Projet Sokoban

Programmation Avancée - L2 Informatique

Année 2024-2025

Présenté par :
MOTA TITOUAN
TRECHOT ALEXIS

Université Clermont-Auvergne

L2 Informatique

15 janvier 2025

Table des matières

Chapitre 1

Introduction

1.1 Contexte du projet

Ce projet s'inscrit dans le cadre de notre deuxième année de licence en informatique. Il a pour but de clore le chapitre consacré au langage C, entamé dès la première année.

L'objectif de ce travail est d'implémenter un jeu existant (Sokoban), en langage C, afin de mettre en pratique les compétences acquises tout au long de ces deux années.

Le projet est réalisé en duo, avec une période de travail allant du 11 décembre 2024 au 15 janvier 2025.

1.2 Présentation du jeu Sokoban

Selon Wikipédia, "le joueur doit ranger des caisses sur des cases cibles. Il peut se déplacer dans les quatre directions, et pousser (mais pas tirer) une seule caisse à la fois. Une fois toutes les caisses rangées (c'est parfois un vrai casse-tête), le niveau est réussi et le joueur passe au niveau suivant, plus difficile en général. L'idéal est de réussir avec le moins de coups possibles (déplacements et poussées)." [1]

1.3 Environnement de développement

Le projet sera développé principalement sur Ubuntu 24.04 LTS (VM) avec l'éditeur de code `VSCode`, en utilisant les extensions `C/C++`, `CMake Tools` et `C/C++ Theme`. Les librairies `SDL2` (2.30.0) et `SDL_ttf` seront installées. Le jeu sera compilé avec `GCC` (Ubuntu 13.3.0-6ubuntu2 24.04) et `GNU Make` 4.3. La gestion de version sera assurée par `Git`, avec un dépôt hébergé sur `GitHub`.

Chapitre 2

Analyse et Conception

2.1 Analyse des besoins

Le jeu doit être réalisé en respectant certaines consignes telles que :
— Chargement et affichage d'un plateau de jeu.
— Positionnement et gestion du personnage.
— Positionnement des points à couvrir.
— Positionnement des caisses et leur déplacement.
— Chargement d'un labyrinthe depuis un fichier texte, à la manière de la description faite dans le TP5, où le caractère # représente les murs, le caractère P représente le personnage, le caractère C représente une caisse et le caractère I représente un point d'intérêt.
— Affichage d'un score, qui pourra être calculé en fonction du temps passé, du nombre de mouvements du personnage et du nombre de caisses bien placées.
— Affichage d'un chronomètre.
— Gestion de niveaux et de difficultés.
— Détection de la condition de partie perdue (c'est-à-dire lorsque le joueur se retrouve dans une situation où il n'est plus possible de gagner : par exemple, une caisse est collée à un mur et ne pourra jamais être poussée vers une cible).
— Stratégie de résolution.
— Conception des niveaux pour garantir la faisabilité d'un plateau (plateau + positionnement des cibles + positionnement des caisses).

À ce stade, nous pouvons déjà identifier plusieurs éléments du jeu, notamment différentes structures de données telles que la carte, le joueur, les coordonnées, ainsi que toutes les fonctions nécessaires au jeu.

2.2 Architecture générale

Le programme est organisé en quatre parties distinctes. Tout d'abord, les fichiers `affichage.h` et `jeu.h` servent à la déclaration des fonctions, tandis que les fichiers `affichage.c` et `jeu.c` contiennent l'implémentation des fonctions déclarées dans les fichiers d'en-tête. Ensuite, le fichier `main.c` orchestre l'appel de toutes les fonctions présentes dans les fichiers annexes pour rendre le jeu jouable. Enfin, le fichier `Makefile` facilite la compilation du projet.

Cette organisation permet de diviser le code en sections distinctes, ce qui le rend à

la fois plus lisible et mieux structuré. Chaque fichier a un rôle spécifique, ce qui facilite la gestion. Cette approche améliore la compréhension du code et permet de résoudre les bugs plus facile.

2.3 Structures de données

Le jeu utilise les deux structures suivantes :

Listing 2.1 – Structures de données

```
typedef struct Coord{
        int x;
        int y;
}Coord;

typedef struct structMap{
        char **map;
        int colonnes;
        int lignes;
        Coord *player;
        Coord *WinCaissePos;
        int nbCaisses;
}Map;
```

La première structure `Coord` permet de manipuler facilement des coordonnées à l'aide de deux variables de type `int`, `x` et `y`.

La deuxième structure `Map` permet de stocker les éléments de la carte, tels qu'un tableau 2D de caractères (`map`) qui représente la carte, un entier (`colonnes`) qui représente le nombre de colonnes, un entier (`lignes`) qui représente le nombre de lignes, un pointeur de type `Coord` (`player`) qui permet de stocker les coordonnées du joueur, et un tableau de structures `Coord` (`WinCaissePos`) pour les positions des caisses. Le nombre total de caisses est stocké dans l'entier `nbCaisses`.

2.4 Algorithmes principaux

Le jeu est structuré autour d'une fonction `main` qui détermine la fonction à appeler en fonction des choix de l'utilisateur. Par exemple, le code suivant montre le processus principal du jeu :

- `choix == 0` : Quitter le jeu. Cette option met fin à la boucle principale et arrête l'exécution du programme.
- `choix == 2` : Lancer une partie. Cette option initialise la partie en appelant la fonction `playGameSDL()` pour démarrer le jeu.
- `choix == 3` : Retour à l'écran d'accueil. Cette option redirige l'utilisateur vers l'écran principal où il peut effectuer un nouveau choix.
- `choix == 4` : Sélectionner un niveau. Cette option permet à l'utilisateur de choisir un niveau spécifique à jouer.

