (ex: `ACTION_LEFT`, `ACTION_ROTATE`).
- Définitions partagées - `defs.h` : Ce fichier centralise les constantes globales (taille de la grille, délais DAS/ARR) et les structures de données principales (`GameContext`, `AppContext`), agissant comme la "colonne vertébrale" reliant les modules.

2.4. Gestion des ressources (Assets, Mémoire et Pointeurs)

La gestion des ressources est un aspect critique en programmation C pour éviter les fuites de mémoire.

- Gestion des Assets : Les ressources graphiques (images PNG) et sonores (MP3, WAV) sont chargées dynamiquement au démarrage via la fonction `initSDL`. Elles sont stockées dans la structure `AppContext` sous forme de pointeurs (`SDL_Texture*`, `Mix_Music*`).

- **Passage par adresse (Pointeurs) :** Pour éviter de copier inutilement de lourdes structures de données à chaque appel de fonction, l'état du jeu (GameContext) et le contexte graphique (AppContext) sont systématiquement passés par pointeurs. Par exemple, la fonction `updateGame(GameContext* game, ...)` modifie directement l'état original du jeu en mémoire.
- **Libération de la mémoire :** Une attention particulière a été portée au nettoyage. La fonction `cleanupSDL` est impérativement appelée en fin de programme pour détruire proprement toutes les textures (`SDL_DestroyTexture`), libérer les musiques (`Mix_FreeMusic`) et fermer les contextes SDL, rendant ainsi la mémoire au système d'exploitation.

III. Conception Algorithmique et Logique de Jeu

Cœur névralgique du programme, la logique de jeu orchestre les règles et les mathématiques qui régissent la partie. Nous analyserons ici les structures de données fondamentales (matrices), le fonctionnement de la boucle principale, ainsi que les algorithmes complexes implémentés pour respecter les standards officiels, tels que le système de rotation SRS et la génération aléatoire équitable.

3.1. Structure de données : La matrice de jeu

La représentation numérique du terrain de jeu est fondamentale pour la logique de Tetris. J'ai opté pour une modélisation matricielle simple mais efficace, définie dans `defs.h` et implémentée dans `GameContext`.

La grille est stockée sous la forme d'un tableau à deux dimensions d'entiers : `int board[20][10]`.

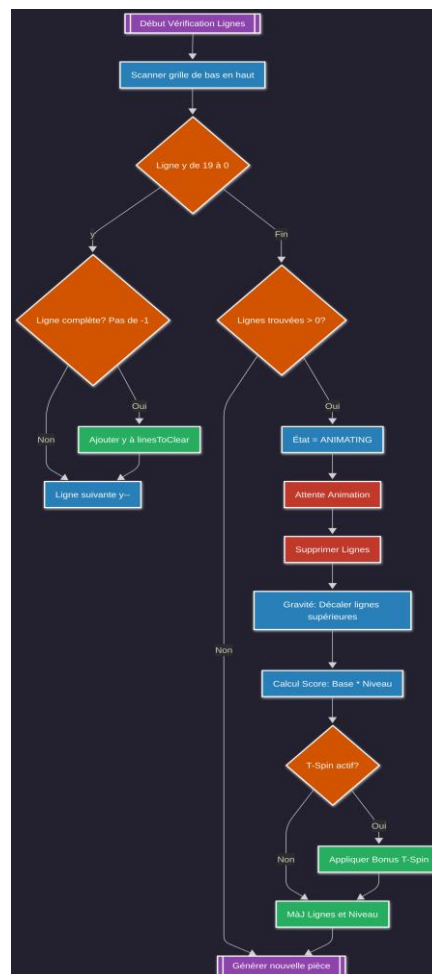
- **Dimensions :** La grille respecte les standards officiels avec 10 colonnes (`BOARD_WIDTH`) et 20 lignes (`BOARD_HEIGHT`).
- **Encodage :** Chaque cellule de la matrice stocke un entier :
 - -1 signifie que la case est vide.
 - Une valeur de 0 à 6 représente l'identifiant de la pièce (ou sa couleur) qui occupe cette case (ex: 0 pour la barre Cyan, 1 pour le carré Jaune, etc.).

La pièce active (`currentPiece`) n'est pas immédiatement intégrée à cette matrice. Elle est gérée par une structure distincte (`Piece`) contenant ses coordonnées `x`, `y`, son type et son état de rotation. Ce n'est qu'au moment du verrouillage (`Lock`) que les blocs de la pièce active sont transférés définitivement dans la matrice `board`.

