

PROJET

DEVELOPPEMENT D'UN JEU DE CASSE- TETE DE STYLE LABYRINTHE (MAZES) EN C++ AVEC RAYLIB



Réalisé par : Siham ELHISSANI

Imran CHARKAOUI

Encadré par Ikram BENABDELOUAHAB

1. Introduction :

Ce projet consiste à développer un jeu de labyrinthe généré de manière procédurale. L'objectif est de permettre au joueur de se déplacer dans un labyrinthe pour atteindre la sortie tout en respectant un chronomètre. Ce jeu a été réalisé dans le cadre de l'apprentissage de la programmation orientée objet en C++.

L'objectif principal est d'appliquer les concepts de programmation orientée objet (POO) en créant un jeu interactif avec plusieurs niveaux de difficulté. Le défi réside dans la génération dynamique de labyrinthes, l'optimisation des performances et la gestion des interactions avec le joueur.

2. Outils et technologies utilisées :

- **Langage de programmation :** C++
- **Bibliothèques utilisées :** Raylib (pour le rendu graphique 2D)
- **Environnement de développement :** Visual Studio Code

3. Etude technique :

3.1. Classes clés :

3.1.1. Labyrinthe :

➤ Sa description :

La classe Labyrinthe représente un labyrinthe généré procéduralement. Elle utilise une grille 2D pour modéliser les murs, les passages, et la sortie. Chaque cellule de cette grille est représentée par un entier, où les murs sont marqués par 1, les passages libres par 0, et la sortie par 2. Les dimensions du labyrinthe sont définies par les attributs largeur et hauteur, tandis que les coordonnées de la sortie sont stockées dans *finishX* et *finishY*.

➤ La classe de Labyrinthe :

```
src > C Labyrinthe.h > ...
1  #ifndef LABYRINTHE_H
2  #define LABYRINTHE_H
3
4  #include <vector>
5  #include <stack>
6  #include <cstdlib>
7  #include <ctime>
8  #include <utility>
9
10 class Labyrinthe {
11 public:
12     int largeur, hauteur;
13     std::vector<std::vector<int>> grille;
14     int finishx, finishy;
15
16     Labyrinthe(int l, int h);
17     void genererLabyrinthe();
18     void setFinishPoint(int x, int y);
19 };
20
21 #endif // LABYRINTHE_H
```

Les méthodes principales :

✓ La méthode genererLabyrinthe :

➤ Sa description :

MST : Sécurité It et Big Data

Algorithmique avancée et programmation

Lorsqu'un objet *Labyrinthe* est créé, la grille est initialisée avec des murs partout, et la méthode *genererLabyrinthe* est appelée pour générer un labyrinthe aléatoire en utilisant un algorithme de *backtracking*. Cet algorithme part d'une cellule de départ et creuse des passages en avançant vers des voisins disponibles, tout en supprimant les murs entre les cellules. Une pile est utilisée pour revenir en arrière lorsqu'aucune avancée n'est possible, garantissant ainsi que le labyrinthe est entièrement connecté.

➤ **Le code correspond :**

```

9 void Labyrinthe::genererLabyrinthe() {
10     int x = 1, y = 1;
11     grille[y][x] = 0;
12
13     std::stack<std::pair<int, int>> pile;
14     pile.push({x, y});
15
16     std::vector<std::pair<int, int>> directions = {{2, 0}, {0, 2}, {-2, 0}, {0, -2}};
17
18     while (!pile.empty()) {
19         auto current = pile.top();
20         int cx = current.first;
21         int cy = current.second;
22         std::vector<std::pair<int, int>> voisins;
23
24         for (auto& dir : directions) {
25             int nx = cx + dir.first;
26             int ny = cy + dir.second;
27             if (nx > 0 && ny > 0 && nx < largeur - 1 && ny < hauteur - 1 && grille[ny][nx] == 1) {
28                 voisins.push_back({nx, ny});
29             }
30         }
31
32         if (!voisins.empty()) {
33             auto next = voisins[rand() % voisins.size()];
34             int nx = next.first;
35             int ny = next.second;
36             grille[cy + (ny - cy) / 2][cx + (nx - cx) / 2] = 0;
37             grille[ny][nx] = 0;
38             pile.push(next);
39         } else {
40             pile.pop();
41         }
42     }
43
44     setFinishPoint(largeur - 2, hauteur - 2);
45 }

```

✓ La methode `setFinishPoint` :

➤ **Sa description :**

Cette méthode permet de définir la sortie du labyrinthe à une position spécifique, généralement proche de l'extrémité opposée à l'entrée.