Listing 2.2 – Boucle principale du jeu

```
while(running) {
        if(choix == 0){
                running = 0;
        }else if (choix == 2){
                SDL_RenderClear(renderer);
                choix = playGameSDL(renderer, &currentLevel);
        }else if (choix == 3){
                SDL_RenderClear(renderer);
                choix = accueil(renderer);
        }else if (choix == 4){
                SDL_RenderClear(renderer);
                choix = selectLevel(renderer, &currentLevel);
        }
}
```

Dans ce système, chaque fonction est appelée selon la variable `choix`. Cette approche a été privilégiée car elle est claire et permet un contrôle facile du déroulement du jeu sans la nécessité d'exécuter plusieurs instances de la même fonction simultanément. Elle assure également une gestion efficace des transitions entre les différents écrans du jeu que nous détaillerons par la suite.

Chapitre 3

Implémentation

3.1 Fonctions du jeu

3.1.1 Représentation du plateau

Les différents niveaux de Sokoban sont stockés dans des fichiers situés à l'emplacement suivant : `sokoban/maps/mapY.txt`, où `Y` représente le niveau souhaité.

Chaque fichier texte contient différents symboles permettant de définir les éléments du jeu :

— `P` : joueur
— `I` : position de victoire d'une caisse
— `C` : caisse
— `#` : murs

Par exemple, pour le niveau 1, cela donne :

```
##################################
#  P          C               I  #
#  C                          I  #
##################################
```

Pour initialiser la carte, nous utilisons la fonction suivante :

Listing 3.1 – Initialisation de la carte

```
Map * initMap(int niveau);
```

Cette fonction commence par récupérer le fichier de la carte correspondant au niveau donné en argument. Le niveau est transformé en chaîne de caractères pour construire le chemin du fichier à l'aide du code suivant :

```
char nomMap[20];
sprintf(nomMap, "maps/map%d.txt", niveau);
```

Une première lecture du fichier est effectuée pour déterminer le nombre de lignes et la longueur de la ligne la plus longue, qui définira le nombre de colonnes.

Ensuite, une matrice 2D est créée pour représenter la carte, ainsi que des variables de type `Coord` pour stocker les coordonnées du joueur (`player`) et un tableau de structures `Coord` pour les positions des `I`.

Le programme copie ensuite le contenu du fichier texte dans la matrice 2D, en évitant de retourner à la ligne lorsque le caractère lu est un `\n`, mais uniquement lorsqu'il atteint

la largeur maximale. Si le programme rencontre un `I`, il l'ajoute à la liste ; et s'il rencontre un `P`, il initialise les coordonnées de départ du joueur, comme illustré ci-dessous :

Listing 3.2 – Lecture et initialisation de la carte

```
while((current = fgetc(fd)) != EOF){
        if(current == '\n') {
                continue;
        }
        if(current == 'P'){
                coordJoueur->x = i;
                coordJoueur->y = j;
        }
        if(current == 'I'){
                tabCoord[count].x = i;
                tabCoord[count].y = j;
                count += 1;
        }
        tab[i][j] = current;
        j++;
        if(j >= MaxColonnes){
                i++;
                j = 0;
        }
        if(i >= lignes){
                break;
        }
}
```

La carte est ensuite créée, assignée avec les valeurs précédemment trouvées, puis renvoyée.

Lorsque le jeu est terminé, nous appelons la fonction suivante :

Listing 3.3 – Libération de l'espace mémoire de la carte

```
void libereEspaceDeLaMap(Map *m);
```

Comme son nom l'indique, cette fonction libère tout l'espace mémoire occupé par la variable `map`.

3.1.2 Gestion des déplacements

Les déplacements sont gérés par les deux fonctions suivantes :

Listing 3.4 – Fonctions principales pour les déplacements

```
int deplaceJoueur(int direction, Map *map);
int deplacementValide(Coord *coord, Map *map);
```

La fonction `deplacementValide()` prend en paramètre les coordonnées de la case souhaitée lors du déplacement et la carte. Elle vérifie si cette case est libre et renvoie des valeurs dans trois situations particulières :

1. La prochaine case est vide ou une cible (`' '` ou `'I'`), alors le déplacement est valide.
2. La case souhaitée est un mur (`#`), le déplacement est invalide.
3. La case contient une caisse (`'C'`), il faudra ajouter une condition supplémentaire dans `deplaceJoueur()`.

Implémentation de `deplacementValide()` :

Listing 3.5 – Vérification de la validité d'un déplacement

```
int deplacementValide(Coord *coord, Map *map){
        char next_pose = map->map[coord->x][coord->y];

        if(next_pose == ' ' || next_pose == 'I'){
                        return 1;
        }else if (next_pose == '\#'){
                        return 0;
        }else if (next_pose == 'C'){
                return 2;
        }
        return 3;
}
```

La fonction `deplaceJoueur()` effectue le déplacement. Elle vérifie dans un premier temps si le déplacement est valide à l'aide de `deplacementValide()`. Si le déplacement est valide :

— Le joueur se déplace sur la case souhaitée.
— Si la case contient une caisse, le joueur et la caisse se déplacent à condition que la case suivante dans cette direction soit libre.

