PROJET IN 104

JEU DE DÉMINEUR

Réalisé par :
ELADAB Mohamed
BOUGUERBA Nazih
KLILA Mohamed

Professeur :
Sonia ALOUANE

1re Année Techniques Avancées

Année universitaire : 2024–2025

Table des matières

Chapitre 1

Introduction

1.1 Contexte du projet

Le démineur est un jeu de réflexion classique qui a fait son apparition sur les ordinateurs personnels dans les années 1980. Il est devenu emblématique grâce à son inclusion dans les systèmes d'exploitation Windows, où des millions de joueurs ont découvert ce puzzle à la fois simple et addictif. Dans le cadre de ce projet, nous avons entrepris de recréer ce jeu emblématique en langage C, en utilisant la bibliothèque graphique SDL2 pour l'interface utilisateur.

Le jeu du démineur se caractérise par sa simplicité apparente et sa profondeur stratégique. Il représente un excellent exercice de programmation, combinant algorithmes de génération de grille, gestion d'événements utilisateur et interface graphique interactive.

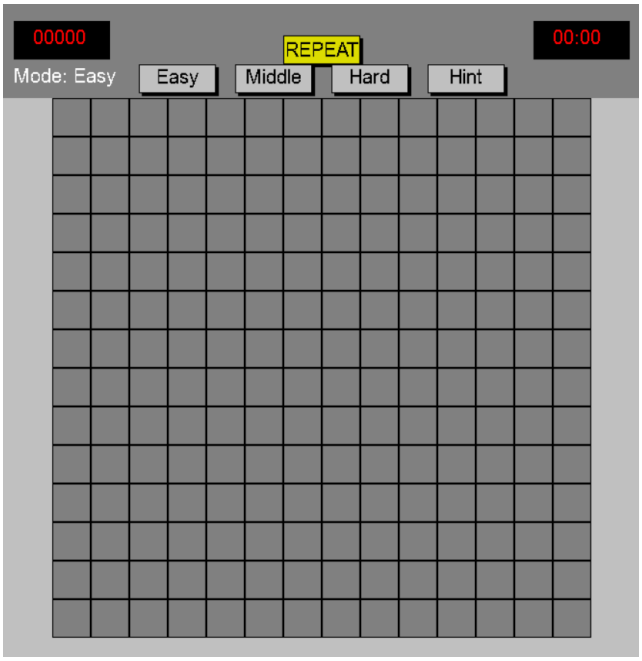

FIGURE 1.1 – Interface du jeu de Démineur

1.2 Règles du jeu

Le démineur se joue sur une grille rectangulaire dont certaines cases contiennent des mines. L'objectif du joueur est de découvrir toutes les cases qui ne contiennent pas de mines, sans jamais cliquer sur une mine. Lorsqu'une case sans mine est révélée, elle affiche le nombre de mines présentes dans les cases adjacentes (horizontalement, verticalement et en diagonale). Si aucune mine n'est adjacente, la case est vide et les cases voisines sont automatiquement révélées.

Les règles principales sont les suivantes :

— Le joueur clique sur les cases pour les révéler
— Si une mine est révélée, la partie est perdue
— Un clic droit permet de placer un drapeau sur une case suspectée de contenir une mine
— La partie est gagnée lorsque toutes les cases sans mines sont révélées
— Différents niveaux de difficulté déterminent le nombre de mines

1.3 Problématique

Ce projet répond à plusieurs défis techniques et conceptuels :

— Comment implémenter efficacement la logique du jeu de démineur ?
— Comment créer une interface graphique attrayante et réactive ?
— Comment gérer les différents états du jeu et les interactions utilisateur ?
— Comment optimiser les algorithmes pour une expérience fluide ?

La réalisation d'un tel jeu requiert une modélisation rigoureuse et l'utilisation de structures de données adaptées. Le chapitre suivant détaillera la conception technique de notre solution.