➤ **Le code correspond :**

```
47 void Labyrinthe::setFinishPoint(int x, int y) {
48     finishX = x;
49     finishY = y;
50     grille[y][x] = 2;
51 }
```

```

3
4 GameScreen Joueur::Update(Labyrinthe& labyrinthe) {
5     const int speedMultiplier = 2;
6
7
8     if (IsKeyDown(KEY_RIGHT)) {
9         for (int i = 0; i < speedMultiplier; i++) {
10             deplacer('R', labyrinthe);
11         }
12     }
13
14     if (IsKeyDown(KEY_LEFT)) {
15         for (int i = 0; i < speedMultiplier; i++) {
16             deplacer('L', labyrinthe);
17         }
18     }
19
20     if (IsKeyDown(KEY_DOWN)) {
21         for (int i = 0; i < speedMultiplier; i++) {
22             deplacer('D', labyrinthe);
23         }
24     }
25
26     if (IsKeyDown(KEY_UP)) {
27         for (int i = 0; i < speedMultiplier; i++) {
28             deplacer('U', labyrinthe);
29         }
30     }
31
32     return SOLOCONTROLS;
33 }

```

➤ **Le code de la classe Difficulty :**

```

1  #ifndef DIFFICULTY_H
2  #define DIFFICULTY_H
3
4  #include "raylib.h"
5  #include "game.h"
6
7
8  class Difficulty {
9  public:
10     Difficulty();
11     ~Difficulty();
12
13     void Difficulty::Draw()
14     void Draw();
15
16 private:
17     Rectangle easyButton;
18     Rectangle mediumButton;
19     Rectangle hardButton;
20     Rectangle returnButton;
21
22     Texture2D easyTexture;
23     Texture2D mediumTexture;
24     Texture2D hardTexture;
25     Texture2D returnTexture;
26 };
27
28 #endif // DIFFICULTY_H
    
```

Les méthodes principales :

✓ La méthode `Difficulty::Difficulty()` :

➤ Sa description :

Le constructeur de la classe *Difficulty* initialise les textures des boutons pour les différents niveaux de difficulté et le bouton de retour. Les textures sont chargées à partir de fichiers d'images spécifiés. Ensuite, il définit les dimensions et la position des boutons à l'écran en fonction de la taille des images des boutons. Les boutons sont positionnés au centre de l'écran avec un espacement fixe entre eux. Les positions sont calculées pour que les boutons soient disposés de manière verticale.

➤ Le code correspond :

```

3  Difficulty::Difficulty() {
4      easyTexture = LoadTexture("buttons/butto/EASY.png");
5      mediumTexture = LoadTexture("buttons/butto/MEDIUM.png");
6      hardTexture = LoadTexture("buttons/butto/HARD.png");
7      returnTexture = LoadTexture("buttons/butto/Back2.png");
8
9
10     float buttonWidth = easyTexture.width;
11     float buttonHeight = easyTexture.height;
12     float spacing = 20;
13
14     easyButton = { (GetScreenWidth() - buttonWidth) / 2, GetScreenHeight() - (buttonHeight * 4 + spacing * 3) - 100, buttonWidth, buttonHeight };
15     mediumButton = { (GetScreenWidth() - buttonWidth) / 2, easyButton.y + buttonHeight + spacing, buttonWidth, buttonHeight };
16     hardButton = { (GetScreenWidth() - buttonWidth) / 2, mediumButton.y + buttonHeight + spacing, buttonWidth, buttonHeight };
17     returnButton = { (GetScreenWidth() - buttonWidth) / 2, hardButton.y + buttonHeight + spacing, buttonWidth, buttonHeight };
18 }
19
20 Difficulty::~Difficulty() {
21     UnloadTexture(easyTexture);
22     UnloadTexture(mediumTexture);
23     UnloadTexture(hardTexture);
24     UnloadTexture(returnTexture);
25 }
    
```

✓ La methode `Difficulty::~~Difficulty()` :

➤ Sa description :

Le destructeur de la classe *Difficulty* libère la mémoire associée aux textures des boutons

MST : Sécurité It et Big Data

Algorithmique avancée et programmation

chargées précédemment. Il utilise la fonction *UnloadTexture* pour chaque texture afin de s'assurer que les ressources graphiques sont correctement libérées lors de la fermeture de l'écran de sélection de difficulté.