Voici une implémentation simplifiée de `deplaceJoueur()` :

Listing 3.6 – Gestion des déplacements du joueur

```
    if(direction == 0){
        next_coord->x = (map->player->x -1);
        next_coord->y = (map->player->y);
        next_coord_caisse->x = (map->player->x -2);
        next_coord_caisse->y = (map->player->y);
}else if(direction == 1){
        next_coord->x = (map->player->x +1);
        next_coord->y = (map->player->y);
        next_coord_caisse->x = (map->player->x +2);
        next_coord_caisse->y = (map->player->y);
}else if(direction == 2){
        next_coord->x = (map->player->x);
        next_coord->y = (map->player->y-1);
        next_coord_caisse->x = (map->player->x);
        next_coord_caisse->y = (map->player->y-2);
}else if(direction == 3){
        next_coord->x = (map->player->x);
        next_coord->y = (map->player->y+1);
```

```
        next_coord_caisse->x = (map->player->x);
        next_coord_caisse->y = (map->player->y+2);
}
```

3.1.3 Logique du jeu

La fonction principale du jeu implémente la logique suivante :

1. **Initialisation** :
   - Initialisation de la carte avec le niveau sélectionné
   - Calcul du centrage de la carte sur l'écran
2. **Boucle principale** :
   - Gestion des entrées utilisateur :
     - Touches directionnelles (haut, bas, gauche, droite) ou ZQSD
     - Touche Échap pour quitter et retourner au menu
     - Clics souris sur les boutons (retour, recommencer, solution)
   - Pour chaque déplacement :
     - Vérification de la validité du mouvement via `deplaceJoueur()`
     - Mise à jour de la direction visuelle du joueur
     - Vérification de la condition de défaite via `playerLoose()`
     - Vérification de la victoire en comparant `nombreCaisseGood()` avec le nombre total de caisses
   - En cas de victoire ou défaite :
     - Arrêt de la boucle de jeu
     - Affichage de l'écran approprié (victoire ou défaite)
   - Mise à jour continue de l'affichage du niveau
3. **Finalisation** :
   - Libération de la mémoire allouée pour la carte
   - Retour du choix de l'utilisateur pour la suite du programme

La fonction utilise trois fonctions essentielles pour la logique du jeu :
- `deplaceJoueur()` : Gère le déplacement du joueur et des caisses
- `nombreCaisseGood()` : Vérifie le nombre de caisses bien placées
- `playerLoose()` : Détermine si le joueur est dans une situation de défaite

3.2 Interface graphique avec SDL

Let me help improve the formatting and clarity of your SDL migration section. Here's an enhanced version :
latex

3.2.1 Migration vers SDL

La migration de notre jeu vers SDL a nécessité la maîtrise de plusieurs concepts fondamentaux de la bibliothèque, notamment :
- Window (fenêtre d'affichage)
- Renderer (moteur de rendu)
- Textures et surfaces
- Rectangles de collision (SDL_Rect)

Initialisation de SDL

L'initialisation s'effectue dans le fichier principal (`main.c`) où nous créons une fenêtre et un renderer, qui seront ensuite passés en paramètres aux différentes fonctions :

```
        if(0 != SDL_Init(SDL_INIT_TIMER | SDL_INIT_AUDIO |
            ↪ SDL_INIT_EVENTS |
        SDL_INIT_VIDEO | SDL_INIT_GAMECONTROLLER)){
                printf("Erreur d'init SDL");
                return EXIT_FAILURE;
        }

        SDL_Window *window = SDL_CreateWindow("SOKOBAN - MOTA -
            ↪ TRECHOT",
        SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,
        LARGEUR, HAUTEUR, SDL_WINDOW_SHOWN);

        SDL_Renderer *renderer = SDL_CreateRenderer(window, -1,
        SDL_RENDERER_ACCELERATED);

        if(window == NULL || renderer == NULL){
                SDL_DestroyWindow(window);
                SDL_Quit();
                return EXIT_FAILURE;
        }
```

Affichage de la carte

Pour transposer le jeu de la console vers SDL, nous avons implémenté une fonction qui parcourt la carte (matrice 2D) et affiche les textures correspondantes :

```
for(int x = 0; x < HAUTEUR / PIXEL; x++){
        for(int y = 0; y < LARGEUR / PIXEL; y++){
                SDL_Rect dest = {y * PIXEL, x * PIXEL, PIXEL,
                    ↪ PIXEL};
                SDL_RenderCopy(renderer, solTex, NULL, &dest);
        }
}


for(int x = 0; x < map->lignes; x++){
        for(int y = 0; y < map->colonnes; y++){
                SDL_Rect dest = {decalageX + y * PIXEL,
                        decalageY + x * PIXEL,
                        PIXEL, PIXEL};

                switch(map->map[x][y]){
                        case '#': SDL_RenderCopy(renderer,
                            ↪ murTex, NULL, &dest); break;
```

```
                        case 'P':

                        switch(depTextureChoice){
                                case -1: SDL_RenderCopy(renderer
                                    ↪ , playerTexFace, NULL, &
                                    ↪ dest); break;
                                case 0:  SDL_RenderCopy(renderer
                                    ↪ , playerTexBack, NULL, &
                                    ↪ dest); break;
                        }
                        break;
                        case 'C': SDL_RenderCopy(renderer,
                            ↪ boxTex, NULL, &dest); break;
                        case 'I': SDL_RenderCopy(renderer,
                            ↪ goalTex, NULL, &dest); break;
                }
        }
}
```

Améliorations graphiques

Pour améliorer l'expérience utilisateur, nous avons ajouté plusieurs fonctionnalités :

- **Animation du joueur** : Une variable `depTextureChoice` permet de modifier la texture du joueur selon sa direction de déplacement (-1 : face, 0 : dos, 1 : gauche, 2 : droite).
- **Interface interactive** : Implémentation de boutons cliquables grâce à la fonction `SDL_PointInRect()`.