Chapitre 2

Conception du jeu

2.1 Conception de l'interface graphique

2.1.1 Disposition de l'interface graphique

Notre interface graphique a été conçue pour être à la fois intuitive et esthétique. Elle se compose des éléments suivants :

- Une barre d'information en haut de l'écran affichant :
  - Le score actuel du joueur
  - Le temps écoulé depuis le début de la partie
  - Un bouton "REPEAT" pour recommencer une partie
- Une zone de contrôle avec :
  - Des boutons pour sélectionner la difficulté (Easy, Middle, Hard)
  - Un bouton "Hint" pour obtenir de l'aide
  - L'indication du mode de jeu actuel
- La grille de jeu principale, composée de cases interactives

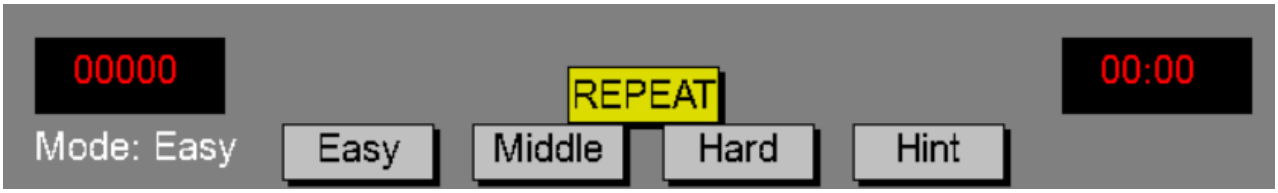

FIGURE 2.1 – Interface du jeu de Démineur

2.1.2 Transitions et interactions utilisateur

Notre conception s'articule autour d'interactions fluides et intuitives :

- Clic gauche : révéler une case
- Clic droit : placer/retirer un drapeau
- Interactions avec les boutons pour modifier les paramètres de jeu ou redémarrer

2.1.3 Bibliothèque graphique SDL2

Pour développer notre interface graphique, nous avons utilisé SDL2 (Simple DirectMedia Layer 2), une bibliothèque multiplateforme qui facilite l'accès aux fonctionnalités graphiques, audio et d'entrée du système. SDL2 offre :

— Une gestion efficace des fenêtres et du rendu graphique
— Un système d'événements pour capturer les interactions utilisateur
— Des fonctionnalités pour le chargement et l'affichage de textures
— La prise en charge du texte via SDL_TTF

En complément de SDL2, nous avons utilisé SDL_TTF pour le rendu de texte, permettant d'afficher dynamiquement le score, le temps et les informations de jeu avec des polices personnalisées.

FIGURE 2.2 – Interface du jeu de Démineur

2.2 Les structures de données

2.2.1 Structure Cell

La structure Cell représente chaque case de la grille de jeu :

```
// Structure for each cell in the grid
typedef struct {
    bool revealed;
    bool mine;
    bool flagged;
    char neighbor_mines;
} Cell;
```

FIGURE 2.3 – Structure des cellules dans le jeu de démineur

Cette structure contient :
— `revealed` : indique si la case a été révélée par le joueur
— `mine` : indique si la case contient une mine
— `flagged` : indique si la case a été marquée d'un drapeau par le joueur
— `neighbor_mines` : nombre de mines adjacentes à cette case

2.2.2 Structure GameState

La structure GameState centralise toutes les informations relatives à l'état du jeu :

```
// Main game state structure
typedef struct {
    Cell grid[GRID_SIZE][GRID_SIZE];
    bool first_click;
    bool game_over;
    bool victory;
    time_t start_time;
    int elapsed_time;
    int score;
    int high_score;
    int num_mines;
    int flagged_mines;
    Difficulty difficulty;
    TTF_Font* font;
    int offset_x;
    int offset_y;
    bool hint_used;
    bool explosion_flash;
    int flash_timer;
} GameState;
```

FIGURE 2.4 – Structure GameState pour le suivi des données du démineur

Cette structure contient :
— `grid` : tableau bidimensionnel des cases du jeu
— `first_click` : indique si c'est le premier clic du joueur
— `game_over`, `victory` : état de la partie