3.2. La Boucle Principale (Game Loop)

Au cœur du programme, située dans le fichier main.c, la boucle principale orchestre le déroulement du jeu image par image. Elle suit le cycle classique de programmation temps réel :

1. Gestion du Temps (DeltaTime) : À chaque itération, le temps écoulé depuis la dernière image est calculé (SDL_GetTicks). Ce deltaTime est transmis à la logique pour gérer les minuteries (chute automatique, délai de blocage) indépendamment de la vitesse du processeur.
2. Entrées (Input) : La fonction SDL_PollEvent capture les interactions joueur (clavier) et les transmet au contrôleur.
3. Mise à jour (Update) : La fonction updateGame applique les règles du jeu : gravité, vérification des délais (Lock Delay) et mouvements automatiques (DAS).
4. Rendu (Render) : Enfin, renderGame dessine l'état actuel de la matrice et de la pièce active à l'écran.



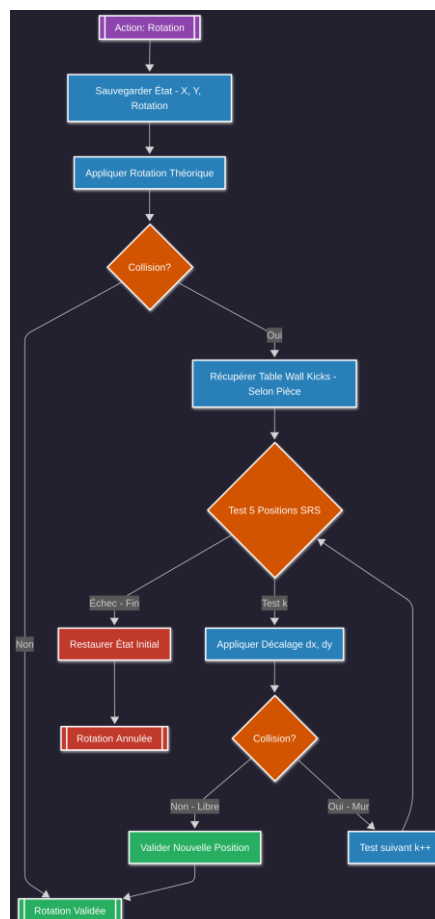
<https://drive.google.com/file/d/1xqmNGWxmzzbQen-YeshrtXbDpQXUKP1T/view?usp=sharing>

3.3. Gestion des collisions et déplacements

Pour garantir l'intégrité du jeu, aucun déplacement n'est appliqué sans validation préalable. La fonction centrale checkCollision dans logic.c agit comme un "garde-fou".

L'algorithme fonctionne par anticipation :

1. Avant de valider un déplacement (gauche, droite, bas ou rotation), le jeu calcule les coordonnées théoriques futures des 4 blocs composant la pièce.
2. Il itère sur ces 4 blocs et vérifie deux conditions pour chacun :
 - a. Limites du plateau : Le bloc sort-il de la grille ($x < 0$, $x \geq 10$, $y \geq 20$) ?
 - b. Occupation : La case cible dans la matrice board est-elle déjà occupée (valeur différente de -1) ?
3. Si une seule de ces conditions est vraie, le mouvement est annulé et la pièce reste à sa position initiale.



<https://drive.google.com/file/d/1s0kmbr9LH-SX0wsPnfPVPLuqFHhybCsl/view?usp=sharing>

3.4. Algorithmes avancés : Implémentation des standards officiels

Pour offrir une expérience de jeu moderne ("Modern Tetris"), deux algorithmes spécifiques ont été implémentés dans `logic.c`.

3.4.1. Générateur aléatoire équitale ("7-Bag Randomizer")

L'aléatoire pur (`rand() % 7`) est problématique dans Tetris car il peut générer des séquences frustrantes (ex: trois pièces "S" d'affilée ou aucune barre "I" pendant 50 tours).

Pour pallier cela, j'ai implémenté le système du "7-Bag" (Sac de 7) dans la fonction `getRandomPieceType`.