Exemple d'implémentation d'un bouton de pause :

```
else if(event.type == SDL_MOUSEBUTTONDOWN){
        int mouseX = event.button.x;
        int mouseY = event.button.y;
        if(SDL_PointInRect(&(SDL_Point){mouseX, mouseY}, &
            ↪ destButtonPause)){
                pauseFlag = !pauseFlag;
        }
}
```

3.2.2 Gestion des assets graphiques

Sources des assets

Pour notre projet, nous avons utilise des assets gratuits et open source provenant de deux sources principales :

- Kenney.nl - Sokoban Pack
- Prinbles - Silent Pack

Organisation et format

Tous les assets graphiques sont stockes dans le repertoire `sokoban/textures` au format BMP. Ce format a ete choisi car il est nativement supporte par SDL, mais presente une limitation importante : l'absence de gestion de la transparence.

Manipulation des textures avec SDL

La gestion des textures dans notre jeu suit un processus en quatre etapes :

```
// 1. Chargement de l'image en memoire sous forme de surface
SDL_Surface *surfaceExemple = SDL_LoadBMP("textures/texture.bmp"
    ↪ );

// 2. Conversion de la surface en texture pour le rendu
SDL_Texture *textureExemple = SDL_CreateTextureFromSurface(
    ↪ renderer,surfaceExemple);

// 3. Liberation de la surface (non necessaire apres creation de
    ↪  la texture)
SDL_FreeSurface(surfaceExemple);

// 4. Liberation de la texture quand elle n'est plus necessaire
SDL_DestroyTexture(textureExemple);
```

Cette approche permet une gestion efficace de la memoire tout en maintenant les textures disponibles pour le rendu pendant toute la duree necessaire a leur utilisation.

Note : Le format BMP, bien que simple a utiliser avec SDL, presente des limitations en termes de transparence. Pour un futur developpement, il pourrait etre interessant d'envisager l'utilisation de SDL_image qui permet de gerer des formats plus avances comme PNG avec transparence.

3.2.3 Intégration de SDL_TTF

Pour l'affichage de texte dans notre jeu, nous avons intégré la bibliothèque SDL_TTF (SDL TrueType Font). Cette bibliothèque complémentaire à SDL permet de rendre du texte de haute qualité à partir de polices TrueType.

Initialisation et utilisation

L'utilisation de SDL_TTF suit un processus similaire à SDL, avec ses propres fonctions d'initialisation et de fermeture :

```
        // 1. Initialisation de SDL_TTF
        TTF_Init();

        // 2. Chargement de la police
        TTF_Font *font = TTF_OpenFont("textures/police.ttf", 24)
            ↪ ;  // 24 = taille en points

        // 3. Creation du texte
```

```
SDL_Color color = {255, 255, 255, 255};  // Couleur
    ↪ blanche
SDL_Surface *textSurface = TTF_RenderText_Solid(font, "
    ↪ Texte a afficher", color);
SDL_Texture *textTexture = SDL_CreateTextureFromSurface(
    ↪ renderer, textSurface);

// 4. Liberation des ressources
SDL_FreeSurface(textSurface);
TTF_CloseFont(font);
TTF_Quit();
```

Utilisation dans notre jeu

Cette bibliothèque est particulièrement utile pour afficher :
— Le nombre de déplacements
— Le chronomètre

3.3 Fonctionnalités avancées

3.3.1 Système de niveaux

Interface de sélection

Le système de sélection des niveaux est implémenté via la fonction :

```
int selectLevel(SDL_Renderer * renderer, int * currentLevel);
```

Cette fonction prend deux paramètres :
— `renderer` : Le contexte de rendu SDL
— `currentLevel` : Un pointeur vers le niveau actuel, stocké dans le main

Interface graphique

L'interface de sélection se présente sous la forme d'une page simple composée de :
— Un fond uniforme
— 20 boutons cliquables, chacun représentant un niveau
— Un bouton retour pour revenir au menu principal
Chaque bouton est implémenté comme un rectangle SDL avec un numéro de niveau, permettant une navigation intuitive entre les différents niveaux disponibles.

Structure des niveaux

Notre jeu propose 20 niveaux avec une progression de difficulté croissante :
— Niveaux 1-5 : Introduction aux mécaniques de base
— Niveaux 6-10 : Puzzles de difficulté moyenne
— Niveaux 11-15 : Défis plus complexes
— Niveaux 16-20 : Puzzles avancés nécessitant une réflexion approfondie

Gestion du niveau courant

L'utilisation d'un pointeur pour `currentLevel` permet :
— La persistance du niveau sélectionné entre les différentes parties
— La modification directe du niveau depuis n'importe quelle fonction
— Le retour au menu de sélection tout en conservant la progression

3.3.2 Gestion des scores

Système de calcul

Pour simplifier le calcul du score des joueurs, nous avons opté pour une formule de score basique mais efficace :

```
score = temps / nombre_de_deplacements
```

Cette formule prend en compte :
— Le temps total de résolution du niveau (en secondes)
— Le nombre total de déplacements effectués