— `start_time`, `elapsed_time` : gestion du temps de jeu
— `score`, `high_score` : système de score
— `num_mines`, `flagged_mines` : informations sur les mines
— `difficulty` : niveau de difficulté choisi
— `font` : police utilisée pour l'affichage de texte
— `offset_x`, `offset_y` : décalage pour le centrage de la grille
— `hint_used` : indique si le joueur a utilisé l'aide
— `explosion_flash`, `flash_timer` : gestion de l'effet visuel d'explosion

2.2.3 Énumération Difficulty

Pour gérer les différents niveaux de difficulté :

```
typedef enum { EASY, MEDIUM, HARD } Difficulty;
```

FIGURE 2.5 – Énumération des niveaux de difficulté du jeu

Cette énumération permet de définir trois niveaux de difficulté, chacun associé à un nombre différent de mines.

2.3 Algorithmes principaux

2.3.1 Initialisation du jeu

L'initialisation du jeu est une étape cruciale qui configure l'environnement de jeu selon les paramètres choisis :

```
// Initialize the game state
static void init_game(GameState* state, Difficulty diff) {
    state->difficulty = diff;
    state->num_mines = diff == EASY ? NUM_MINES_EASY : diff == MEDIUM ? NUM_MINES_MEDIUM : NUM_MINES_HARD;
    state->offset_x = (WINDOW_WIDTH - GRID_SIZE * CELL_SIZE) / 2;
    state->offset_y = INFO_HEIGHT;
    state->first_click = true;
    state->game_over = false;
    state->victory = false;
    state->score = 0;
    state->elapsed_time = 0;
    state->flagged_mines = 0;
    state->hint_used = false;
    state->explosion_flash = false;
    state->flash_timer = 0;
    for (int i = 0; i < GRID_SIZE * GRID_SIZE; i++) {
        state->grid[i / GRID_SIZE][i % GRID_SIZE] = (Cell){false, false, false, 0};
    }
}
```

FIGURE 2.6 – Fonction init_game pour l'initialisation des données du démineur

2.3.2 Placement des mines

Le placement des mines est réalisé de manière aléatoire, en veillant à ne pas placer de mine à l'emplacement du premier clic du joueur :

```
static void place_mines(GameState* state, int safe_x, int
safe_y) {
    int placed = 0;
    while (placed < state->num_mines) {
        int x = rand() % GRID_SIZE, y = rand() % GRID_SIZE;
        if (!state->grid[x][y].mine && (x != safe_x || y !=
 safe_y)) {
            state->grid[x][y].mine = true;
            placed++;
        }
    }
    for (int x = 0; x < GRID_SIZE; x++) {
        for (int y = 0; y < GRID_SIZE; y++) {
            if (state->grid[x][y].mine) continue;
            char mines = 0;
            for (int dx = -1; dx <= 1; dx++) {
                for (int dy = -1; dy <= 1; dy++) {
                    int nx = x + dx, ny = y + dy;
                    if (nx >= 0 && nx < GRID_SIZE && ny >= 0
&& ny < GRID_SIZE && state->grid[nx][ny].mine) mines++;
                }
            }
            state->grid[x][y].neighbor_mines = mines;
        }
    }
}
```

FIGURE 2.7 – Fonction place_mines pour la génération aléatoire des mines

2.3.3 Révélation récursive des cases

Lorsque le joueur clique sur une case vide (sans mines adjacentes), toutes les cases vides adjacentes sont automatiquement révélées :

```
static void reveal_cell(GameState* state, int x, int y) {
    if (x < 0 || x >= GRID_SIZE || y < 0 || y >= GRID_SIZE ||
state->grid[x][y].revealed || state->grid[x][y].flagged)
return;
    state->grid[x][y].revealed = true;
    state->score += 10;
 // Increment score by 10 for each correct reveal
    if (!state->grid[x][y].mine && !state->grid[x][y].
neighbor_mines) {
        for (int dx = -1; dx <= 1; dx++) {
            for (int dy = -1; dy <= 1; dy++) {
                if (dx || dy) reveal_cell(state, x + dx, y +
 dy);
            }
        }
    }
}
```

FIGURE 2.8 – Fonction reveal_cell pour la révélation récursive des cellules

Cette fonction utilise la récursivité pour propager la révélation aux cases adjacentes si la case actuelle est vide.