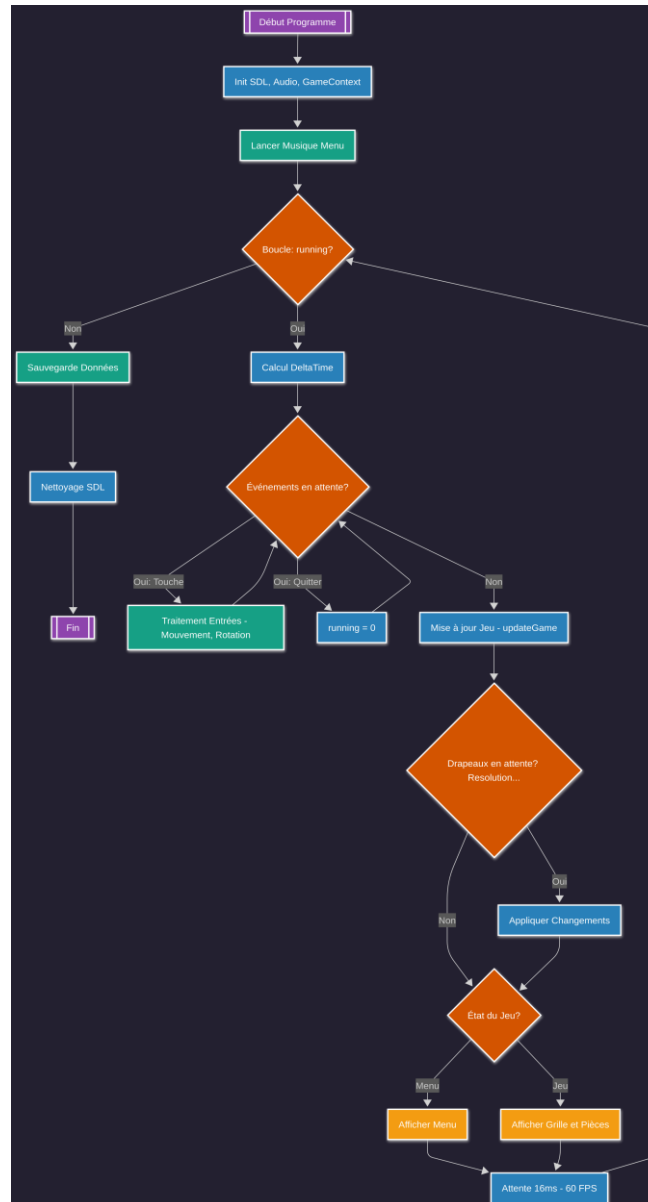
- Principe : Le jeu génère une séquence contenant exactement un exemplaire de chacune des 7 pièces.
- Mélange : Cette séquence est mélangée (shuffled) aléatoirement.
- Distribution : Les pièces sont distribuées une à une. Une fois le "sac" vide, un nouveau sac de 7 pièces est généré et mélangé. Cela garantit mathématiquement que le joueur ne peut jamais attendre plus de 12 pièces avant de revoir une forme spécifique.

3.4.2. Système de rotation avancé (SRS et Wall Kicks)

La rotation des pièces est complexe car une pièce ne tourne pas simplement autour d'un pixel, mais à l'intérieur d'une boîte de délimitation (bounding box). Le **Super Rotation System (SRS)** a été implémenté pour gérer les cas limites.

Lorsqu'une rotation classique échoue (car la pièce heurterait un mur ou un autre bloc), le jeu ne bloque pas l'action. Il consulte les tables de **"Wall Kicks"** (Sauts de mur) définies dans `logic.c` (tableaux `WALL_KICKS_JLSTZ` et `WALL_KICKS_I`).

L'algorithme tente séquentiellement 5 positions alternatives (décalage de 1 case à droite, 1 case en haut, etc.). La première position valide est acceptée. C'est ce qui permet aux joueurs experts d'effectuer des manœuvres techniques comme les "T-Spins".



