Université : Faculté des Sciences Semlalia, Marrakech

Année universitaire : 2024-2025

Enseignants référents : Mme R. Hannane

RAPPORT DE PROJET

Jeu de Course Moto « Jamaa-el-Fnaa »

Binôme : **Oussama Samid & Ahmed Elmaki**

Rapport de Projet – Jeu de Course Moto « Jamaa-el-Fnaa

Filière : Informatique

Cadre du projet :Projet de module « Programmation Orientée C++ »

Année : 2024-2025

introduction à l'application

Ce projet vise à créer un jeu 2D de course à moto se déroulant sur la place Jamaa-el-Fnaa à Marrakech. Le joueur contrôle une moto, accélère ou freine à l'aide des flèches, change de voie pour éviter des obstacles traditionnels (jarres, autres motos, piétons stylisés). L'objectif est de couvrir une distance fixe (4000 m) le plus rapidement possible sans collision. Le jeu intègre un menu principal, un écran «Jouer, À propos,Quitter », et des commandes audio (volume, muet).

Spécification des besoins

1. Menu principal

Boutons « Jouer », « À propos », « Quitter »

Visuels personnalisés (hover, état normal)

2. Jeu

Affichage :Fond animé, piste, moto du joueur

Obstacles :Générés aléatoirement sur 3 voies, dimensionnés dynamiquement

Mécaniques : Accélération, freinage, glissade inertielle, changement de voie animé

HUD : Affichage de la distance parcourue et du chrono

Conditions :

Victoire : distance atteinte

Défaite : collision avant la fin

3. Audio

Musique de menu et de jeu

Effets sonores (clic, collision, chrono)

