Université d'Évry Paris-Saclay

Faculté des Sciences et Institut de Biologie Génétique et Bio-informatique

3ème année de Licence en Mathématiques et applications

université
PARIS-SACLAY

Tetris

Réalisé par :
SAMER Ahmed Wahid

Année académique : 2024 - 2025

Table des matières

Chapitre 1

Introduction

1.1 Contexte du projet

Tetris, un jeu emblématique créé en 1985 par Alekseï Pajitnov, Dmitri Pavlovski et Vadim Guerassimov, a marqué l'histoire des jeux vidéo grâce à son gameplay simple et captivant. L'objectif principal du jeu est de manipuler des pièces géométriques appelées *Tetriminos* pour former des lignes horizontales complètes, qui disparaissent lorsqu'elles sont remplies. Ce concept addictif a conduit à une popularité mondiale et à des adaptations sur presque toutes les plateformes.

1.2 Objectif du projet

Le projet vise à implémenter une version du jeu Tetris en respectant les spécifications essentielles du jeu original. Cela inclut :

— le déplacement et la rotation des Tetriminos,
— la gestion du score et du niveau en fonction des lignes complétées,
— l'adaptation de la vitesse des Tetriminos selon le niveau atteint,
— l'affichage des statistiques du jeu telles que le score, le niveau, et les lignes détruites,
— l'utilisation de la bibliothèque SDL pour la partie graphique.

1.3 Compilation du code

Le sujet du projet impose que le code source soit compilé avec la commande suivante :

```
$ gcc -g -O2 -Wall -Wextra -o tetris *.c $(pkg-config --cflags --libs
   sdl2 SDL2_ttf)
```

Cependant, dans notre implémentation, nous avons ajouté une fonctionnalité sonore en utilisant la bibliothèque `SDL2_mixer`. Par conséquent, la commande de compilation a été modifiée pour inclure cette dépendance :

```
$ gcc -g -O2 -Wall -Wextra -o Tetris *.c $(pkg-config --cflags --libs
   sdl2 SDL2_ttf SDL2_mixer)
```

Cette modification garantit une intégration fluide des éléments sonores.

Chapitre 2

Implémentation

Le projet est organisé en plusieurs fichiers source et d'en-tête pour structurer le code et faciliter son développement. Voici un aperçu des principaux fichiers et leur rôle :

2.1 Gestion des déplacements des Tetriminos

2.1.1 Représentation des Tetriminos

Les Tetriminos sont représentés sous forme de tableaux 5×5, où chaque case contient une valeur indiquant la présence ou l'absence d'une partie du bloc. Par exemple, un Tetrimino de type `I` est représenté ainsi lorsqu'il n'est pas encore tourné :

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

2.1.2 Types de déplacements

Les déplacements possibles sont :
— **Vers la gauche** : réduire la colonne du Tetrimino.
— **Vers la droite** : augmenter la colonne du Tetrimino.
— **Vers le bas** : augmenter la ligne du Tetrimino.
Avant d'effectuer un déplacement, une vérification est effectuée pour s'assurer que la nouvelle position est valide (pas de collision avec un autre bloc ou un mur).

2.1.3 Vérification des collisions

La fonction de vérification des collisions, comme `tetris_can_go_left`, parcourt chaque case du tableau `buffer` du Tetrimino. Si une case occupée par le Tetrimino entre en conflit avec un mur ou un bloc déjà fixé dans la matrice principale, le déplacement est annulé.

```
int tetris_can_go_left(Tetris *tet) {
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            if (tet->buffer[i][j]) { // Case occupée
                int row = tet->current_line + i;
                int col = tet->current_column + j;
                // Si on est au bord gauche ou collision à gauche
                if (col == 0 || tet->matrix[row][col - 1] != 0) {
                    return 0; // Déplacement impossible
                }
            }
        }
    }
    return 1; // Déplacement possible
}
```

2.1.4 Implémentation des déplacements

Exemple : Vérification à gauche Les fonctions de déplacement, comme `tetris_move_left`, modifient les coordonnées du Tetrimino si le déplacement est valide.

```
void tetris_move_left(Tetris *tet) {
    if (tetris_can_go_left(tet)) {
        tet->current_column--; // Déplace le Tetrimino d'une colonne à
   gauche
    }
}
```

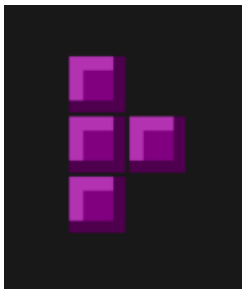
De manière similaire, les déplacements vers la droite (`tetris_move_right`) et vers le bas (`tetris_move_down`) suivent le même principe, mais en adaptant la vérification des collisions et la mise à jour des coordonnées.

2.2 Gestion des rotations des Tetriminos

2.2.1 Principe de la rotation

Les Tetriminos peuvent pivoter contre le sens des aiguilles d'une montre. Cette opération est réalisée en transposant leur tableau 5×5, puis en inversant les colonnes. Par exemple, une rotation de 90° du Tetrimino `T` passe de cette forme initiale :

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 6 & 6 & 0 \\ 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$



à :

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 6 & 0 & 0 \\ 0 & 6 & 6 & 6 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

2.2.2 Vérification de la validité de la rotation

Avant d'appliquer une rotation, une vérification est nécessaire pour s'assurer que :
— La nouvelle position du Tetrimino ne dépasse pas les limites de la matrice.
— La rotation ne provoque pas de collision avec d'autres blocs.
Cette vérification est réalisée par la fonction `tetris_is_valid_position`, qui vérifie chaque case du tableau après rotation.

```
int tetris_is_valid_position(Tetris *tet, char new_buffer[5][5]) {
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            if (new_buffer[i][j]) { // Case occupée
                int row = tet->current_line + i;
                int col = tet->current_column + j;
                // Vérification des limites
                if (row < 0 || row >= 20 || col < 0 || col >= 10)
   return 0; // Hors limites
                // Vérification des collisions
                if (tet->matrix[row][col]) return 0; // Collision dé
   tectée
            }
        }
    }
    return 1; // Rotation valide
}
```

2.2.3 Implémentation de la rotation

Exemple : Validation d'une rotation La rotation est gérée par cette fonction :
— `tetris_rotate` : Effectue la rotation en modifiant le tableau `buffer` si elle est valide.

```
void tetris_rotate(Tetris *tet) {
    int new_rotation = (tet->current_rotation + 1) % 4;
    char new_buffer[5][5] = {0};
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            new_buffer[j][4 - i] = tetriminos[tet->current_type][
   new_rotation][i][j];
        }
    }
    if (tetris_is_valid_position(tet, new_buffer)) {
        memcpy(tet->buffer, new_buffer, sizeof(new_buffer));
        tet->current_rotation = new_rotation;
    }
}
```

2.3 Calcul du score et des niveaux

2.3.1 Principe général

Le score et les niveaux sont des mécanismes clés pour récompenser le joueur et augmenter la difficulté progressivement. Chaque fois qu'une ou plusieurs lignes sont complétées, elles sont supprimées, et un score est attribué en fonction du nombre de lignes éliminées simultanément.

2.3.2 Calcul du score

Le score est calculé selon une formule qui dépend du nombre de lignes détruites en une seule fois et du niveau actuel du joueur. Plus il y a de lignes éliminées simultanément, plus la récompense est importante.