➤ Le code correspond :

```
20 Difficulty::~Difficulty() {
21     UnloadTexture(easyTexture);
22     UnloadTexture(mediumTexture);
23     UnloadTexture(hardTexture);
24     UnloadTexture(returnTexture);
25 }
```

✓ La methode **GameScreen Difficulty::Update()** :

➤ Sa description :

La méthode *Update* vérifie l'interaction de l'utilisateur avec les boutons en fonction de la position de la souris. Elle écoute l'événement de clic gauche de la souris, et lorsque l'utilisateur clique sur un bouton, elle vérifie si la souris est située à l'intérieur de la zone du bouton concerné à l'aide de la fonction *CheckCollisionPointRec*. En fonction du bouton cliqué, elle retourne l'état correspondant à un mode de jeu spécifique, comme *EASYSOLOMODE*, *MEDIUMSOLOMODE*, *HARDSOLOMODE*, ou *MENU* si le bouton de retour est cliqué. Si aucun bouton n'est cliqué, l'état du jeu reste sur l'écran de sélection de difficulté.

➤ Le code correspond :

```
27 GameScreen Difficulty::Update() {
28     if (IsMouseButtonPressed(MOUSE_BUTTON_LEFT)) {
29         Vector2 mousePos = GetMousePosition();
30
31         if (CheckCollisionPointRec(mousePos, easyButton)) return EASYSOLOMODE;
32         if (CheckCollisionPointRec(mousePos, mediumButton)) return MEDIUMSOLOMODE;
33         if (CheckCollisionPointRec(mousePos, hardButton)) return HARDSOLOMODE;
34         if (CheckCollisionPointRec(mousePos, returnButton)) return MENU;
35     }
36
37     return DIFFICULTY;
38 }
```

✓ Méthode **void Difficulty::Draw()** :

➤ Sa description :

La méthode *Draw* est responsable de l'affichage des boutons à l'écran. Elle utilise la fonction *DrawTextureRec* pour dessiner chaque texture de bouton à l'endroit calculé dans le constructeur. Chaque bouton est dessiné avec sa texture respective (facile, moyen, difficile et retour) à sa position définie, en utilisant la couleur blanche pour l'affichage. Cette méthode permet de rendre visuellement les boutons interactifs à l'écran afin que l'utilisateur puisse les voir et interagir avec eux.

➤ Le code correspond :

```
40 void Difficulty::Draw() {
41     DrawTextureRec(easyTexture, { 0, 0, (float)easyTexture.width, (float)easyTexture.height }, { easyButton.x, easyButton.y }, WHITE);
42     DrawTextureRec(mediumTexture, { 0, 0, (float)mediumTexture.width, (float)mediumTexture.height }, { mediumButton.x, mediumButton.y }, WHITE);
43     DrawTextureRec(hardTexture, { 0, 0, (float)hardTexture.width, (float)hardTexture.height }, { hardButton.x, hardButton.y }, WHITE);
44     DrawTextureRec(returnTexture, { 0, 0, (float)returnTexture.width, (float)returnTexture.height }, { returnButton.x, returnButton.y }, WHITE);
45 }
```

3.1.1. Menu :

➤ Sa description :

La classe *Menu* gère l'écran principal du menu du jeu, où l'utilisateur peut choisir entre différents modes, accéder aux paramètres ou quitter le jeu. Elle contient des boutons interactifs qui permettent à l'utilisateur de naviguer entre les différentes options. Cette classe utilise la bibliothèque *Raylib* pour l'affichage et la gestion des événements.

MST : Sécurité It et Big Data

Algorithmique avancée et programmation

➤ **Le code de la Classe menu :**

```
1 #ifndef MENU_H
2 #define MENU_H
3
4 #include "raylib.h"
5 #include "game.h"
6
7 class Menu {
8 public:
9     Menu();
10    ~Menu();
11
12    GameScreen Update();
13    void Draw();
14
15 private:
16    Rectangle soloButton;
17    Rectangle settingsButton;
18    Rectangle exitButton;
19
20    Texture2D soloTexture;
21    Texture2D settingsTexture;
22    Texture2D exitTexture;
23
24 };
25
26 #endif // MENU_H
```

Les méthodes principaux :

- ✓ **Méthode Menu::Menu() (Constructeur) :**

➤ **Sa description :**

Le constructeur de la classe *Menu* charge les textures des boutons du menu à partir de fichiers d'images. Ces boutons incluent ceux pour le mode solo, les paramètres et la sortie du jeu. Ensuite, il définit les dimensions et la position des boutons en fonction de la taille des textures et de l'écran. Les boutons sont centrés horizontalement et disposés verticalement avec un espacement entre eux. Le calcul des positions permet de placer les boutons de manière lisible et équilibrée sur l'écran.