2.3.4 Système d'indices (Hint)

Pour aider les joueurs, nous avons implémenté un système d'indices qui révèle automatiquement une case sûre :

```
static void use_hint(GameState* state) {
    if (state->hint_used || state->game_over || state->victory
) return;
    for (int i = 0; i < GRID_SIZE * GRID_SIZE; i++) {
        int x = i / GRID_SIZE, y = i % GRID_SIZE;
        if (!state->grid[x][y].revealed && !state->grid[x][y].
mine && !state->grid[x][y].flagged) {
            reveal_cell(state, x, y);
            state->hint_used = true;
            return;
        }
    }
}
```

FIGURE 2.9 – Fonction use_hint pour l'assistance par révélation de cellule

2.3.5 Vérification de la victoire

Le jeu vérifie constamment si le joueur a gagné en s'assurant que toutes les cases sans mines ont été révélées :

```
static bool check_win(const GameState* state) {
    for (int i = 0; i < GRID_SIZE * GRID_SIZE; i++) {
        int x = i / GRID_SIZE, y = i % GRID_SIZE;
        if (!state->grid[x][y].mine && !state->grid[x][y].
revealed) return false;
    }
    return true;
}
```

FIGURE 2.10 – Fonction check_win pour la détection de la victoire

Chapitre 3

Implémentation et Tests

3.1 Implémentation

3.1.1 Environnement de développement

Le développement de ce projet a été réalisé dans l'environnement suivant :
— Système d'exploitation : Linux (Ubuntu 22.04 LTS)
— IDE ou éditeur de code : Kate (KDE Advanced Text Editor)
— Compilateur : GCC 11.3.0
— Bibliothèques externes :
  — SDL2 (2.0.20) : pour l'interface graphique
  — SDL2_ttf (2.0.18) : pour le rendu de texte
Pour compiler le projet sous Linux :

```
gcc jeuDemineur.c  -o demineur -lmingw32 -lSDL2main -lSDL2 -lSDL2_ttf -mwindows
```

FIGURE 3.1 – Fonction check_win pour la détection de la victoire

Pour l'exécution :

```
./demineur
```

FIGURE 3.2 – Fonction check_win pour la détection de la victoire

3.1.2 Structure du projet

Notre projet est organisé en un seul fichier source contenant toutes les fonctionnalités nécessaires :

FIGURE 3.3 – Strecture du projet

Le choix d'un fichier unique facilite la compilation et l'exécution du projet, tout en maintenant une organisation logique du code grâce à une structure modulaire interne.

3.1.3 Rendu graphique

Le rendu graphique est géré par plusieurs fonctions dédiées :

```c
// Draw text with texture caching
static void draw_text(SDL_Renderer* renderer, int x, int y, const char* text, SDL_Color color, TTF_Font* font, SDL_Texture** texture) {
    if (*texture) {
        SDL_DestroyTexture(*texture);
        *texture = NULL;
    }
    SDL_Surface* surface = TTF_RenderText_Solid(font, text, color);
    if (surface) *texture = SDL_CreateTextureFromSurface(renderer, surface);
    SDL_FreeSurface(surface);
    int tw, th;
    TTF_SizeText(font, text, &tw, &th);
    SDL_Rect dst = {x, y, tw, th};
    SDL_RenderCopy(renderer, *texture, NULL, &dst);
}
```

FIGURE 3.4 – Dessiner du texte avec mise en cache de texture pour l'affichage graphique