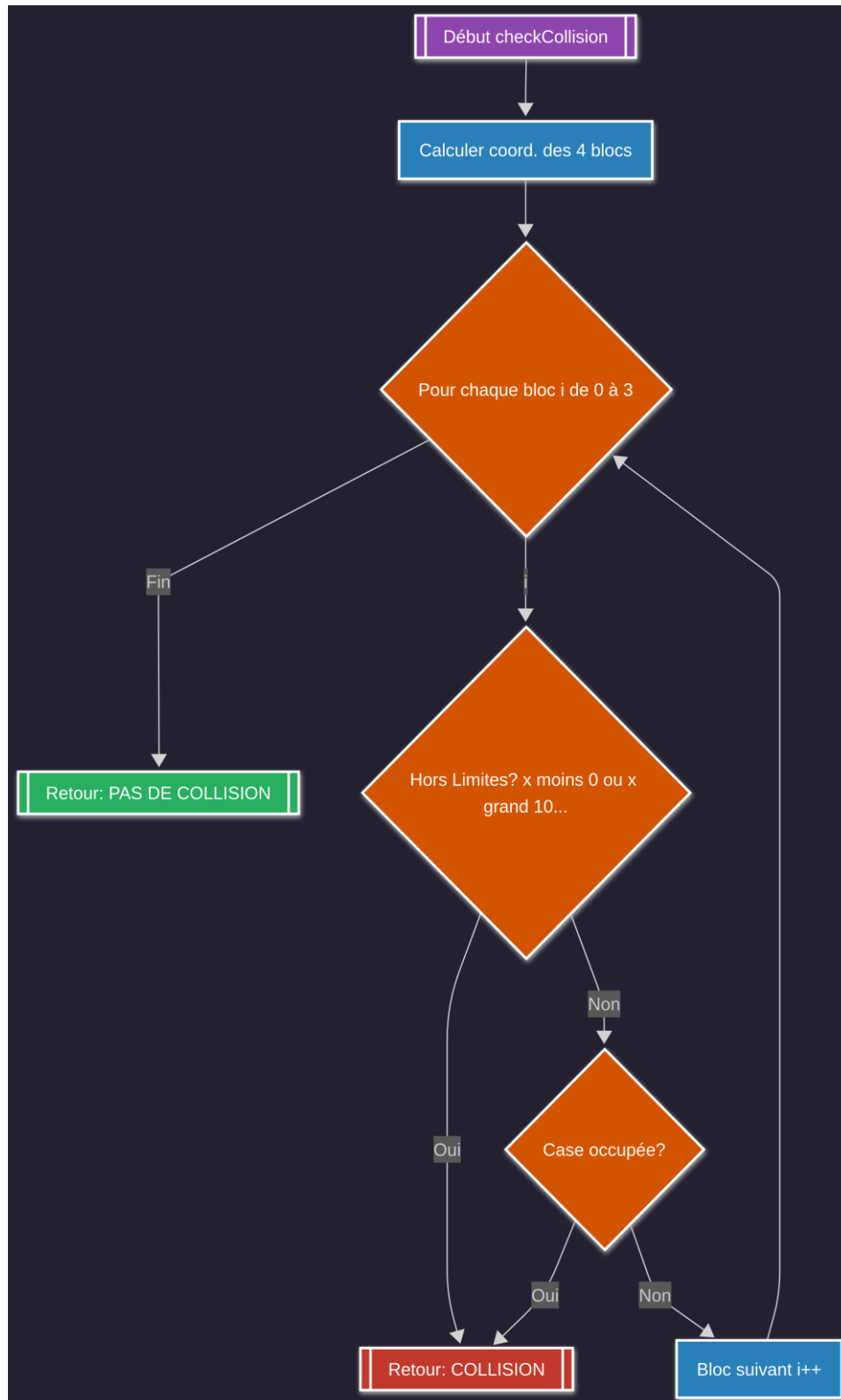
<https://drive.google.com/file/d/1WLPglpWBNPCBIWSOBLybGL-YxaNA3lQu/view?usp=sharing>

3.5. Gestion des lignes et calcul du score

La suppression des lignes est gérée par la fonction detectLines et performLineClear.

1. Détection : Après le verrouillage d'une pièce, l'algorithme scanne la grille de bas en haut. Si une ligne ne contient aucune case vide (-1), son index est ajouté à une liste linesToClear.
2. Animation : Le jeu passe brièvement en état STATE_ANIMATING pour donner un feedback visuel au joueur.
3. Suppression et Gravité : Les lignes identifiées sont supprimées. Un algorithme de "gravité naïve" décale ensuite toutes les lignes situées au-dessus vers le bas.

4. Scoring : Le score est calculé en fonction du nombre de lignes effacées simultanément (100, 300, 500 ou 800 points), multiplié par le niveau actuel. Des bonus spécifiques sont attribués si un T-Spin a été détecté juste avant le verrouillage.



https://drive.google.com/file/d/1nVtSQToyD_6EP_7rjU8EdSmZYyi5j4lP/view?usp=sharing

IV. Implémentation Graphique et Sonore (SDL2)

L'interaction utilisateur est primordiale pour transformer des algorithmes abstraits en une expérience ludique fluide. Ce chapitre détaille l'utilisation de la bibliothèque SDL2 pour la gestion de l'affichage dynamique, le rendu des textures, le traitement réactif des entrées clavier (mécaniques DAS/ARR) et l'immersion sonore via le mixage audio.

4.1. Gestion de l'affichage et résolutions dynamiques

Contrairement à une console de jeu où la résolution est fixe, un jeu PC doit s'adapter à différents écrans. J'ai implémenté un système de résolution flexible dans le module draw.c, piloté par la fonction applyWindowResolution.