Table des points :

$$\text{Score} = \text{multiplicateur} \times \text{niveau}$$

où le multiplicateur dépend du nombre de lignes détruites :

— 1 ligne : 100
— 2 lignes : 300
— 3 lignes : 500
— 4 lignes : 800 (appelé un "Tetris").

Exemple : Un joueur au niveau 3 qui détruit 2 lignes obtiendra $300 \times 3 = 900$ points.

```c
if (lines_cleared > 0) {
    tet->nbr_lines += lines_cleared; // Mise à jour des lignes totales

    switch (lines_cleared) {
        case 1:
            tet->score += 100 * tet->level;
            break;
        case 2:
            tet->score += 300 * tet->level;
            break;
        case 3:
            tet->score += 500 * tet->level;
            break;
        case 4: // "Tetris"
            tet->score += 800 * tet->level;
            break;
        default:
            // Rare, au cas où plus de 4 lignes sont détruites simultan
    ément
            tet->score += 800 * tet->level;
            break;
    }
}
```

2.3.3 Gestion des niveaux

Extrait de code : Gestion des points Le niveau du joueur augmente au fur et à mesure qu'il progresse dans le jeu. Chaque fois que le joueur élimine un total de 10 lignes, le niveau augmente de 1, jusqu'à un maximum de 15.

```
while (tet->nbr_lines >= 10) {
    tet->nbr_lines -= 10; // Réinitialise les lignes comptabilisées
   pour le niveau actuel
    tet->level++;
    if (tet->level > 15) {
        tet->level = 15; // Niveau maximal
        break;
    }
}
```

2.3.4 Vitesse de descente des Tetriminos

Extrait de code : Mise à jour du niveau Avec chaque niveau, la vitesse à laquelle les Tetriminos descendent augmente. Cette vitesse est calculée à l'aide d'une formule :

$$\text{vitesse} = 0.8 - (0.007 \times (\text{niveau} - 1))^{\text{niveau}-1}$$

À partir du niveau 15, la vitesse est fixée à 0.007 seconde par ligne.

Table approximative des vitesses :

Niveau	Vitesse (secondes par ligne)
1	1.0
5	0.355
10	0.064
15	0.007

```
float tetris_get_drop_speed(Tetris *tet) {
    int l = tet->level;
    if (l < 1) l = 1;
    if (l > 15) l = 15;

    float base = 0.8f - ((l - 1) * 0.007f);
    if (base < 0.0f) base = 0.0f;

    float speed = powf(base, (float)(l - 1));
    return speed < 0.007f ? 0.007f : speed; // Minimum 0.007
}
```

2.4 L'affichage graphique avec SDL

2.4.1 Initialisation de SDL

L'initialisation de SDL configure la fenêtre, le renderer, et les polices nécessaires pour l'affichage du jeu. Cette étape est essentielle pour établir l'environnement graphique. Le code ci-dessous montre les étapes principales, comme la création d'une fenêtre et d'un renderer.

```
// Initialisation de SDL pour la vidéo
if (SDL_Init(SDL_INIT_VIDEO) != 0) {
    fprintf(stderr, "Erreur SDL_Init: %s\n", SDL_GetError());
    return 0;
}

// Création de la fenêtre de jeu
g->win = SDL_CreateWindow("Tetris", x, y, 500, 600, SDL_WINDOW_SHOWN);
if (!g->win) {
    fprintf(stderr, "Erreur SDL_CreateWindow\n");
    return 0;
}

// Création du renderer pour gérer le dessin
g->ren = SDL_CreateRenderer(g->win, -1, SDL_RENDERER_ACCELERATED |
   SDL_RENDERER_PRESENTVSYNC);
if (!g->ren) {
    fprintf(stderr, "Erreur SDL_CreateRenderer\n");
    return 0;
}

// Initialisation des polices pour le texte
g->fontSmall = TTF_OpenFont("Tetris.ttf", 14);
g->fontLarge = TTF_OpenFont("Tetris.ttf", 64);

// Initialisation de l'audio pour les effets sonores
if (Mix_OpenAudio(44100, MIX_DEFAULT_FORMAT, 2, 2048) < 0) {
    fprintf(stderr, "Erreur Mix_OpenAudio\n");
    return 0;
}
```

Dans cet exemple, on remarque l'utilisation des bibliothèques SDL pour la gestion des fenêtres, des polices et des sons. Chaque étape vérifie les erreurs pour s'assurer que le jeu peut continuer à s'exécuter.

2.4.2 Affichage du plateau de jeu

Le plateau est une matrice de 10 colonnes et 20 lignes. Les Tetriminos fixés et en mouvement sont affichés séparément pour garantir une distinction visuelle claire.

Affichage des blocs fixés Les blocs déjà placés dans la matrice principale sont parcourus et affichés en fonction de leur type.

```
// Dessin du cadre du plateau
SDL_Rect game_frame = {0, 0, 300, 600};
SDL_SetRenderDrawColor(g->ren, 255, 0, 255, 255);
SDL_RenderDrawRect(g->ren, &game_frame);

// Dessin de l'arrière-plan des stats
SDL_Rect stats_bg = {300, 0, 200, 600};
SDL_SetRenderDrawColor(g->ren, 30, 30, 30, 255);
SDL_RenderFillRect(g->ren, &stats_bg);

for (int i = 0; i < 20; i++) {
    for (int j = 0; j < 10; j++) {
        if (g->tet->matrix[i][j] != 0) {
            // Affichage d'un bloc (fixé) en fonction de sa position
            mino_display(g, g->tet->matrix[i][j] - 1, i, j);
        }
    }
}
```

Affichage des Tetriminos en mouvement Les Tetriminos en mouvement sont affichés en utilisant leur position actuelle dans le buffer.

```
for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 5; j++) {
        if (g->tet->buffer[i][j] != 0) {
            // Calcul des coordonnées absolues
            int row = g->tet->current_line + i;
            int col = g->tet->current_column + j;
            // Affichage du bloc correspondant
            mino_display(g, g->tet->current_type, row, col);
        }
    }
}
```

2.4.3 Affichage des statistiques

Les statistiques du jeu (score, niveau, lignes détruites) sont affichées sur une section dédiée de la fenêtre, à droite du plateau. Une prévisualisation du prochain Tetrimino est également fournie pour aider le joueur à planifier ses mouvements.

```
// Affichage du score du joueur
snprintf(info, sizeof(info), "Score: %d", g->tet->score);
surface = TTF_RenderText_Solid(g->fontSmall, info, color);
texture = SDL_CreateTextureFromSurface(g->ren, surface);
rect = (SDL_Rect){325, 50, surface->w, surface->h};
SDL_RenderCopy(g->ren, texture, NULL, &rect);
SDL_FreeSurface(surface);
SDL_DestroyTexture(texture);
```