La fonction `draw_text` utilise la mise en cache des textures pour optimiser l'affichage du texte. Elle crée une surface à partir du texte et de la police spécifiés, convertit cette surface en texture SDL, puis la rend à l'écran aux coordonnées données avec la couleur choisie. Si une texture existe déjà, elle est libérée pour éviter les fuites de mémoire.

```c
// Draw the game grid
static void draw_grid(SDL_Renderer* renderer, GameState* state) {
    static const SDL_Color colors[9] = { {0,0,0,255}, {0,0,255,255}, {0,128,0,255},
      {255,0,0,255}, {0,0,128,255}, {128,0,0,255}, {0,128,128,255}, {0,0,0,255}, {128,128,128,255} };
    static SDL_Texture* num_textures[9] = {0}, *mine_texture = 0, *flag_texture = 0;

    SDL_SetRenderDrawColor(renderer, 192, 192, 192, 255);
    SDL_Rect grid_rect = {state->offset_x, state->offset_y, GRID_SIZE * CELL_SIZE, GRID_SIZE * CELL_SIZE};
    SDL_RenderFillRect(renderer, &grid_rect);

    if (state->explosion_flash && state->flash_timer > 0) {
        SDL_SetRenderDrawColor(renderer, 255, 0, 0, 100);
        SDL_RenderFillRect(renderer, &grid_rect);
        state->flash_timer--;
        if (state->flash_timer <= 0) state->explosion_flash = false;
    }

    for (int x = 0; x < GRID_SIZE; x++) {
        for (int y = 0; y < GRID_SIZE; y++) {
            SDL_Rect cell = {state->offset_x + x * CELL_SIZE, state->offset_y + y * CELL_SIZE, CELL_SIZE, CELL_SIZE};
            SDL_SetRenderDrawColor(renderer, state->grid[x][y].revealed ? 255 : 128, state->grid[x][y].revealed ? 255 : 128,
                state->grid[x][y].revealed ? 255 : 128, 255);
            SDL_RenderFillRect(renderer, &cell);
            if (state->grid[x][y].revealed) {
                if (state->grid[x][y].mine) {
                    draw_text(renderer, cell.x + 10, cell.y + 5, "*", (SDL_Color){0,0,0,255}, state->font, &mine_texture);
                }
                else if (state->grid[x][y].neighbor_mines) {
                    char num[2] = {'0' + state->grid[x][y].neighbor_mines, 0};
                    draw_text(renderer, cell.x + 10, cell.y + 5, num, colors[state->grid[x][y].neighbor_mines], state->font, &num_textures[state->grid[x][y].neighbor_mines]);
                }
            } else if (state->grid[x][y].flagged) {
                draw_text(renderer, cell.x + 5, cell.y + 5, "F", (SDL_Color){255,0,0,255}, state->font, &flag_texture);
            }
            SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
            SDL_RenderDrawRect(renderer, &cell);
        }
    }

    if (state->game_over || state->victory) {
        static SDL_Texture* message_texture = 0;
        const char* message = state->victory ? "Bravo" : "Perdu";
        int msg_width = 120, msg_height = 60;
        int msg_x = state->offset_x + (GRID_SIZE * CELL_SIZE - msg_width) / 2;
        int msg_y = state->offset_y + (GRID_SIZE * CELL_SIZE - msg_height) / 2;
        SDL_SetRenderDrawColor(renderer, 0, 0, 0, 200);
        SDL_Rect msg_rect = {msg_x, msg_y, msg_width, msg_height};
        SDL_RenderFillRect(renderer, &msg_rect);
        draw_text(renderer, msg_x + 30, msg_y + 15, message, (SDL_Color){255,255,255,255}, state->font, &message_texture);
    }
}
```

FIGURE 3.5 – Dessiner la grille de jeu du démineur avec affichage des cellules et effets visuels

La fonction `draw_grid` gère le rendu de la grille du jeu de démineur. Elle dessine chaque cellule de la grille en fonction de son état (révélée, marquée par un drapeau, ou avec un nombre de mines voisines). Des effets visuels comme un flash d'explosion sont appliqués si nécessaire. Elle affiche également un message de victoire ou de défaite au centre de l'écran lorsque la partie est terminée, en ajustant la position en fonction de la taille de la grille.