➤ **Le code correspond :**

```
3 Menu::Menu() {
4
5     soloTexture = LoadTexture("buttons/butto/SOLO.png");
6     settingsTexture = LoadTexture("buttons/butto/SETTINGS.png");
7     exitTexture = LoadTexture("buttons/butto/exit.png");
8
9     float buttonWidth = soloTexture.width;
10    float buttonHeight = soloTexture.height;
11    float spacing = 20;
12
13    soloButton = { (GetScreenWidth() - buttonWidth) / 2, GetScreenHeight() - (buttonHeight * 4 + spacing * 3) - 100, buttonWidth, buttonHeight };
14    settingsButton = { (GetScreenWidth() - buttonWidth) / 2, soloButton.y + buttonHeight + spacing, buttonWidth, buttonHeight };
15    exitButton = { (GetScreenWidth() - buttonWidth) / 2, settingsButton.y + buttonHeight + spacing, buttonWidth, buttonHeight };
16 }
```

- ✓ La méthode ~Menu() (Destructeur) :

➤ **Sa description :**

Le destructeur de la classe `Menu` libère les ressources associées aux textures des boutons chargées dans le constructeur. Il utilise `UnloadTexture` pour chaque texture afin de s'assurer qu'aucune mémoire n'est laissée inutilisée une fois que le menu est fermé.

➤ **Le code correspond :**

MST : Sécurité It et Big Data

Algorithmique avancée et programmation

```

18 Menu::~Menu() {
19
20     UnloadTexture(soloTexture);
21     UnloadTexture(settingsTexture);
22     UnloadTexture(exitTexture);
23 }
    
```

✓ La méthode `GameScreen Menu::Update()` :

➤ Sa description :

La méthode `Update` vérifie les interactions de l'utilisateur avec les boutons du menu principal. Lorsqu'un clic de souris est détecté (bouton gauche), elle récupère la position de la souris et vérifie si elle se trouve à l'intérieur de l'une des zones cliquables (rectangles des boutons). Si la souris clique sur le bouton "*Solo*", le jeu passe à l'écran de sélection de la difficulté (*DIFFICULTY*). Si le bouton "*Settings*" est cliqué, le jeu accède à l'écran des paramètres (*SETTINGS*). Enfin, si le bouton "Exit" est cliqué, la fenêtre du jeu se ferme en appelant `CloseWindow()`. Si aucune interaction n'est détectée, l'état du jeu reste sur l'écran du menu principal.

➤ Le code correspond :

```

24
25 GameScreen Menu::Update() {
26     if (IsMouseButtonPressed(MOUSE_BUTTON_LEFT)) {
27         Vector2 mousePos = GetMousePosition();
28
29         if (CheckCollisionPointRec(mousePos, soloButton)) return DIFFICULTY;
30         if (CheckCollisionPointRec(mousePos, settingsButton)) return SETTINGS;
31         if (CheckCollisionPointRec(mousePos, exitButton)) CloseWindow();
32     }
33
34     return MENU;
35 }
    
```

✓ Méthode `void Menu::Draw()` :

➤ Sa description :

La méthode `Draw` est responsable de l'affichage graphique des boutons du menu à l'écran. Elle utilise la fonction `DrawTextureRec` pour dessiner chaque bouton (solo, paramètres, quitter) à sa position respective. Chaque bouton est rendu avec sa texture chargée précédemment, affichée en couleur blanche sur l'écran. Cette méthode permet de visualiser les boutons interactifs et de les rendre cliquables.