Prévisualisation du prochain Tetrimino La prévisualisation affiche le Tetrimino suivant dans une petite zone à droite.

```
SDL_Rect preview_frame = {325, 200, 150, 150};
SDL_SetRenderDrawColor(g->ren, 255, 0, 255, 255);
SDL_RenderDrawRect(g->ren, &preview_frame);
for (int i = 0; i < 5; i++)
{
    for (int j = 0; j < 5; j++)
    {
        if (tetriminos[g->tet->next_type][0][i][j])
        {
            // Coordonnées du mini-bloc
            SDL_Rect block = {
                preview_frame.x + j * 30,
                preview_frame.y + i * 30,
                27, 27};

            // Récupère la couleur de base du Tetrimino
            Color base = TetriminoColors[g->tet->next_type];
            Color light = {
                base.r + 50 > 255 ? 255 : base.r + 50,
                base.g + 50 > 255 ? 255 : base.g + 50,
                base.b + 50 > 255 ? 255 : base.b + 50};
            Color dark = {
                base.r - 50 < 0 ? 0 : base.r - 50,
                base.g - 50 < 0 ? 0 : base.g - 50,
                base.b - 50 < 0 ? 0 : base.b - 50};

            // Peins le bloc avec la couleur de base
            SDL_SetRenderDrawColor(g->ren, base.r, base.g, base.b, 255)
   ;
            SDL_RenderFillRect(g->ren, &block);
```

```
30
31              // Ajoute la bande claire (en haut)
32              SDL_SetRenderDrawColor(g->ren, light.r, light.g, light.b,
    255);
33              SDL_Rect top_band = {block.x, block.y, (3 * block.w) / 4,
    block.h / 4};
34              SDL_RenderFillRect(g->ren, &top_band);
35
36              // Ajoute la bande claire (à gauche)
37              SDL_Rect left_band = {block.x, block.y, block.w / 4, (3 *
    block.h) / 4};
38              SDL_RenderFillRect(g->ren, &left_band);
39
40              // Ajoute la bande sombre (en bas)
41              SDL_SetRenderDrawColor(g->ren, dark.r, dark.g, dark.b, 255)
    ;
42              SDL_Rect bottom_band = {block.x, block.y + (3 * block.h) /
    4, block.w, block.h / 4 + 1};
43              SDL_RenderFillRect(g->ren, &bottom_band);
44
45              // Ajoute la bande sombre (à droite)
46              SDL_Rect right_band = {block.x + (3 * block.w) / 4, block.y
    , block.w / 4 + 1, block.h};
47              SDL_RenderFillRect(g->ren, &right_band);
48          }
49      }
50  }
```

2.4.4 Gestion des états du jeu

L'affichage change en fonction de l'état du jeu : menu principal, jeu en cours, pause ou fin de partie.

```
case STATE_GAMEOVER:
{
    // Affichage du message "Game Over"
    SDL_Color color1 = {255, 0, 0, 255}; // Rouge
    SDL_Surface *surface = TTF_RenderText_Solid(g->fontLarge, "GAME
   OVER", color1);
    SDL_Texture *texture = SDL_CreateTextureFromSurface(g->ren, surface
   );
    SDL_Rect dstRect;
    dstRect.x = (500 - surface->w) / 2;
    dstRect.y = (600 - surface->h) / 4;
    dstRect.w = surface->w;
    dstRect.h = surface->h;
    SDL_RenderCopy(g->ren, texture, NULL, &dstRect);
    SDL_FreeSurface(surface);
    SDL_DestroyTexture(texture);
    /*.................*/
}
```

2.4.5 Boucle de rendu

Exemple : Écran "Game Over" La boucle de rendu rafraîchit l'écran pour refléter les changements dans l'état du jeu.

```
void game_update(Game *g) {
    SDL_SetRenderDrawColor(g->ren, 24, 24, 24, 255);
    SDL_RenderClear(g->ren);
    switch (g->state) {
        case STATE_MENU:
            // Affichage du menu principal
            break;
        case STATE_PLAY:
            // Dessin du plateau et des statistiques
            break;
        case STATE_GAMEOVER:
            // Affichage de l'écran "Game Over"
            break;
        case STATE_PAUSE:
            // Affichage de l'écran de pause
            break;
    }
    SDL_RenderPresent(g->ren);
}
```

2.5 La gestion des entrées utilisateur

2.5.1 Principe général

Dans Tetris, les entrées utilisateur permettent d'interagir avec le jeu en déplaçant les Tetriminos, en les faisant pivoter, ou en contrôlant la vitesse de leur descente. Ces entrées sont gérées via les événements détectés par SDL dans une boucle d'événements.

Les actions principales gérées par les entrées utilisateur sont :

— **Déplacement gauche/droite** : Déplacer le Tetrimino horizontalement.
— **Descente rapide** : Accélérer la descente du Tetrimino.
— **Rotation** : Pivoter le Tetrimino.
— **Pause ou Quitter le jeu** : Gérer des états comme la pause ou l'arrêt du jeu.

2.5.2 Gestion des événements SDL

SDL capture les événements utilisateur (comme les appuis sur des touches) via la fonction `SDL_PollEvent`. Chaque événement est ensuite analysé pour déterminer quelle action doit être effectuée.

```
SDL_Event e;
while (SDL_PollEvent(&e)) {
    // Si l'utilisateur ferme la fenêtre
    if (e.type == SDL_QUIT) {
        running = 0; // Arrêter la boucle du jeu
    }
    // Si une touche est appuyée
    else if (e.type == SDL_KEYDOWN) {
        switch (e.key.keysym.sym) {
            case SDLK_LEFT: // Flèche gauche
                tetris_move_left(g->tet);
                break;
            case SDLK_RIGHT: // Flèche droite
                tetris_move_right(g->tet);
                break;
            case SDLK_DOWN: // Flèche bas
                tetris_move_down(g->tet);
                break;
            case SDLK_UP: // Flèche haut
                tetris_rotate(g->tet);
                break;
            case SDLK_p: // Pause
                g->state = (g->state == STATE_PLAY) ? STATE_PAUSE :
   STATE_PLAY;
                break;
            case SDLK_ESCAPE: // Quitter
                running = 0;
                break;
            default:
```

```
                    break;
            }
        }
}
```

2.5.3 Implémentation des actions

Exemple : Gestion des touches Chaque action correspond à une fonction spécifique :

- **Déplacement à gauche/droite** : `tetris_move_left` et `tetris_move_right` mettent à jour la position du Tetrimino.
- **Rotation** : `tetris_rotate` applique une rotation si elle est valide.
- **Descente rapide** : `tetris_move_down` accélère la descente du Tetrimino en mettant à jour sa position plus fréquemment.

```
case SDLK_UP: // Flèche haut
    if (tetris_can_rotate(g->tet)) { // Vérifie si la rotation est
   possible
        tetris_rotate(g->tet);       // Effectue la rotation
    }
    break;
```

2.5.4 Gestion des états (Pause et Quitter)

Exemple : Rotation avec `SDL_KEYDOWN` Des touches spécifiques permettent de gérer l'état global du jeu :

- **Pause** : La touche `p` bascule entre les états `STATE_PLAY` et `STATE_PAUSE`.
- **Quitter** : La touche `ESC` arrête le jeu en mettant fin à la boucle principale.

```
case SDLK_p: // Pause
    if (g->state == STATE_PLAY) {
        g->state = STATE_PAUSE;      // Mettre en pause
        Mix_HaltMusic();             // Arrêter la musique de fond
    } else if (g->state == STATE_PAUSE) {
        g->state = STATE_PLAY;       // Reprendre le jeu
        Mix_PlayMusic(g->music, -1); // Relancer la musique
    }
    break;
```