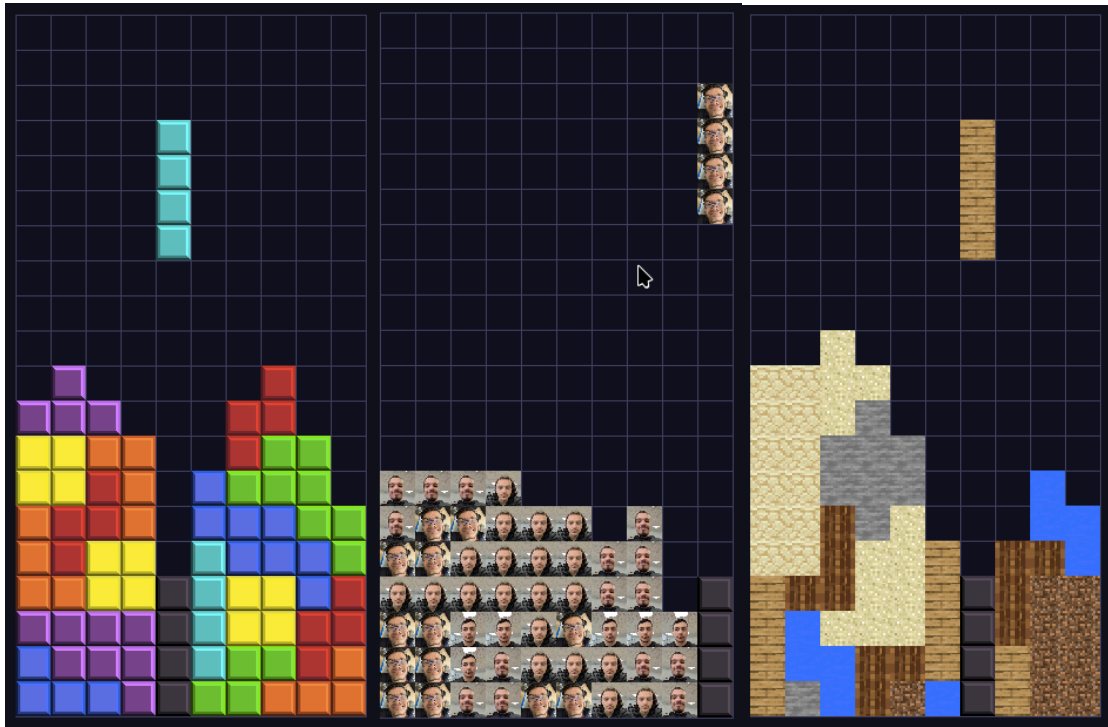
Pour éviter de recalculer les coordonnées de chaque bloc à chaque redimensionnement (ce qui serait fastidieux et source d'erreurs), j'utilise la fonctionnalité de "Logical Size" de la SDL2 (SDL_RenderSetLogicalSize).

- Principe : Le jeu "pense" et dessine toujours dans une grille virtuelle fixe de 800x700 pixels (LOGICAL_WIDTH et LOGICAL_HEIGHT).
- Mise à l'échelle : Le GPU se charge d'étirer cette image virtuelle pour qu'elle remplisse la fenêtre physique, qu'elle soit en 1200x1050 ou en Plein Écran (SDL_WINDOW_FULLSCREEN_DESKTOP). Cela garantit que le ratio d'aspect est conservé et que l'interface reste lisible sur tous les moniteurs.

4.2. Système de textures et packs graphiques

Afin de varier l'esthétique du jeu sans modifier le code, j'ai mis en place un système de chargement de ressources dynamique. La fonction loadBlockTextures accepte un paramètre style qui détermine quel sous-dossier d'assets est scanné (default, alt, ou mc pour un style Minecraft).

Le chargement s'appuie sur SDL_image (IMG_Load) pour convertir les fichiers PNG en SDL_Texture, optimisés pour le rendu matériel. Ces textures sont stockées dans le tableau blockTextures[7] de la structure AppContext. Lorsqu'un joueur change de style dans les menus, un indicateur reloadAssetsPending est activé dans la boucle principale, déclenchant le rechargement immédiat des textures sans redémarrer le jeu.



Comparaison des différents packs de textures chargés dynamiquement.

4.3. Gestion événementielle (Clavier, DAS & ARR)

La réactivité est le critère numéro un d'un Tetris moderne. Le traitement des entrées se fait en deux étapes distinctes :

1. Abstraction Physique/Logique : Le fichier input.c sépare la touche pressée de l'action en jeu. Le tableau KeyBinding associe un code touche SDL (ex: SDLK_UP) à une intention de jeu (ex: ACTION_UP). Cela permet au joueur de reconfigurer ses touches sans que la logique de jeu n'ait à connaître la configuration physique du clavier.
2. Mécanique DAS & ARR : Pour permettre des déplacements rapides, j'ai implémenté le standard DAS (Delayed Auto Shift) et ARR (Auto Repeat Rate) directement dans la fonction updateGame.
 - a. Phase 1 (DAS) : Lorsqu'une touche est maintenue, le jeu attend 170ms (DAS_DELAY) pour confirmer l'intention du joueur.
 - b. Phase 2 (ARR) : Une fois ce délai passé, la pièce se déplace latéralement toutes les 50ms (ARR_RATE) tant que la touche reste enfoncée.
 - c. Interruption : L'événement SDL_KEYUP est capté dans main.c via handleInputUp pour réinitialiser immédiatement ce compteur dès que le joueur relâche la touche, offrant une précision au pixel près.