➤ Le code correspond :

```

37 void Menu::Draw() {
38     DrawTextureRec(soloTexture, { 0, 0, (float)soloTexture.width, (float)soloTexture.height }, { soloButton.x, soloButton.y }, WHITE);
39     DrawTextureRec(settingsTexture, { 0, 0, (float)settingsTexture.width, (float)settingsTexture.height }, { settingsButton.x, settingsButton.y }, WHITE);
40     DrawTextureRec(exitTexture, { 0, 0, (float)exitTexture.width, (float)exitTexture.height }, { exitButton.x, exitButton.y }, WHITE);
41 }
    
```

3.1.2. Settings :

➤ Sa description :

Le constructeur de la classe `Settings` initialise les textures des boutons pour chaque option de réglage, comme le retour au menu, le volume, la résolution, et le choix du joueur. Il charge les images correspondantes depuis les fichiers spécifiés. Ensuite, il définit les rectangles représentant les zones cliquables des boutons, en les positionnant de manière logique à l'écran avec un espacement adéquat. Le calcul des positions permet de garantir que les boutons sont bien disposés pour une navigation facile.

➤ La Classe de Setting :

```
src > C settings.h > Settings > backButton
1 #ifndef SETTINGS_H
2 #define SETTINGS_H
3
4 #include "raylib.h"
5 #include "game.h"
6
7 class Settings {
8 public:
9     Settings();
10    ~Settings();
11    GameScreen Update();
12    void Draw();
13
14 private:
15     Rectangle backButton;
16     Rectangle volumeButton;
17     Rectangle resolutionButton;
18     Rectangle player1Button;
19
20     Texture2D backTexture;
21     Texture2D volumeTexture;
22     Texture2D resolutionTexture;
23     Texture2D player1Texture;
24 };
25
26 #endif // SETTINGS_H
```

Les méthodes principales :

✓ Méthode Settings::Settings() (Constructeur) :

➤ Sa description :

Le constructeur de la classe Settings initialise les textures des boutons du menu des paramètres, qui incluent les boutons pour revenir au menu principal, ajuster le volume, modifier la résolution et accéder aux contrôles du joueur. Il charge les images de chaque bouton à partir des fichiers spécifiés. Ensuite, il définit les rectangles représentant les zones cliquables pour chaque bouton, en les disposant verticalement sur l'écran avec un espacement fixe. La position initiale du premier bouton (pour les contrôles du joueur) est définie à 600 pixels en Y, et les autres boutons sont placés en dessous, à une distance déterminée par la taille des boutons et l'espacement.

➤ Le code correspond :

```
3 Settings::Settings() {
4     backTexture = LoadTexture("buttons/butto/Back2 - Copie.png");
5     volumeTexture = LoadTexture("buttons/butto/volume.png");
6     resolutionTexture = LoadTexture("buttons/butto/res.png");
7     player1Texture = LoadTexture("buttons/butto/control.png");
8
9     float buttonWidth = volumeTexture.width;
10    float buttonHeight = volumeTexture.height;
11    float spacing = 20;
12
13    float startY = 600;
14
15    player1Button = { (GetScreenWidth() - buttonWidth) / 2, startY, buttonWidth, buttonHeight };
16    volumeButton = { player1Button.x, player1Button.y + buttonHeight + spacing, buttonWidth, buttonHeight };
17    resolutionButton = { volumeButton.x, volumeButton.y + buttonHeight + spacing, buttonWidth, buttonHeight };
18    backButton = { resolutionButton.x, resolutionButton.y + buttonHeight + spacing, buttonWidth, buttonHeight };
19 }
```

✓ Méthode ~Settings() (Destructeur) :

➤ Sa description :

Le destructeur de la classe Settings libère les ressources associées aux textures des boutons qui ont été chargées en mémoire. Il appelle la fonction *UnloadTexture* pour chaque texture afin de s'assurer que la mémoire est libérée de manière propre et efficace lorsqu'on quitte l'écran des paramètres.

Ce code gère le flux principal d'un jeu avec plusieurs écrans interactifs. Il initialise les ressources, charge les éléments graphiques et audio, met à jour l'état du jeu selon les interactions de l'utilisateur, et gère la transition entre différents écrans du jeu, comme le menu principal, les paramètres, et les différents modes de jeu. Après chaque interaction,

MST : S curit  It et Big Data

Algorithmique avanc e et programmation

le jeu redessine l' cran et met   jour l'audio, assurant une exp rience fluide pour l'utilisateur.