2.5.5 Synchronisation avec la boucle principale

Les entrées utilisateur sont synchronisées avec la boucle principale du jeu. Chaque action est appliquée immédiatement et l'écran est mis à jour en conséquence via la fonction `game_update`.

```
while (running) {
    // Gestion des événements utilisateur

    // Mise à jour de l'état du jeu
    if (g->state == STATE_PLAY) {
        game_update(g);
    }
}
```

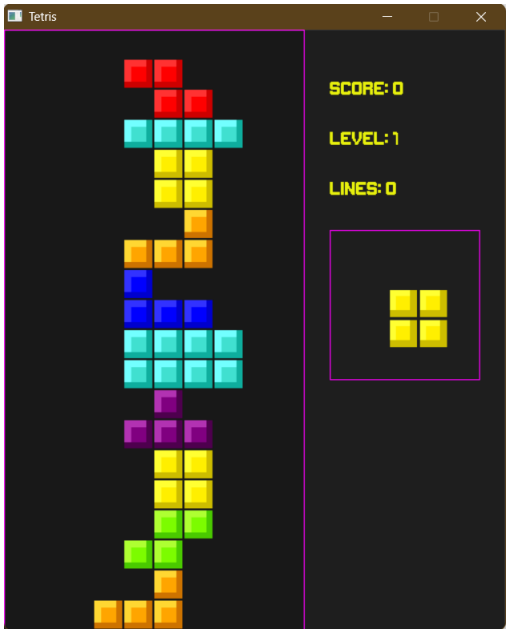
2.6 Fin de partie

Exemple : Synchronisation dans la boucle

2.6.1 Détection de la fin de partie

La fin de partie dans Tetris est détectée lorsqu'un Tetrimino ne peut pas être placé en haut de la matrice, car la première ligne est déjà occupée par des blocs fixés. Cela se produit lorsque le joueur n'a pas réussi à éliminer suffisamment de lignes.

Exemple : Matrice bloquée Voici un exemple où un nouveau Tetrimino ne peut pas être placé, déclenchant ainsi la fin de la partie :



2.6.2 Implémentation

La vérification de la fin de partie est réalisée par la fonction suivante, S'il ne peut pas descendre ET qu'il est tout en haut dans la fonction tetris move down :

```
if (tet->current_line == 0)
{
    return -1;
}
```

2.6.3 Affichage de l'écran "Game Over"

Lorsqu'une fin de partie est détectée, un écran dédié s'affiche avec le message "GAME OVER", des instructions pour rejouer ou quitter, ainsi que les statistiques finales du joueur.

Statistiques finales affichées :

- **Score final** : Affiche le score obtenu par le joueur.
- **Niveau atteint** : Indique le niveau final du joueur.
- **Lignes détruites** : Montre le nombre total de lignes supprimées.

2.6.4 Relancer ou quitter la partie

Après la fin de la partie, le joueur peut choisir de relancer une nouvelle partie ou de quitter le jeu. Ces actions sont gérées via des entrées clavier :

- Appuyer sur `ESPACE` ou `ENTRÉE` pour relancer la partie.
- Appuyer sur `ESC` pour quitter le jeu.

```
if (g->state == STATE_GAMEOVER) {
    switch (e.key.keysym.sym) {
        case SDLK_SPACE:
        case SDLK_RETURN:
            memset(g->tet->matrix, 0, sizeof(g->tet->matrix));
            g->tet->score = 0;
            g->tet->nbr_lines = 0;
            tetris_reset(g->tet);
            g->state = STATE_PLAY; // Revenir à l'état de jeu
            break;
        case SDLK_ESCAPE:
            running = 0; // Quitter le jeu
            break;
    }
}
```

Chapitre 3

Problèmes rencontrés

3.1 Problèmes rencontrés et solutions

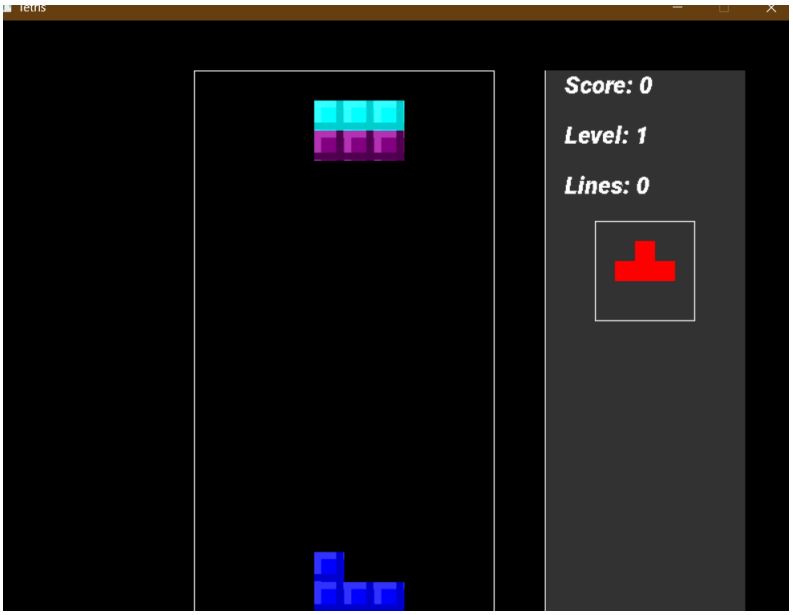
3.1.1 Absence de chiffres dans la police utilisée

Problème : Au début, la police utilisée pour afficher les textes (`Tetris.ttf`) ne contenait pas de chiffres. Cela empêchait l'affichage des informations essentielles, comme le score, le niveau, et le nombre de lignes détruites.

Solution : Une nouvelle police a été téléchargée et intégrée dans le projet, permettant d'afficher correctement toutes les informations. Les modifications ont porté sur le fichier où la police était initialisée.

3.1.2 Blocs stagnants en haut du plateau

Problème : Certains Tetriminos apparaissaient en haut du plateau mais restaient immobiles, sans descendre comme prévu. Cela bloquait le déroulement normal du jeu.



Solution : Ce problème était lié à un bug dans la logique de descente des Tetriminos. En corrigeant les appels à la fonction de descente et en vérifiant les conditions de collision dans la matrice, les Tetriminos ont repris leur mouvement dès leur apparition.

3.1.3 Segmentation fault imprévisible

Problème : Une erreur de segmentation survenait aléatoirement pendant l'exécution, mais elle semblait se résoudre d'elle-même après certaines modifications dans le code.

Solution : Bien que ce problème n'ait pas été réglé explicitement, la cause était probablement liée à des indices de matrice invalides. Après avoir corrigé d'autres parties du code, le problème a disparu.

3.1.4 Game Over non détecté

Problème : Lorsque les blocs atteignaient le sommet du plateau, le jeu continuait à générer de nouveaux Tetriminos au lieu de détecter la fin de partie. Cela rendait le jeu infini et bloquait l'utilisateur.

```
int tetris_move_down(Tetris *tet)
{
    // Peut-on descendre ?
    if (tetris_can_go_down(tet))
    {
        tet->current_line++;
        return 1; // Succès : on a bougé
    }
    else
    {
        // S'il ne peut pas descendre ET qu'il est tout en haut
        // => Game Over (on renvoie un code spécial, disons -1)
        if (tet->current_line == 0)
        {
            // La pièce n'a jamais pu bouger => bloqué dès le spawn
            return -1;
        }

        // Sinon, c'est juste la fin de la descente (on fixe la pièce)
        tetris_shift_board(tet);
        tetris_reset(tet);

        return 0; // "J'ai fini, mais pas de Game Over"
    }
}
```

Solution : la fonction tetris move down a été corrige pour vérifier s'il ne peut pas descendre et qu'il est tout en haut. l'état du jeu passe à `STATE_GAMEOVER`.