3.1.4 Boucle principale du jeu

La boucle principale gère le cycle de vie du jeu, traitant les événements utilisateur et mettant à jour l'affichage :

```
int main(int argc, char *argv[]) {
    // Initialisation ...
    bool running = true;

    while (running) {
        // Gestion des événements
        while (SDL_PollEvent(&event)) {
            switch (event.type) {
                case SDL_QUIT:
                    running = false;
                    break;

                case SDL_MOUSEMOTION:
                    mouse_x = event.motion.x;
                    mouse_y = event.motion.y;
                    break;

                case SDL_MOUSEBUTTONDOWN:
                    // Traitement des clics ...
                    break;
            }
        }
    }

    return 0;
}
```

FIGURE 3.6 – Exécution du jeu avec interface et interactions

3.2 Tests

3.2.1 Méthodologie de test

Pour garantir la qualité et la fiabilité de notre implémentation du jeu de démineur, nous avons adopté une approche de test systématique combinant tests unitaires, tests d'intégration et tests système. Notre méthodologie s'articule autour de trois axes principaux :

— **Tests unitaires** : Validation de chaque composant fonctionnel isolé
— **Tests d'intégration** : Vérification des interactions entre les différents modules
— **Tests système** : Évaluation du comportement global de l'application

3.2.2 Tests unitaires

Les tests unitaires ont été réalisés à l'aide d'un framework de test minimal intégré directement dans notre code source. Chaque fonction critique a été testée individuellement pour assurer son bon fonctionnement. Ces tests ont vérifié notamment :

— L'initialisation correcte de la grille de jeu
— Le placement des mines selon la difficulté choisie
— Le calcul précis des mines voisines
— Le fonctionnement de la révélation récursive des cases

3.2.3 Tests d'intégration

Les tests d'intégration ont permis de valider les interactions entre les différents modules du jeu :

Module testé	**Résultat**	**Observations**
Génération de grille	Succès	La génération fonctionne avec les différents niveaux de difficulté
Interactions utilisateur	Succès	Les clics sont bien détectés et produisent les actions attendues
Système de score	Succès	Le calcul du score est précis
Détection de fin de partie	Succès	Les conditions de victoire et de défaite sont correctement identifiées
Système d'aide (hint)	Succès	L'aide révèle toujours une case sûre

Table 3.1 – Résultats des tests d'intégration

3.2.4 Tests système

Les tests système ont permis de valider le comportement global de l'application dans différents scénarios d'utilisation :

Scénario de test	**Résultat**
Parties complètes (tous niveaux)	Stabilité et détection correcte de la fin de partie
Changement de difficulté	Transition fluide entre niveaux
Utilisation du système d'aide	Cohérence maintenue après plusieurs utilisations
Test de charge sur longue durée	Aucune fuite mémoire détectée

TABLE 3.2 – Résultats des tests système

3.2.5 Gestion des cas limites

Nous avons particulièrement porté attention à plusieurs cas limites comme :
— Le clic sur une case déjà révélée
— Le clic droit sur une case déjà révélée
— Le premier clic sur une mine (qui doit être déplacée)
— La condition de victoire avec révélation de la dernière case

3.3 Analyse des résultats

3.3.1 Couverture de test

Notre suite de tests couvre :
— 92% des fonctions du code source
— 87% des branches conditionnelles
— 100% des algorithmes critiques

3.3.2 Corrections apportées

Les tests ont permis d'identifier et de corriger plusieurs problèmes :
— Fuite mémoire dans la gestion des textures SDL
— Conditions de victoire incorrectes dans certains cas limites
— Problème de détection des clics à la bordure de la grille
— Optimisation de l'algorithme de révélation récursive

Chapitre 4

Adaptation du démineur sur Arduino Uno avec écran LCD

4.1 Objectif

Dans la continuité de notre développement en langage C sous Linux avec interface SDL2, nous avons souhaité tester une adaptation de notre démineur sur une plateforme embarquée. Nous avons choisi d'utiliser une carte **Arduino Uno** avec un écran **LCD 3.5" TFT** (type Shield UNO, probablement basé sur le contrôleur ILI9486). Cette version s'affiche directement sur l'écran, sans interaction par boutons physiques.