4.4. Moteur audio : Ambiance et effets sonores (SDL_Mixer)

L'immersion sonore est gérée par l'extension SDL_mixer via le module audio.c, qui distingue deux types de flux :

- Musique (Mix_Music) : Les fichiers MP3 (comme tetris1.mp3) sont streamés depuis le disque, car ils sont trop volumineux pour être chargés entièrement en RAM. Le jeu propose 12 pistes musicales sélectionnables.
- Effets Sonores (Mix_Chunk) : Les bruitages courts, comme la validation d'une ligne (clear.mp3), sont pré-chargés en mémoire brute pour être joués instantanément sans latence (playClearSound).

Le système audio intègre également une gestion fine du volume. La structure audioSystem stocke le volume sous forme de flottant (0.0 à 1.0) pour l'interface utilisateur, et le convertit en valeur interne SDL (0 à 128) via la fonction ApplyVolume avant de l'envoyer au mixeur.

V. Fonctionnalités Avancées et Persistance des Données

"Au-delà du gameplay immédiat, un logiciel complet doit offrir une ergonomie de configuration et une continuité de progression. Nous aborderons ici les mécanismes de persistance des données par sérialisation binaire, l'architecture interne basée sur une machine à états, ainsi que les systèmes de personnalisation des contrôles et de l'audio offerts au joueur."

5.1. Système de sauvegarde binaire (Sérialisation)

Pour permettre au joueur de reprendre une partie interrompue ou de conserver ses réglages, j'ai implémenté un système de sérialisation binaire directe (Raw Binary Dump).

Contrairement aux formats texte (JSON ou XML) qui nécessitent un parsing coûteux, la sauvegarde binaire consiste à écrire l'état exact de la mémoire vive directement sur le disque dur via la fonction fwrite.

- Structure sauvegardée : Le fichier save.dat contient une copie brute de la structure GameContext (qui inclut la grille, le score, la pièce en cours) ainsi que la configuration des touches.
- Mécanisme : La fonction saveGameData ouvre le fichier en mode binaire ("wb") et y déverse le contenu des structures. Un indicateur (saveFullState) permet de distinguer si l'on sauvegarde une partie en cours (avec la position des blocs) ou simplement les préférences du menu.
- Chargement : Au démarrage, loadGameData effectue l'opération inverse (fread), restaurant instantanément l'état du jeu tel qu'il était à la fermeture, y compris les positions précises des pièces et les timers.

5.2. Architecture des menus et machine à états

La structure globale du programme repose sur un automate fini (Finite State Machine - FSM), défini par l'énumération GameState dans defs.h.

Cet automate pilote le flux d'exécution dans la boucle principale (main.c) et garantit que les entrées clavier sont interprétées contextuellement :

- États définis : Le jeu peut se trouver dans l'état STATE_MENU, STATE_PLAYING, STATE_SETTINGS, STATE_KEY_CONFIG ou STATE_GAMEOVER.
- Séparation logique :
 - Dans logic.c, la fonction handleInput utilise un switch(game->state) pour déterminer si la touche "Flèche Haut" doit déplacer le curseur du menu ou effectuer un "Hard Drop" de la pièce.
 - Dans draw.c, le rendu est également conditionné par cet état : renderMenu est appelé si l'état est un menu, sinon renderGame prend le relais. Cette architecture évite l'utilisation de multiples drapeaux booléens et rend le code extensible (ajout facile d'un état "Crédits" ou "Pause" par exemple).

5.3. Paramétrage utilisateur (Remapping des touches, Audio)

Afin d'assurer l'accessibilité et le confort de jeu (notamment pour les claviers AZERTY vs QWERTY), j'ai développé un système complet de configuration.

- Remapping des touches (Input Mapping) : Le fichier input.c gère un tableau de KeyBinding. Chaque action du jeu (ex: ACTION_LEFT) possède deux "slots" : une touche primaire et une touche secondaire. L'interface de configuration (STATE_KEY_CONFIG) permet au joueur de modifier dynamiquement ces assignations. Lorsqu'une modification est effectuée via UpdateKeyBinding, elle est immédiatement active et sera persistée dans le fichier de sauvegarde via SaveInputProfile.
- Gestion Audio : Le module audio offre un contrôle granulaire. La structure AudioState maintient le volume global (0.0 à 1.0) et l'état "Mute". Ces préférences sont modifiables en temps réel dans le menu STATE_SETTINGS et appliquées via Mix_VolumeMusic et Mix_Volume pour les effets sonores.