Chapitre 4

Choix techniques et justifications

4.1 Utilisation d'un buffer 5×5 au lieu de 20×10

Problème et contexte : Dans les spécifications du projet, il était mentionné d'utiliser une matrice de buffer de 20×10 pour gérer les déplacements et les collisions des Tetriminos. Cependant, dans ce projet, un buffer de 5×5 a été utilisé pour les Tetriminos en mouvement. Ce choix diffère de la spécification originale, mais il s'avère plus adapté pour certains aspects de la gestion des Tetriminos.

4.1.1 Avantages d'un buffer 5×5

- **Simplicité de gestion :** Un buffer de 5×5 correspond exactement à la taille maximale d'un Tetrimino, peu importe sa rotation. Cela facilite les calculs et les vérifications.
- **Optimisation mémoire :** Le buffer de 5×5 utilise beaucoup moins de mémoire que celui de 20×10. Comme il ne contient que le Tetrimino en mouvement, il n'a pas besoin de refléter toute la matrice de jeu.
- **Clarté du code :** Les fonctions comme `tetris_can_go_down` ou `tetris_can_rotate` n'ont besoin de parcourir que $5 \times 5 = 25$ cases, au lieu de $20 \times 10 = 200$ cases.

4.1.2 Implémentation

Le buffer 5×5 est utilisé pour stocker le Tetrimino en cours de mouvement. Les calculs de collisions et de positions sont effectués en utilisant ce buffer.

```
typedef struct {
    char matrix[20][10]; // Matrice des blocs fixés
    char buffer[5][5];   // Buffer pour le Tetrimino en mouvement
    ...
} Tetris;
```

Utilisation dans les fonctions : Les fonctions comme `tetris_can_go_down` et `tetris_rotate` utilisent le buffer 5×5 pour gérer les déplacements et les rotations.

```
// Exemple : Vérification de la descente
int tetris_can_go_down(Tetris *tet) {
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            if (tet->buffer[i][j]) { // Case occupée
                int row = tet->current_line + i;
                int col = tet->current_column + j;
                // Vérification des limites et collisions
                if (row == 19 || tet->matrix[row + 1][col] != 0) {
                    return 0; // Descente impossible
                }
            }
        }
    }
    return 1; // Descente possible
}
```

4.1.3 Limites de ce choix

— **Calculs supplémentaires :** Les coordonnées globales du Tetrimino doivent être recalculées à chaque vérification, car le buffer 5×5 n'est pas aligné avec la matrice 20×10.

4.2 Affichage des Minos et Raffinements esthétiques

4.2.1 Choix des dimensions des bandes de relief

Dans notre projet, la fonction `mino_display` est responsable de l'affichage des Minos avec un effet de relief, obtenu grâce à l'utilisation de bandes claire et sombre. Les spécifications initiales du projet proposaient des bandes très minces, reflétant un rendu discret (comme indiqué dans le sujet). Cependant, nous avons choisi de privilégier des reliefs plus épais pour donner un effet visuel plus immersif et captivant.

Deux configurations principales ont été testées :

— **Proportions `(7 * size) / 8` et `size / 8` :** Ce rendu donne un effet élégant et équilibré, avec des reliefs minces qui rappellent une approche minimaliste.

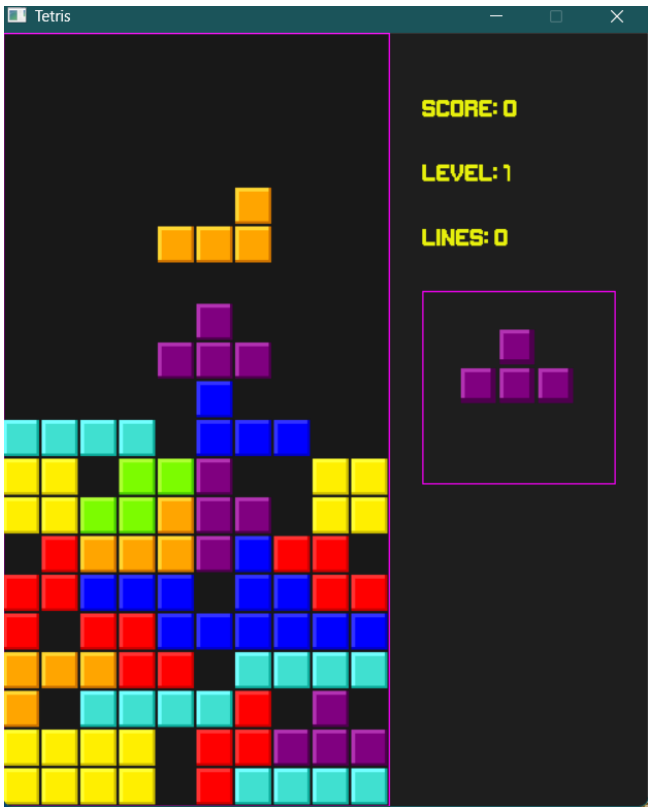
— **Proportions `(3 * size) / 4` et `size / 4` :** Ce choix, finalement adopté, offre des reliefs plus épais, évoquant des bonbons translucides ou des émeraudes. Nous avons préféré cette approche car elle apporte une touche distinctive et accrocheuse au design.

4.2.2 Ajustements des proportions et impact visuel

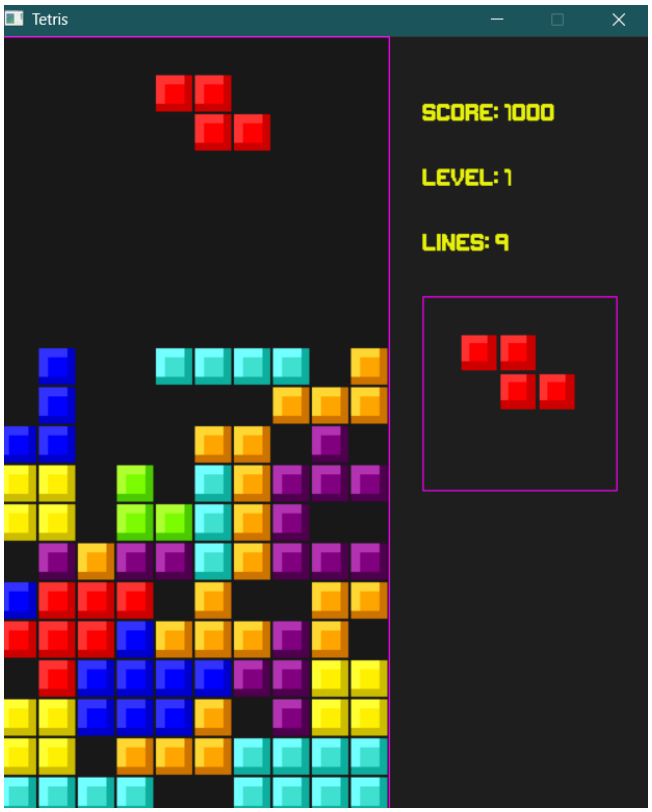
Voici le code de la fonction `mino_display` :

```
void mino_display(Game *g, Type t, int l, int c)
{   /*.............*/
    // Couleur de base (plein du mino)
    SDL_SetRenderDrawColor(g->ren, base.r, base.g, base.b, 0xFF);
    rect = (SDL_Rect){x, y, (3 * size) / 4, (3 * size) / 4};
    SDL_RenderFillRect(g->ren, &rect);

    // Ligne supérieure (claire)
    SDL_SetRenderDrawColor(g->ren, light.r, light.g, light.b, 0xFF);
    rect = (SDL_Rect){x, y, (3 * size) / 4, size / 4};
    SDL_RenderFillRect(g->ren, &rect);
    // Colonne gauche (claire)

    rect = (SDL_Rect){x, y, size / 4, (3 * size) / 4};
    SDL_RenderFillRect(g->ren, &rect);
    // Ligne inférieure (sombre)

    SDL_SetRenderDrawColor(g->ren, dark.r, dark.g, dark.b, 0xFF);
    rect = (SDL_Rect){x, y + (3 * size) / 4, size - 2, size / 4 - 1};
    SDL_RenderFillRect(g->ren, &rect);
    // Colonne droite (sombre)

    rect = (SDL_Rect){x + (3 * size) / 4, y, size / 4 - 1, size - 2};
    SDL_RenderFillRect(g->ren, &rect);
}
```