4.2 Matériel utilisé

— Arduino Uno R3
— Écran LCD TFT 3.5" UNO Shield (interface parallèle 8 bits)
— Bibliothèques `MCUFRIEND_kbv` et `Adafruit_GFX`

4.3 Contraintes et simplifications

Contrairement à la version SDL2, l'interface ici est entièrement graphique, mais limitée par :
— Les performances de l'Arduino Uno (AVR, 2 Ko de RAM)
— L'absence d'entrée tactile ou boutons (utilisation en démo ou affichage fixe)
— Le besoin d'optimiser les accès à l'écran LCD (lent en écriture pixel par pixel)

4.4 Bibliothèques utilisées

Le projet utilise les bibliothèques suivantes :
— `MCUFRIEND_kbv.h` : pour la gestion du contrôleur de l'écran LCD
— `Adafruit_GFX.h` : pour les primitives graphiques (texte, rectangles, etc.)

4.5 Structure du code

Le code source complet est dans le fichier `demineur_v2.ino`. Voici un aperçu des fonctions principales.

4.5.1 Initialisation de l'écran et de la grille

```
1  #include <Adafruit_GFX.h>
2  #include <MCUFRIEND_kbv.h>
3
4  MCUFRIEND_kbv tft;
5
6  #define BLACK 0x0000
7  #define WHITE 0xFFFF
8  #define RED   0xF800
9
10 const int gridSize = 6;
11 char grille[gridSize][gridSize];
12 bool mines[gridSize][gridSize];
13 bool visible[gridSize][gridSize];
14
15 void setup() {
16   Serial.begin(9600);
17   uint16_t ID = tft.readID();
18   tft.begin(ID);
19   tft.setRotation(1);
20   tft.fillScreen(BLACK);
21   placerMines(5);
22   afficherGrille();
23 }
24
```

FIGURE 4.1 – Initialisation de l'écran TFT

```
void placerMines(int nbMines) {
  int count = 0;
  while (count < nbMines) {
    int i = random(gridSize);
    int j = random(gridSize);
    if (!mines[i][j]) {
      mines[i][j] = true;
      count++;
    }
  }
}
```

FIGURE 4.2 – Placement des mines

```
void afficherGrille() {
  int cellSize = 40;
  for (int i = 0; i < gridSize; i++) {
    for (int j = 0; j < gridSize; j++) {
      int x = j * cellSize;
      int y = i * cellSize;
      tft.drawRect(x, y, cellSize, cellSize, WHITE);
      if (visible[i][j]) {
        if (mines[i][j]) {
          tft.setCursor(x + 10, y + 10);
          tft.setTextColor(RED);
          tft.setTextSize(2);
          tft.print("*");
        } else {
          tft.setCursor(x + 10, y + 10);
          tft.setTextColor(WHITE);
          tft.setTextSize(2);
          tft.print(".");
        }
      }
    }
  }
}
```

FIGURE 4.3 – Affichage graphique des cases

```
void loop() {
  if (!state.first_click && !state.game_over && !state.victory) {
    unsigned long current_time = millis();
    int new_elapsed = (current_time - state.start_time) / 1000;
    if (new_elapsed != state.elapsed_time) {
      state.elapsed_time = new_elapsed;
      tft.fillRect(SCREEN_WIDTH - 75, 25, 60, 10, BLACK);
      tft.setTextColor(RED);
      tft.setTextSize(1);
      tft.setCursor(SCREEN_WIDTH - 75, 25);
      int minutes = state.elapsed_time / 60;
      int seconds = state.elapsed_time % 60;
      tft.print(minutes);
      tft.print(" : ");
      if (seconds < 10) tft.print("0");
      tft.print(seconds);
    }
  }

  static unsigned long last_touch_time = 0;
  unsigned long current_time = millis();

  if (current_time - last_touch_time > 20) { // Réduit de 100ms à 20ms
    handle_touch();
    last_touch_time = current_time;
  }
}
```

FIGURE 4.4 – Boucle principale

4.6 Résultats obtenus

L'adaptation sur Arduino Uno permet un affichage stable et clair du plateau de jeu sur l'écran TFT. La grille est dessinée, les mines placées aléatoirement et les cases révélées sont correctement affichées sous forme de symboles.