VI. Résultats et Démonstration

"Cette section présente le fruit du développement à travers l'analyse de l'interface finale et du déroulement concret d'une partie. Nous y détaillerons l'ergonomie des menus, l'intégration des aides visuelles en jeu (Ghost Piece, Hold) et la gestion de la fin de partie, validant ainsi l'atteinte des objectifs fixés dans le cahier des charges."

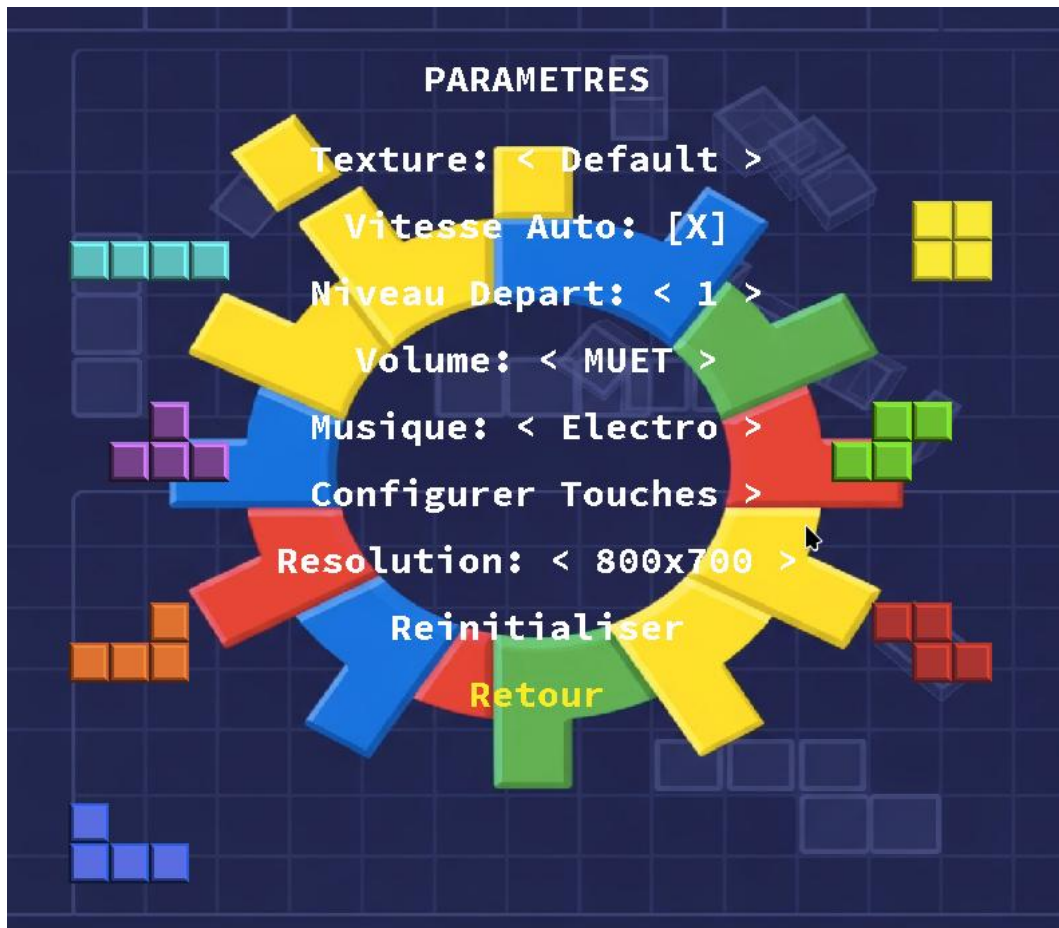
6.1. Présentation de l'interface (Menu, Options)

Dès le lancement, l'utilisateur est accueilli par une interface graphique soignée, rendue via la SDL2.

- Menu Principal : L'écran titre affiche le logo "TETRIS" ainsi que les records actuels ("High Score" et "Best Lines") récupérés depuis le fichier de sauvegarde. La navigation se fait au clavier, avec un système de surbrillance jaune indiquant l'option sélectionnée.
- Menu Paramètres (STATE_SETTINGS) : Ce sous-menu démontre la flexibilité du moteur. Il permet de modifier la résolution (800x700, 1200x1050 ou Plein Écran), de changer le style visuel des blocs (Default, Alt, Minecraft) et d'ajuster le volume audio en temps réel. Un aperçu visuel des 7 pièces s'affiche en arrière-plan pour valider immédiatement le choix du pack de textures.
- Configuration des touches (STATE_KEY_CONFIG) : Un écran dédié permet de remapper les actions (Gauche, Droite, Rotation, etc.) avec support d'une touche primaire et secondaire, assurant l'accessibilité pour tous les types de claviers.