4.2.3 Comparaison visuelle des rendus



Rendu initial avec proportions `(7 * size) / 8, size / 8`.

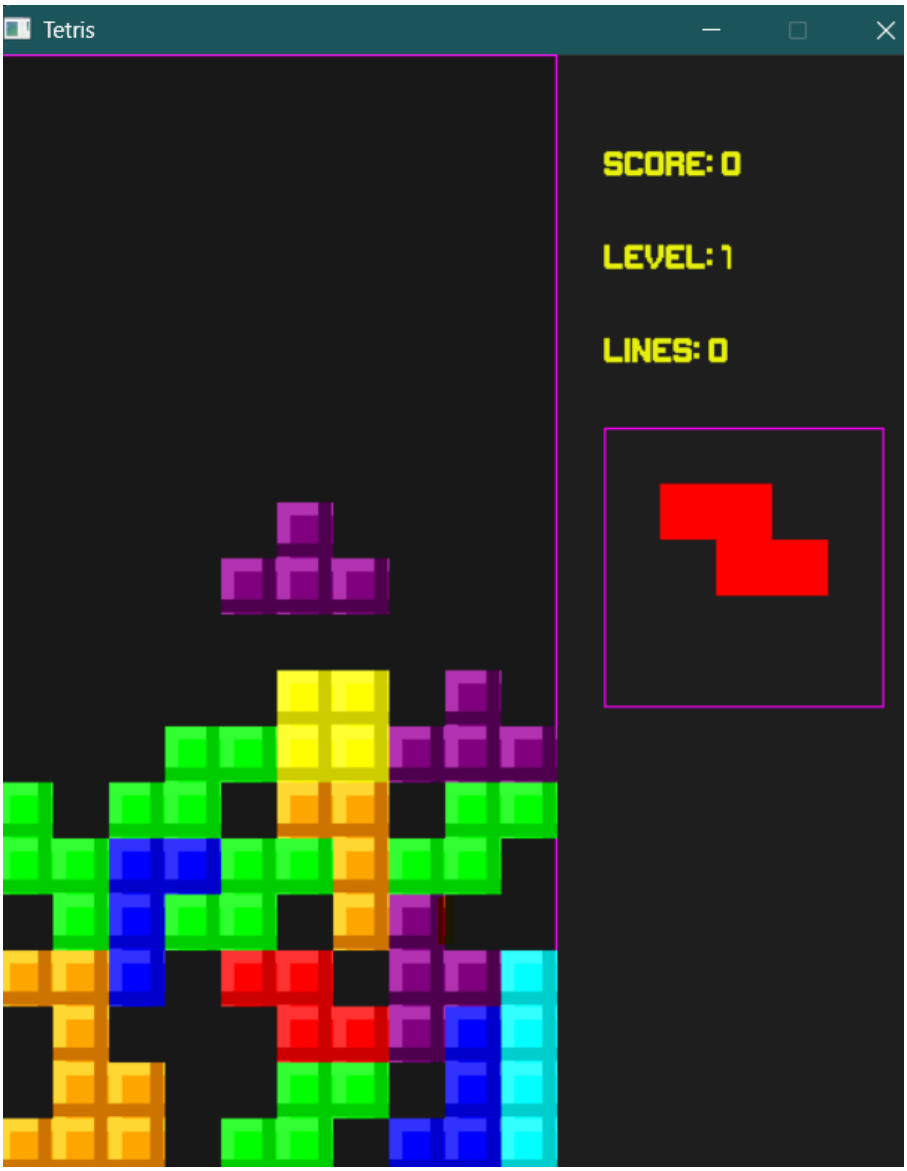


Rendu final avec proportions `(3 * size) / 4, size / 4`.

4.2.4 Ajustements spécifiques des bandes sombre

Pour améliorer encore le rendu visuel :

Nous avons réduit légèrement les dimensions des bandes sombre en appliquant des décalages (`-1` et `-2`) sur leurs coordonnées. Ces ajustements subtils permettent de créer une impression de profondeur accrue sans chevauchement visuel.



Rendu déséquilibré inspiré de l'exemple mal ajusté.

Chapitre 5

Résultats

5.1 Présentation du jeu final

Le projet Tetris a été développé en utilisant la bibliothèque SDL pour gérer les graphismes et les entrées utilisateur. Voici une présentation des différentes étapes et fonctionnalités du jeu final :