Affichage des scores

Le score est affiché dans deux interfaces distinctes de fin de partie :
— `playerWin()` : Interface affichée en cas de victoire
— `playerLooseSDL()` : Interface affichée en cas de défaite
Ces interfaces présentent :
— Le score final du joueur
— Des options pour :
  — Revenir au menu principal
  — Rejouer le niveau actuel
  — Accéder au niveau suivant (uniquement en cas de victoire)

```
int playerWin(SDL_Renderer* renderer, int* currentLevel,
int nbDep, unsigned int time);

int playerLooseSDL(SDL_Renderer* renderer, int* currentLevel,
int nbDep, unsigned int time);
```

3.3.3 Chronometre

Calcul du temps ecoule

SDL ne fournit pas de fonction de chronometrage direct. Le processus se decompose en plusieurs etapes :

```
// Initialisation du chronometre
unsigned int time = SDL_GetTicks();

// Calcul du temps ecoule en millisecondes
unsigned int tempsPasser = SDL_GetTicks() - time;
```

```
// Conversion en secondes et minutes
int seconds = tempsPasser / 1000;       // Conversion ms -> s
int minutes = seconds / 60;             // Extraction des minutes
seconds = seconds % 60;                 // Reste en secondes
```

Affichage avec SDL_TTF

Pour afficher le temps a l'ecran, il est necessaire de convertir les valeurs numeriques en chaine de caracteres formatee, puis d'utiliser SDL_TTF pour le rendu :

```
// Format du temps en texte
char timeText[20];
snprintf(timeText, sizeof(timeText), "%02d:%02d", minutes,
    ↪ seconds);

// Creation de la surface de texte
SDL_Surface *timeSurface = TTF_RenderText_Solid(font, timeText,
    ↪ color);

// Conversion en texture pour l'affichage
SDL_Texture *timeTexture = SDL_CreateTextureFromSurface(renderer
    ↪ , timeSurface);
```

Notes techniques

- Le format `%02d:%02d` assure l'affichage sur deux chiffres
- `SDL_TTF` convertit le texte en texture
- Les surfaces et textures doivent etre liberees apres usage

3.3.4 Détection des situations bloquantes

Notre système de détection des situations bloquantes est implémenté dans la fonction :

```
int playerLoose(Map * map);
```

Principe de détection

La fonction vérifie les conditions critiques pour chaque caisse :

- Si une caisse touche deux murs du même côté (sauf si la position est une case objectif)

Implémentation

Le code suivant effectue la vérification des situations bloquantes :

```
// Verification des limites de la carte
if(i+1 < map->lignes && j+1 < map->colonnes && i-1 >= 0 && j-1
    ↪ >= 0) {
```

```
    // Detection des configurations bloquantes
    if((map->map[i+1][j] == '#' && map->map[i][j+1] == '#') || //
        ↪ Coin bas-droite
        (map->map[i-1][j] == '#' && map->map[i][j-1] == '#') ||
            ↪ // Coin haut-gauche
        (map->map[i+1][j] == '#' && map->map[i][j-1] == '#') ||
            ↪ // Coin bas-gauche
        (map->map[i-1][j] == '#' && map->map[i][j+1] == '#'))
            ↪ // Coin haut-droite
    {
        choix1 = 1;
        break; // Sortie immediate si situation bloquante
            ↪ detectee
    }
}
```

Configurations bloquantes

La fonction vérifie quatre configurations de blocage :
- Coin bas-droite : murs au sud et à l'est
- Coin haut-gauche : murs au nord et à l'ouest
- Coin bas-gauche : murs au sud et à l'ouest
- Coin haut-droite : murs au nord et à l'est

Optimisation

Pour améliorer les performances :
- Vérification préalable des limites de la carte
- Sortie immédiate dès qu'une situation bloquante est détectée
- Pas de vérification supplémentaire si la caisse est sur une case objectif
- Vérification uniquement quand un joueur déplace une caisse pour éviter les appels intensifs à cette fonction.

Chapitre 4

Optimisation et Debugging

4.1 Stratégies d'optimisation

4.2 Optimisation du jeu

Pour optimiser notre jeu, nous avons adopté plusieurs stratégies.

4.2.1 Gestion des instances

Nous avons d'abord réfléchi à une logique qui évite la création de multiples instances d'une même fonction. Cette approche permet de :

— Réduire la consommation des ressources
— Limiter les risques de crash
— Optimiser les performances

Cette logique est implémentée dans la boucle principale du main qui appelle chaque fonction une par une selon la situation.

4.2.2 Gestion de la mémoire

Nous avons veillé à ce que chaque ressource soit correctement libérée :

— Pointeurs
— Surfaces
— Textures

Cette libération est garantie dans tous les cas, quel que soit le déroulement du code.

4.2.3 Optimisation des variables

Nous avons optimisé la gestion des variables de plusieurs façons :

— Déclaration au début de chaque fonction
— Initialisation unique des variables
— Modification des valeurs plutôt que réinitialisation dans les boucles
— Élimination des doubles initialisations présentes avant l'optimisation

4.3 Résolution des bugs

4.3.1 Méthodologie de test

Outre le fait de tester le jeu à de multiples reprises pour observer son comportement, nous avons utilisé l'outil d'analyse mémoire Valgrind pour détecter les fuites de mémoire.