Interface du menu principal et navigation au clavier.



Interface du menu des paramètres et navigation au clavier.

6.2. Déroulement d'une partie (Gameplay, Ghost Piece, Hold)

L'expérience de jeu ("In-Game") concentre l'essentiel des efforts d'ergonomie visuelle.

- L'Aire de Jeu : La grille est centrée, entourée par les éléments d'information essentiels (Score, Niveau, Lignes).
- Aides Visuelles (Ghost Piece & Preview) : Pour assister le joueur, une "Ghost Piece" (spectre) translucide est projetée au bas de la colonne, indiquant exactement où la pièce atterrira. À droite, la "Next Piece" permet d'anticiper le coup suivant.
- Mécaniques de Sauvetage (Hold) : La fonction de réserve ("Hold"), affichée à gauche de l'écran, permet de stocker une pièce tactiquement difficile pour l'utiliser plus tard via la touche C.
- Feedback Visuel : Lorsqu'une action technique est réalisée, comme un "TETRIS" (4 lignes) ou un "T-SPIN", un message textuel clignotant en or apparaît temporairement à l'écran pour gratifier le joueur.



1) Pièce 2) Ghost pièce 3) Hold 4) Piece Suivante 5) Score



Shéma du T spin

6.3. Gestion de la défaite et High Scores

La boucle de jeu se ferme proprement lorsque les pièces atteignent le haut de la grille.

- Game Over : L'état STATE_GAMEOVER déclenche l'affichage d'un bandeau sombre en superposition avec le texte "GAME OVER".
- Persistance des Records : À cet instant précis, le moteur compare le score actuel avec le highScore enregistré. Si le joueur a battu le record, la nouvelle valeur est immédiatement sérialisée dans le fichier save.dat, garantissant que la performance est conservée pour la prochaine session.



Image du game over

VII. Bilan et Conclusion

Pour clore ce rapport, nous dresserons un bilan critique du projet. Nous reviendrons sur les défis techniques surmontés, notamment ceux liés à la gestion manuelle de la mémoire et aux algorithmes de rotation, avant de synthétiser les compétences acquises et d'envisager les axes d'évolution futurs, tels que l'intégration d'une intelligence artificielle.

7.1. Difficultés rencontrées et solutions techniques

Le développement ne s'est pas fait sans heurts, plusieurs défis techniques majeurs ont dû être surmontés :

1. Complexité du SRS (Super Rotation System) : Implémenter les règles officielles de rotation avec les "Wall Kicks" a été la partie la plus ardue. Il a fallu transcrire des tables de données complexes (matrices de décalage WALL_KICKS) et gérer des systèmes de coordonnées rigoureux pour éviter que les pièces ne traversent les murs lors des rotations.
2. Gestion de la mémoire avec la SDL : Contrairement aux langages managés, la SDL en C oblige à une gestion manuelle stricte. J'ai dû apprendre à charger les textures et les sons une seule fois (InitSDL) et à les libérer proprement (CleanupSDL) pour éviter les fuites de mémoire qui ralentissaient le jeu sur la durée.
3. Synchronisation Temporelle : Gérer le temps (DeltaTime) pour que la chute des pièces soit indépendante de la vitesse du processeur, tout en gérant en parallèle des timers très courts pour le DAS (170ms), a nécessité une compréhension fine de la boucle de jeu principale.

7.2. Compétences acquises

Ce projet a été un vecteur d'apprentissage complet :

- Maîtrise du C Avancé : Manipulation intensive de pointeurs, de structures imbriquées et d'allocation dynamique.
- Architecture Logicielle : Compréhension du pattern "Game Loop" et séparation du code en modules (Logique / Rendu / Entrées).
- Traitement de fichiers : Lecture et écriture binaire pour la sauvegarde.
- Algorithmique graphique : Compréhension des systèmes de coordonnées, du blitting de textures et de la gestion des collisions matricielles.

7.3. Axes d'amélioration (IA, Multijoueur)

Bien que le jeu soit fonctionnel et complet ("Feature Complete"), plusieurs pistes d'évolution sont envisageables :

- Intelligence Artificielle : La séparation claire de la logique (logic.c) permettrait de brancher facilement une IA qui simulerait des entrées clavier pour jouer seule, en utilisant des heuristiques simples (minimiser la hauteur des trous).
- Mode Multijoueur : L'ajout d'un mode 1v1 (local ou réseau via SDL_Net) serait la suite logique. Cela impliquerait de gérer deux instances de GameContext simultanément et d'afficher deux grilles à l'écran.
- Effets de Particules : Remplacer l'animation simple de suppression de ligne par un système de particules (explosions de blocs) pour dynamiser encore davantage le rendu visuel.