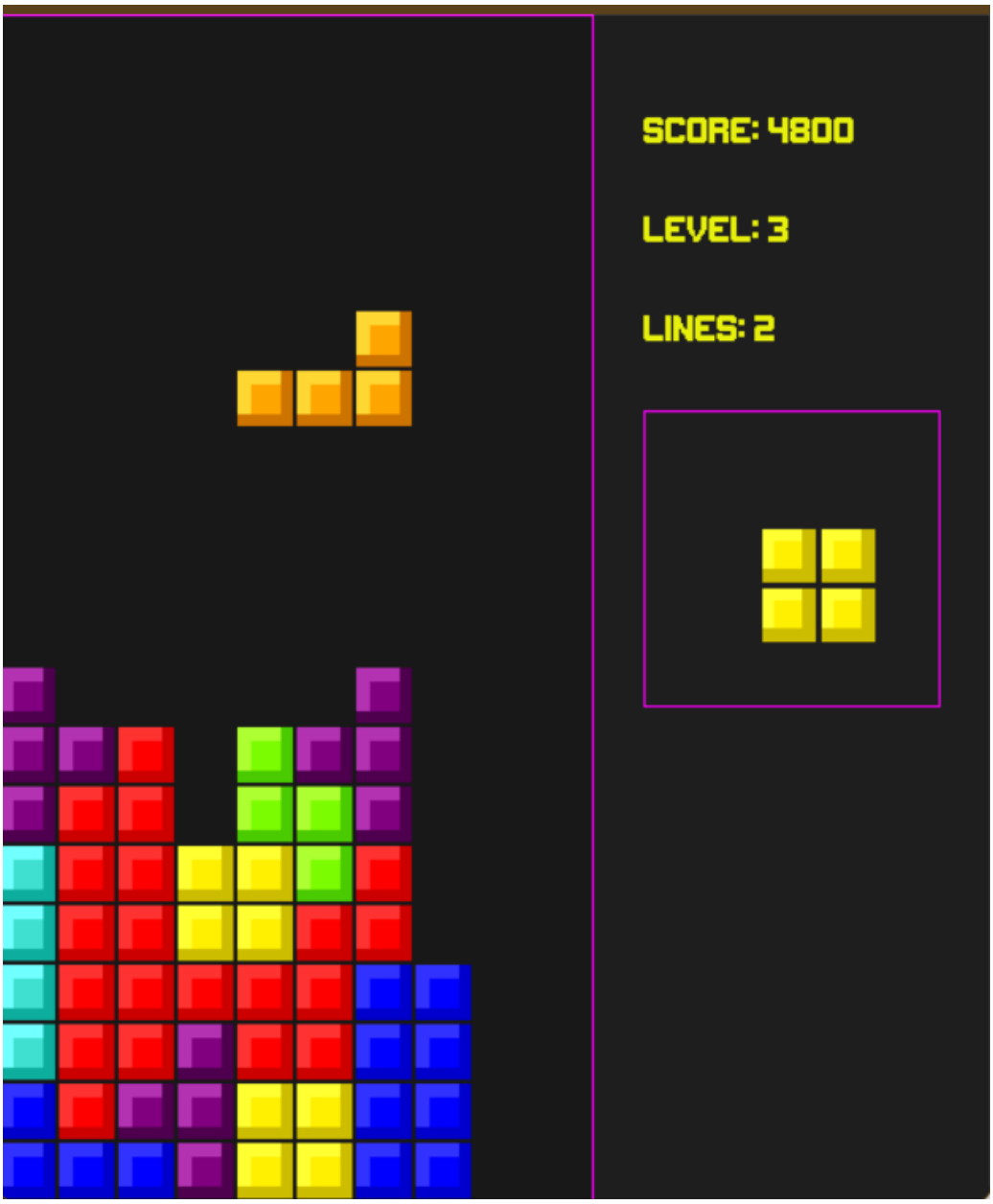
5.1.1 Écran de menu

Lorsque le jeu est lancé, le joueur est accueilli par un écran de menu. Celui-ci affiche le titre du jeu, des instructions, et une option pour commencer la partie en appuyant sur la touche `ESPACE` ou `ENTRÉE`.

Écran de menu du jeu Tetris

5.1.2 En jeu

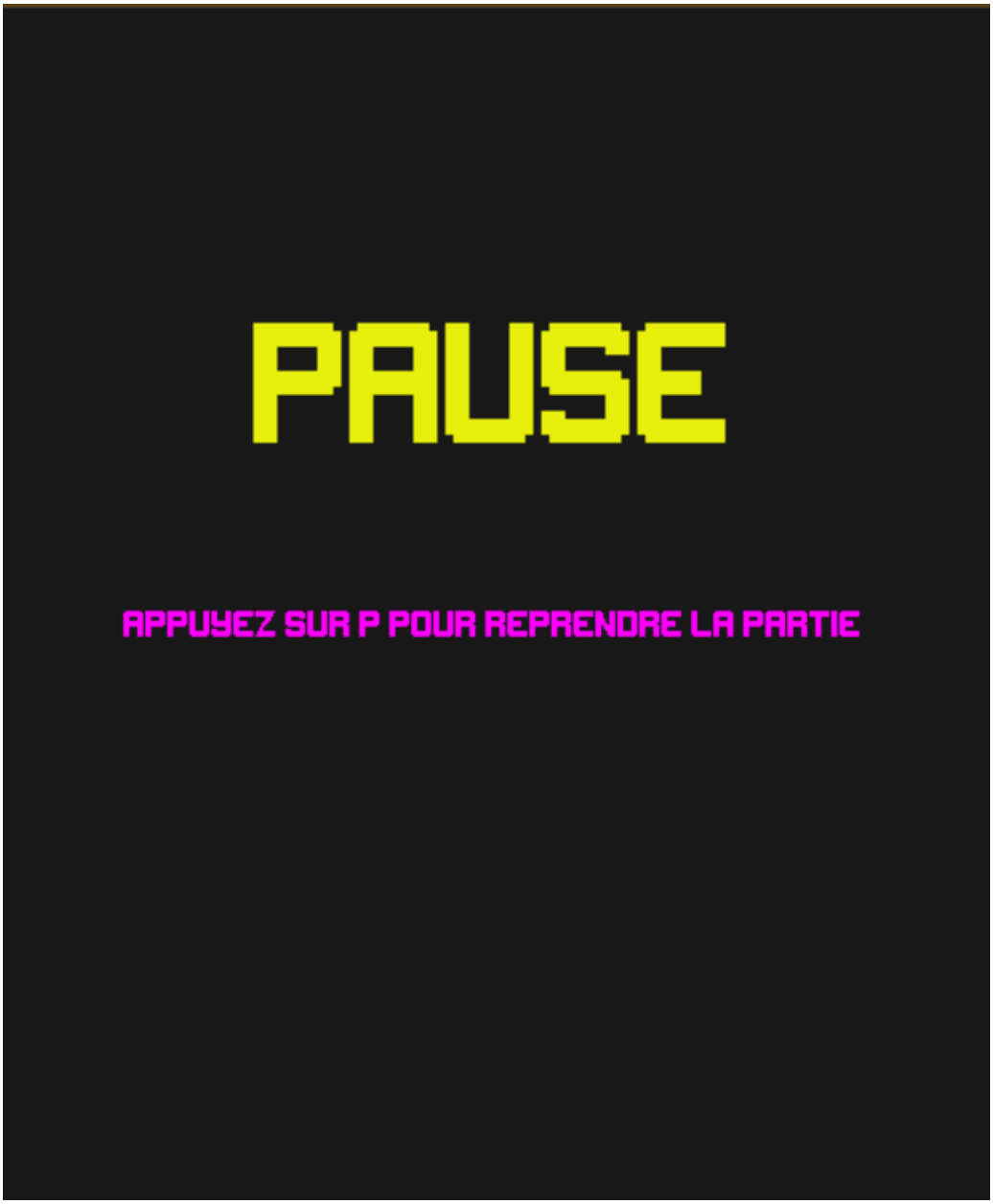
Pendant la partie, le joueur peut contrôler les Tetriminos à l'aide des flèches directionnelles pour les déplacer ou les faire pivoter. Le score, le niveau, et le nombre de lignes détruites sont affichés sur le côté droit de l'écran. Le prochain Tetrimino est également prévisualisé pour aider le joueur à planifier ses mouvements.