4.3.2 Exemple de correction d'une fuite mémoire

Prenons un exemple concret où Valgrind nous a signalé une fuite mémoire :

```
==5355==
at 0x4846828: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
by 0x109BD6: deplaceJoueur (jeu.c:151)
by 0x10A631: playGameSDL (jeu.c:321)
==5355== by 0x1096D1: main (main.c:58)
==5355==
```

Cette trace nous indique une fuite de mémoire dans la fonction `deplaceJoueur`.

4.3.3 Analyse du problème

Le problème se situait dans la structure du code :

— Les structures `current_coord`, `next_coord` et `next_coord_caisse` n'étaient jamais libérées

— Les conditions effectuaient un `return` direct, faisant jamais la libération de la mémoire

4.3.4 Solution implémentée

Pour résoudre ce problème, nous avons :

1. Créé une variable `choix` pour stocker le résultat
2. Déplacé les `return` en fin de fonction
3. Ajouté la libération de mémoire avant le `return` final

```c
int deplaceJoueur(int direction,Map * map){
    }else if(direction == 3){          //x
        next_coord->x = (map->player->x);
        next_coord->y = (map->player->y+1);
        next_coord_caisse->x = (map->player->x);
        next_coord_caisse->y = (map->player->y+2);
        enregistreDep('d');
    }

    // On vérifie dans quelle situation nous sommes (déplacement autorisé ou non)
    int situation = deplacementValide(next_coord,map);

    if(situation == 1){
        map->map[current_coord->x][current_coord->y] = ' ';
        map->map[next_coord->x][next_coord->y] = 'P';
        map->player->x = next_coord->x;
        map->player->y = next_coord->y;
        return 1;
    }else if(situation == 0){
        return 0;
    }else if(situation == 2){
        //On vérifie si la caisse peut être déplacée (si la case suivante est vide)
        if(deplacementValide(next_coord_caisse,map) == 1){
        map->map[current_coord->x][current_coord->y] = ' ';
        map->map[next_coord->x][next_coord->y] = 'P';
        map->map[next_coord_caisse->x][next_coord_caisse->y] = 'C';
        map->player->x = next_coord->x;
        map->player->y = next_coord->y;
        return 1;
        }


    }
    free(current_coord);
    free(next_coord);
    free(next_coord_caisse);
    return 0;
}
```

FIGURE 4.1 – Code original avec fuite mémoire

```c
int deplaceJoueur(int direction,Map * map){
    }else if(direction == 3){          //x
        next_coord_caisse->y = (map->player->y+2);
        enregistreDep('d');
    }

    // On vérifie dans quelle situation nous sommes (déplacement autorisé ou non)
    int situation = deplacementValide(next_coord,map);

    if(situation == 1){
        map->map[current_coord->x][current_coord->y] = ' ';
        map->map[next_coord->x][next_coord->y] = 'P';
        map->player->x = next_coord->x;
        map->player->y = next_coord->y;
        choix = 1;
    }else if(situation == 0){
        choix = 0;
    }else if(situation == 2){
        //On vérifie si la caisse peut être déplacée (si la case suivante est vide)
        if(deplacementValide(next_coord_caisse,map) == 1){
        map->map[current_coord->x][current_coord->y] = ' ';
        map->map[next_coord->x][next_coord->y] = 'P';
        map->map[next_coord_caisse->x][next_coord_caisse->y] = 'C';
        map->player->x = next_coord->x;
        map->player->y = next_coord->y;
        choix = 1;
        }


    }
    free(current_coord);
    free(next_coord);
    free(next_coord_caisse);
    return choix;
}

//Calcul le nombre de caisses bien placées avec les donnees de la map
int nombreCaisseGood(Map * map){
```

FIGURE 4.2 – Code corrigé avec libération mémoire

Chapitre 5

Compilation et utilisation

5.1 Makefile et compilation

5.1.1 Dependances

Notre projet necessite les bibliotheques suivantes :
- SDL2 pour la gestion graphique
- SDL2_ttf pour la gestion des polices
- Bibliotheques standard C :
  - stdio.h
  - stdlib.h
  - time.h

5.1.2 Structure du Makefile

Voici la configuration de notre Makefile :

```
CC = gcc
CFLAGS = -Wall -Wextra -g $(shell sdl2-config --cflags)
LIBS = $(shell sdl2-config --libs) -lSDL2_ttf

TARGET = sokoban
SRCS = main.c jeu.c affichage.c
OBJS = $(SRCS:.c=.o)

# Compilation par defaut
all: $(TARGET)

# Creation de l'executable
$(TARGET): $(OBJS)
$(CC) $(OBJS) -o $(TARGET) $(CFLAGS) $(LIBS)

# Compilation des fichiers objets
%.o: %.c jeu.h affichage.h
$(CC) $(CFLAGS) -c $< -o $@

# Nettoyage des fichiers
```

```
clean:
rm -f $(OBJS) $(TARGET)

# Execution du programme
run: $(TARGET)
./$(TARGET)
```

5.1.3 Options de compilation

Le Makefile comprend :
— Flags de detection d'erreurs (-Wall -Wextra)
— Mode debug active (-g)
— Configuration SDL2 automatique
— Gestion des dependances entre fichiers

5.1.4 Commandes disponibles

Pour utiliser le projet :

```
make        # Compile le projet
make run    # Lance le programme
make clean  # Nettoie les fichiers
```

Chapitre 6

Manuel d'utilisation

6.1 Installation des dépendances

Pour installer le jeu, vous aurez besoin des éléments suivants :