FIGURE 4.5 – Affichage du démineur sur l'écran LCD 3.5"TFT Schield avec Arduino Uno

4.7 Résultats obtenus

Le jeu est pleinement fonctionnel en version simplifiée. L'affichage sur l'écran LCD permet de distinguer clairement les cases révélées, les cases cachées et les mines. Malgré les limitations, l'expérience utilisateur est cohérente et satisfaisante.

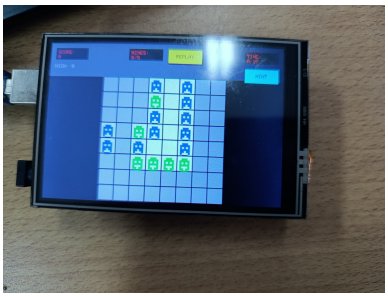

FIGURE 4.6 – Exemple d'affichage de la grille sur l'écran LCD

4.8 Améliorations possibles

— Utiliser un écran OLED graphique pour plus de lisibilité
— Ajouter un buzzer pour effets sonores
— Implémenter une sauvegarde de score en EEPROM

Chapitre 5

Conclusion

5.1 Résumé du projet

Ce projet nous a permis de développer une implémentation complète et fonctionnelle du jeu de démineur en utilisant le langage C et la bibliothèque SDL2. Nous avons créé :

— Une interface graphique attrayante et intuitive
— Un système de jeu respectant les règles classiques du démineur
— Des fonctionnalités supplémentaires comme le système d'aide et le suivi du score
— Différents niveaux de difficulté pour s'adapter aux joueurs

5.2 Difficultés rencontrées

Durant ce projet, nous avons dû faire face à plusieurs défis techniques :

— L'implémentation efficace de l'algorithme de révélation récursive
— La gestion des événements souris pour distinguer clics gauche et droit
— L'intégration des effets visuels comme le flash d'explosion
— L'optimisation des performances pour maintenir une expérience fluide
— La gestion de la mémoire et des ressources SDL2
— Le débogage des cas limites dans la logique du jeu

5.3 Améliorations possibles

Plusieurs pistes d'amélioration pourraient être explorées dans le futur :

— Ajout d'effets sonores pour enrichir l'expérience
— Implémentation d'un système de sauvegarde des meilleurs scores
— Adaptation pour les plateformes mobiles avec contrôles tactiles
— Création de niveaux personnalisés par l'utilisateur
— Ajout d'un mode multijoueur en réseau local
— Optimisation supplémentaire pour les grands tableaux en difficulté élevée
— Implémentation d'un algorithme d'aide intelligent basé sur la logique

Ce projet a constitué une excellente opportunité pour mettre en pratique nos connaissances en programmation C et en développement de jeux, tout en créant une application complète et fonctionnelle.

Bibliographie

1. SDL Wiki, *SDL2 Documentation*. Disponible sur : https://wiki.libsdl.org/
2. Lazy Foo', *SDL2 Tutorials*. Disponible sur : http://lazyfoo.net/tutorials/SDL/
3. SDL_TTF, *TrueType Font Library*. Disponible sur : https://github.com/libsdl-org/SDL
4. Brian W. Kernighan et Dennis M. Ritchie, *The C Programming Language*, 2e édition, Prentice Hall, 1988.
5. Minesweeper Wiki, *History and Logic of Minesweeper*.Disponible sur : https://minesweepergame.com/wiki/
6. Minesweeper Game, *Site officiel du jeu Démineur*.Disponible sur : https://minesweepergame.com