➤ Initialisation du jeu :

- Fen tre du jeu : La fen tre du jeu est initialis e avec une taille de *1920x1080* pixels et un titre "*Maze*". La fonction *InitWindow()* cr e la fen tre, et *InitAudioDevice()* initialise le syst me audio du jeu.
- Musique et arri re-plan : L'arri re-plan du jeu est charg    partir de l'image "*images/Background.jpg*". Si l'image n'est pas trouv e (les dimensions sont nulles), le programme se termine. Ensuite, un flux musical est charg    partir du fichier "*Sounds/music.mp3*", et si le flux  choue, le programme lib re les ressources et se termine. La musique est jou e en boucle et son volume est r gl    *0.5f*.
- Police personnalis e : La police "*Font/monogram.ttf*" est charg e pour  tre utilis e dans l'interface utilisateur, si n cessaire.

➤ Initialisation des objets du jeu :

Des objets repr sentant diff rents  crans du jeu sont cr  s, incluant :

- **Menu** : Un objet de la classe *Menu* pour g rer l' cran principal.
- **Settings** : Un objet de la classe *Settings* pour g rer les options du jeu (volume, r solution, etc.).
- **Difficulty** : Un objet de la classe *Difficulty* pour g rer la s lection de la difficult  du jeu.
- **Solocontrols, Easysolomode, Mediumsolomode, Hardsolomode** : Des objets qui repr sentent les diff rents modes du jeu.
- **Volume, Resolution** : Des objets pour g rer respectivement les r glages du volume et de la r solution.

➤ Boucle principale du jeu :

La boucle principale (*while (!WindowShouldClose())*) continue de s'ex cuter tant que l'utilisateur n'a pas ferm  la fen tre du jeu. Elle comprend plusieurs  tapes :

- **Mise   jour de la musique** : La fonction *UpdateMusicStream(music)* est appel e   chaque it ration pour mettre   jour le flux audio. Si la musique a atteint sa fin, elle red marre automatiquement pour jouer en boucle.
- **Gestion de l' cran actuel** : La variable *currentScreen* d termine quel  cran doit  tre affich  et mis   jour. En fonction de l' tat actuel, l'un des objets correspondants (comme *menu*, *settings*, *difficulty*, etc.) est mis   jour via la m thode *Update()*. Cette m thode g re les interactions utilisateur, telles que les clics sur les boutons, et permet de changer d' cran.

➤ Rendu graphique :

- D but du dessin : La fonction *BeginDrawing()* commence le processus de dessin dans la fen tre. Ensuite, l'arri re-plan est effac  et remplac  par une couleur blanche, suivi du dessin de l'image d'arri re-plan sur toute la fen tre   l'aide de la fonction *DrawTexture(background, 0, 0, WHITE)*.

MST : Sécurité It et Big Data

Algorithmique avancée et programmation

- Affichage de l'écran actuel : Selon la valeur de *currentScreen*, l'écran approprié est dessiné en appelant la méthode *Draw()* de l'objet correspondant. Cela permet de rendre l'interface graphique et les éléments interactifs comme les boutons, les options de menu, les modes de jeu, etc.
- Fin du dessin : La fonction *EndDrawing()* termine le processus de dessin.

➤ Libération des ressources :

Avant de quitter le jeu, le programme libère toutes les ressources qui ont été utilisées pendant le jeu :

- **Textures** : L'arrière-plan est déchargé via *UnloadTexture(background)*.
- **Flux audio** : Le flux musical est déchargé via *UnloadMusicStream(music)*.
- **Police** : La police personnalisée est déchargée via *UnloadFont(customFont)*.

Finalement, la fonction *CloseWindow()* ferme la fenêtre du jeu.

1.1. Captures d'écrans



Menu de jeu.



a



Manuel de jouabilité.



Labyrinthe : Niveau facile.

MST : Sécurité It et Big Data
Algorithmique avancée et programmation



Labyrinthe : Niveau moyen.



Labyrinthe : Niveau difficile.

2. Conclusion

Le programme principal du jeu gère l'ensemble du flux de jeu, en permettant une interaction fluide entre différents écrans (menu, paramètres, modes de jeu, etc.) et en assurant la gestion continue des ressources multimédia comme la musique et les images. À travers une boucle principale, il gère l'état du jeu, met à jour l'affichage en temps réel, et répond aux actions de l'utilisateur. Cette structure modulaire permet de séparer clairement les différentes fonctionnalités du jeu, facilitant ainsi l'extension et la maintenance du code. Finalement, lorsque le joueur termine sa session, le programme se ferme proprement en libérant toutes les ressources utilisées.