Écran principal en cours de partie

5.1.3 Pause

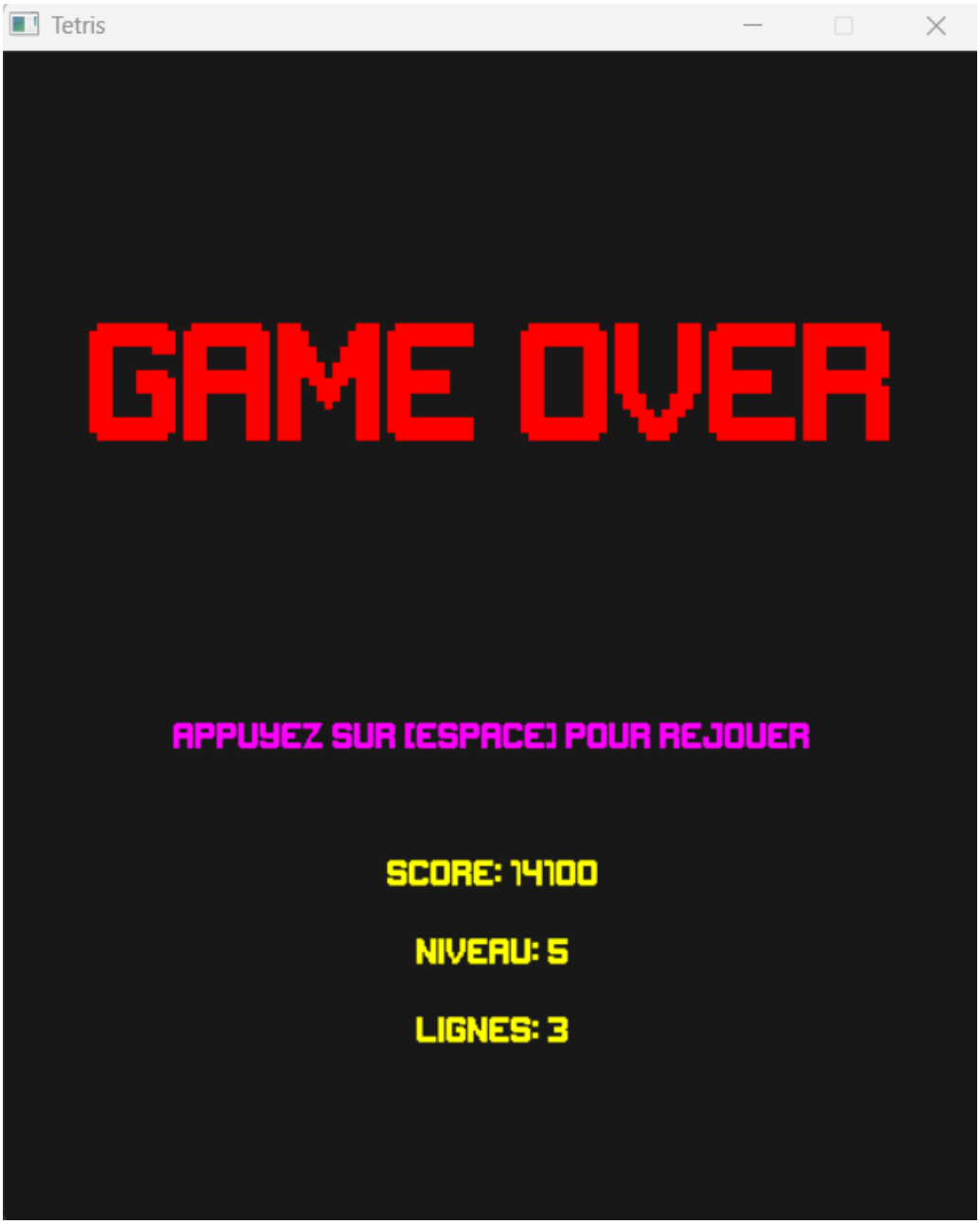
Le joueur peut mettre la partie en pause à tout moment en appuyant sur la touche `P`. Un écran de pause s'affiche alors avec des instructions pour reprendre la partie.



Écran de pause du jeu Tetris

5.1.4 Fin de partie

Lorsque les blocs atteignent le sommet du plateau, la partie est terminée. Un écran de Game Over s'affiche, indiquant le score final, le niveau atteint, et le nombre de lignes détruites.



Écran de fin de partie (Game Over)

5.2 Limites et aspects à améliorer

Bien que le jeu soit fonctionnel et respecte les exigences initiales du projet, certaines limites et améliorations potentielles ont été identifiées :

5.2.1 Limites actuelles

- **Manque de modes de jeu :** Le jeu se limite à un mode solo classique sans variation.
- **Graphismes simples :** Bien que fonctionnels, les graphismes pourraient être améliorés avec des animations ou des effets visuels.
- **Absence de sauvegarde :** Le jeu ne propose pas de sauvegarde du score ou d'historique des parties.
- **Gestion sonore basique :** Les sons sont simples et pourraient être enrichis par des effets plus variés ou une bande-son personnalisée.

5.2.2 Propositions d'amélioration

- **Ajout de modes de jeu :** Implémenter des modes supplémentaires comme un mode contre-la-montre ou un mode multijoueur local.
- **Animations visuelles :** Ajouter des animations pour la rotation des Tetriminos.
- **Niveaux de difficulté :** Proposer des options pour ajuster la difficulté, comme un mode débutant ou expert.
- **Sauvegarde des scores :** Créer un tableau des meilleurs scores pour encourager la rejouabilité.

Chapitre 6

Conclusion

Le développement de ce projet Tetris a permis de mettre en pratique des compétences clés en programmation C, en gestion de projets logiciels, et en utilisation de la bibliothèque SDL pour les graphismes et les entrées utilisateur. À travers les différentes étapes de conception et d'implémentation, de nombreux défis ont été relevés, allant de la gestion des collisions à l'affichage graphique, en passant par la détection des états de jeu comme le Game Over.

Le résultat final est un jeu fonctionnel respectant les règles classiques de Tetris. Il inclut des fonctionnalités telles que le calcul du score, l'affichage du prochain Tetrimino, et la progression des niveaux. Ces éléments offrent une expérience utilisateur fluide et fidèle au concept original.

Cependant, certaines limitations subsistent, comme l'absence de modes de jeu alternatifs, des graphismes simples, et l'absence de fonctionnalités de sauvegarde ou de personnalisation. Ces aspects pourraient être développés dans le futur pour enrichir l'expérience utilisateur et améliorer la qualité globale du projet.

Ce projet a non seulement permis de créer un jeu, mais il a également constitué une excellente opportunité d'apprentissage et de perfectionnement. Il a renforcé les compétences en algorithmique, en gestion mémoire, et en programmation événementielle, tout en offrant une perspective concrète sur la création d'un jeu vidéo. Avec des améliorations futures, ce projet pourrait devenir une version encore plus aboutie et moderne du Tetris classique.

Chapitre 7

Références

Font Tetris :
https ://www.dafont.com/fr/gameplay.font ?text=TETRIS

Musique de fond :
https ://www.youtube.com/watch ?v=NmCCQxVBfyM

Musique pour l'état Pause :
https ://www.youtube.com/watch ?v=9TPazxfqVLk

Son joué lors du Game Over :
1er son dans la vidéo : https ://www.youtube.com/watch ?v=bug1b0fQS8Y

Son joué lorsque des lignes sont détruites :
4ème son dans la vidéo : https ://www.youtube.com/watch ?v=VhHJomZDpbA

Son joué lorsqu'on passe un niveau :
https ://www.youtube.com/watch ?v=qvBteOqddqk

Son joué lorsque le Tetrimino atterrit :
Fait par nous en frappant deux bouteilles en verre.