6.1.1 Compilateur GCC

Installation du compilateur C :

```
sudo apt install gcc
```

6.1.2 Bibliothèques SDL2

Installation de SDL2 et SDL2_TTF :

```
sudo apt install libsdl2-dev
sudo apt install libsdl2-ttf-dev
```

6.1.3 Utilitaire Make

Installation de l'outil de compilation :

```
sudo apt install make
```

6.2 Compilation et exécution

Une fois les dépendances installées, suivez ces étapes :

1. Ouvrez un terminal dans le dossier du jeu
2. Compilez le projet avec la commande :

   ```
   make
   ```

3. Lancez le jeu avec :

   ```
   ./sokoban
   ```

6.3 Note importante

— Ces commandes sont valables pour les distributions Linux basées sur Debian/Ubuntu
— Pour d'autres distributions, les commandes peuvent varier

6.4 Commandes et contrôles

Le jeu propose deux configurations de contrôle pour s'adapter aux préférences du joueur.

6.4.1 Déplacements

— **Configuration ZQSD :**
  — Z : Déplacement vers le haut
  — Q : Déplacement vers la gauche
  — S : Déplacement vers le bas
  — D : Déplacement vers la droite
— **Configuration flèches directionnelles :**
  — ↑ : Déplacement vers le haut
  — ← : Déplacement vers la gauche
  — ↓ : Déplacement vers le bas
  — → : Déplacement vers la droite

6.4.2 Commandes système

— **Échap** : Quitter le jeu

6.4.3 Note

Les deux configurations de contrôle sont disponibles simultanément, permettant au joueur d'utiliser celle qui lui convient le mieux sans avoir à modifier les paramètres.

6.5 Fonctionnalités disponibles

6.5.1 Menu d'accueil

— Bouton **Play** : Accès à la sélection des niveaux
— Bouton **Quitter** : Fermeture du jeu

6.5.2 Interface de jeu

Pendant une partie, plusieurs éléments sont affichés à l'écran :
— **Timer** : Affiche le temps écoulé depuis le début du niveau
— **Compteur de déplacements** : Indique le nombre de mouvements effectués
— **Boutons de contrôle** :
  — Recommencer le niveau

— Quitter le niveau

6.5.3 Système d'aide

Un système d'aide est disponible via le bouton représenté par une ampoule, situé en bas à droite de l'écran. Ce système offre :

— Visualisation de la solution du niveau
— Contrôles de la démonstration :
  — Réglage de la vitesse d'exécution
  — Bouton pause/reprise de la démonstration

Chapitre 7

Conclusion

7.1 Conclusion

7.1.1 Évolution du projet

Ce qui avait débuté comme un simple jeu en mode console s'est progressivement transformé en un projet plus ambitieux. Malgré nos appréhensions initiales concernant le temps imparti, nous avons réussi à :

— Développer une interface graphique complète
— Implémenter des fonctionnalités avancées
— Créer une expérience utilisateur satisfaisante

7.1.2 Apprentissages

Ce projet nous a permis de :
— Sortir de notre zone de confort universitaire
— Acquérir des compétences en développement de jeux
— Apprendre à gérer un projet de bout en bout
— Maîtriser de nouveaux outils et technologies

7.1.3 Perspectives

Cette expérience nous a motivés à :
— Envisager le développement d'autres jeux
— Explorer de nouveaux projets personnels
— Approfondir nos connaissances en développement graphique

7.1.4 Bilan

Pour une première expérience en développement de jeux, nous sommes satisfaits du résultat obtenu. Les défis rencontrés et surmontés nous ont permis de progresser significativement dans notre parcours de développeurs.

7.2 Perspectives d'amélioration

7.2.1 Optimisation technique

- **Gestion de la mémoire :**
  - Création d'une structure centralisée pour les textures
  - Amélioration du système de libération des ressources
  - Prévention des fuites mémoire
- **Optimisation du code :**
  - Refactorisation pour plus de clarté
  - Amélioration des performances
  - Réduction de la complexité

7.2.2 Nouvelles fonctionnalités

- **Système de progression :**
  - Verrouillage des niveaux
  - Système de déblocage progressif
  - Sauvegarde de la progression
- **Améliorations potentielles :**
  - Système de scores
  - Classement des joueurs
  - Nouveaux niveaux
  - Éditeur de niveaux

7.3 Captures d'écran du jeu

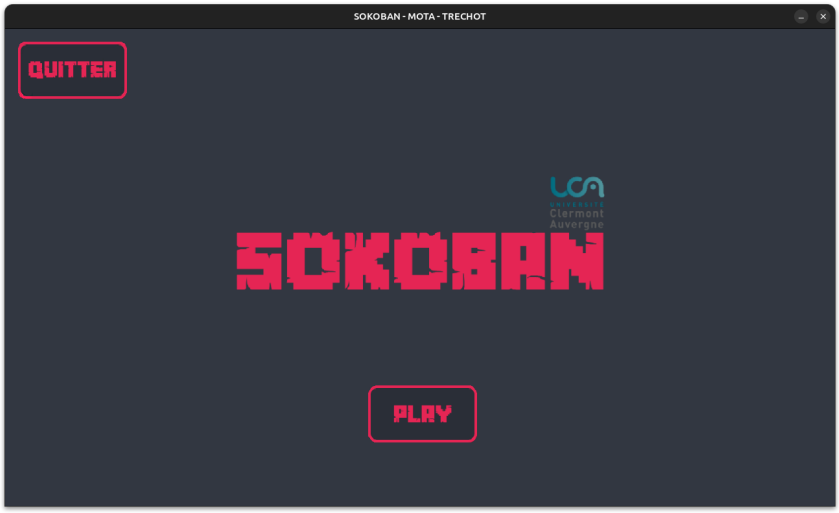

FIGURE 7.1 – Menu d'accueil du jeu

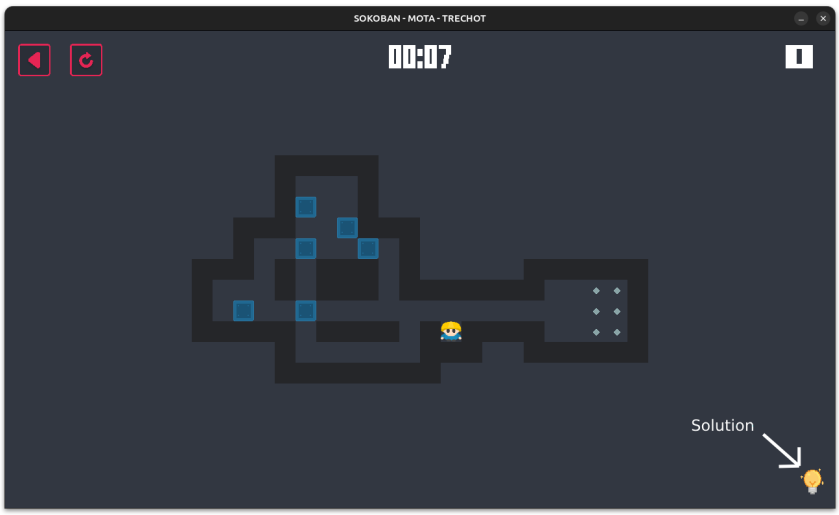

FIGURE 7.2 – Vue d'un niveau en cours de jeu

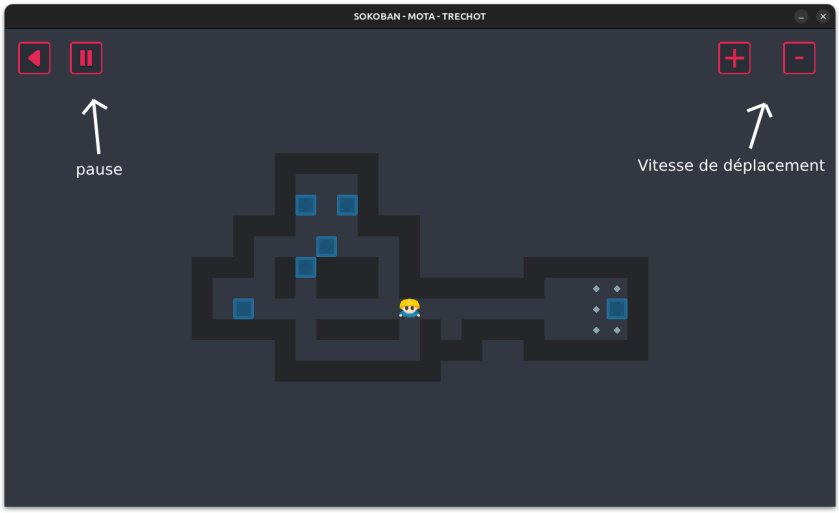

FIGURE 7.3 – Affichage de la solution du niveau

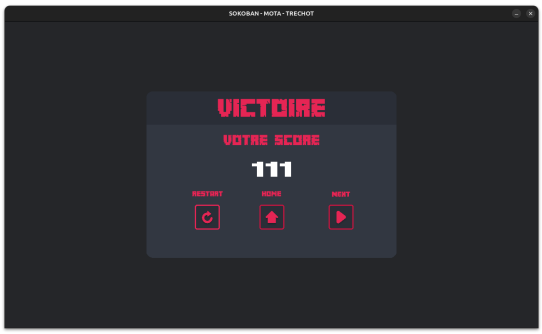

FIGURE 7.4 – Écran de victoire

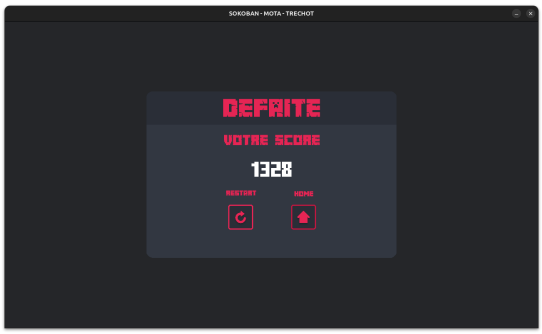

FIGURE 7.5 – Écran de défaite

Bibliographie

[1] Wikipedia , *Sokoban* : https://fr.wikipedia.org/wiki/Sokoban

[2] *Tutorial SDL* : https://wiki.libsdl.org/SDL2/Tutorials.

[3] *Tutorial SDL* : https://alexandre-laurent.developpez.com/tutoriels/sdl-2/creer-premieres-fenetres/.

[4] *Tutorial C* : https://zestedesavoir.com/tutoriels/755/le-langage-c-1/.

[5] *Tutorial SDL* : https://perso.isima.fr/loic/unixc/tpc-sdlflood.php.

[6] *Tutorial Makefile* : https://borntocode.fr/creer-un-makefile-pour-son-projet/.

[7] *Tutorial Makefile* : https://perso.univ-lyon1.fr/jean-claude.iehl/Public/educ/Makefile.html.