Contrôles volume, mute/unmute

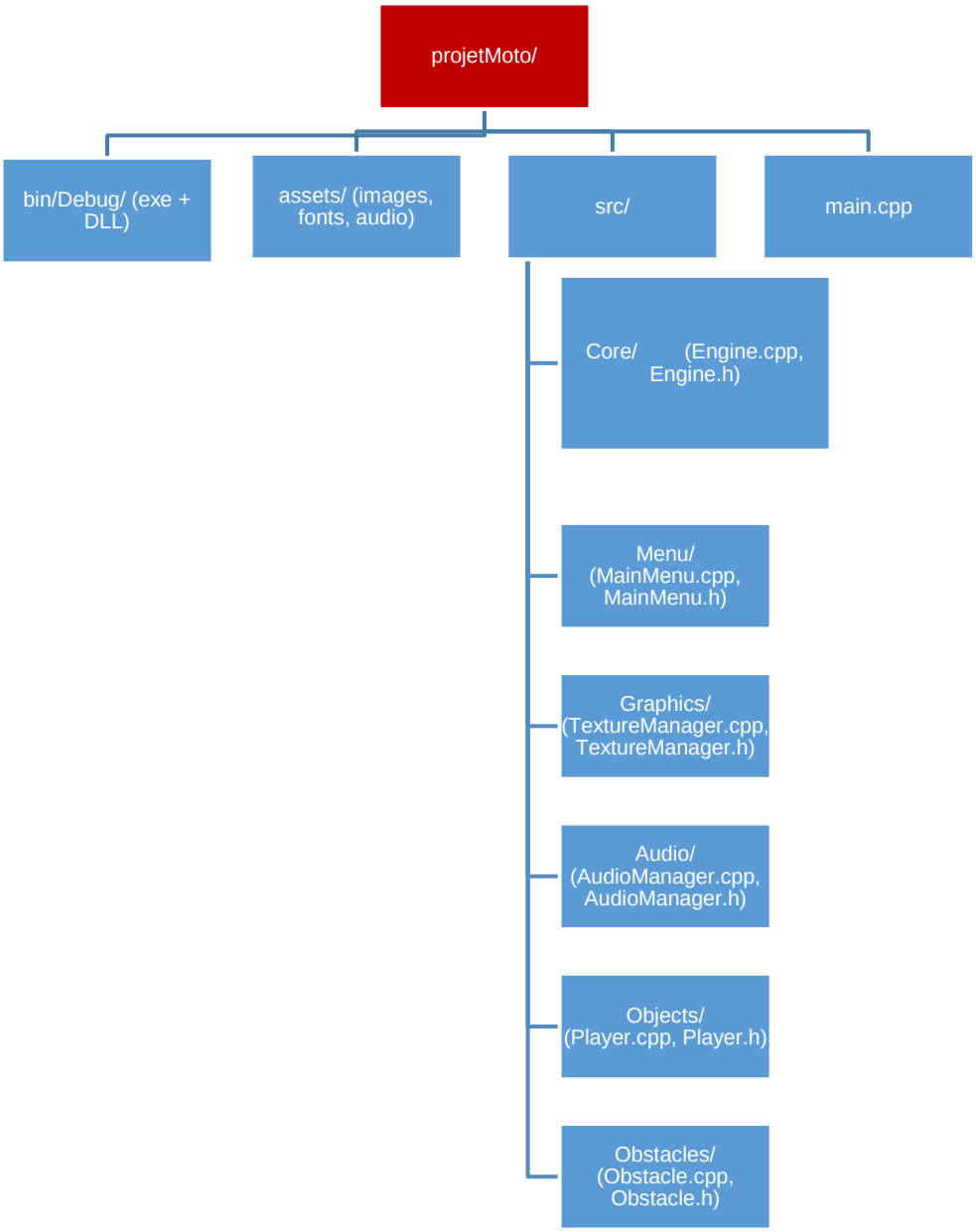
4. Techniques

Langage : C++17

Bibliothèques : SDL2, SDL_image, SDL_ttf, SDL_mixer

Compilation : MinGW-w64 (mingw32 target) sous Windows 10

Structure de projet :



Description des classes

Classe	Responsabilité
Engine	Initialisation SDL, boucle principale, état du jeu
MainMenu	Chargement du menu, gestion des événements clic/souris
TextureManager	Chargement, conservation et rendu des textures
AudioManager	Initialisation SDL_mixer, chargement et lecture son
Player	Position, mouvement, gestion de la moto
Obstacle	Données de collision et identification de texture

Chaque classe applique le principe SRP (Single Responsibility). Les managers (Texture, Audio) sont des Singletons pour partage global

Fonctionnement du jeu

1. Lancement : main.cpp appelle Engine::Init().
2. Menu: état STATE_MAIN_MENU, MainMenu::Render() affiche le fond et les boutons.
3. Démarrage : clic « Jouer » → STATE_START_SCREEN (chrono animé) → touche flèche droite → STATE_PLAYING.
4. Boucle de jeu (Engine::Update())

Lecture des entrées (SDL_PollEvent)

Player::update(dt) calcule la nouvelle vitesse et position verticale (lanes).

Défilement du fond et de la piste proportionnel à la vitesse.

SpawnObstacle() toutes les m_obstacleSpawnInterval secondes, évitant chevauchement de voie.

Collision : réduction de la zone de contact, appel Player::ApplySpeedPenalty().

Mise à jour du HUD : distance (m), chrono (s).

5Fin de partie

STATE_GAME_OVER ou STATE_WIN, affichage de l'écran correspondant, invitation à ESC ou R pour retour ou restart.

6. Nettoyage* : Engine::Clean() libère les textures, sons, fenêtre, renderer

Explication du Code

Explication du Code et des Fonctionnalités Principales du Jeu

Cette section présente une explication détaillée des composants essentiels du code du jeu. Elle met l'accent sur la logique de gameplay, la gestion des événements, les interactions utilisateur ainsi que la gestion des ressources audio et visuelles.

---

1. Gestion de l'audio

La classe `AudioManager` centralise la gestion de l'audio, assurant le chargement et la lecture appropriée de la musique et des effets sonores selon le contexte du jeu.

➤ *Contrôle du volume général :*

```
void Engine::ApplyMasterVolume() {
  if (m_isMuted) {
    AudioManager::GetInstance()->SetMusicVolume(0);
    AudioManager::GetInstance()->SetAllSoundsVolume(0);
  } else {
    AudioManager::GetInstance()-
>SetMusicVolume(m_currentMasterVolume);
    AudioManager::GetInstance()-
>SetAllSoundsVolume(m_currentMasterVolume);
  }
}
```

- `ApplyMasterVolume()` ajuste le volume global : si le jeu est en mode muet (`m_isMuted`), le volume est coupé ; sinon, il est réglé sur la valeur courante définie par l'utilisateur.
- Les méthodes `IncreaseVolume()` et `DecreaseVolume()` permettent respectivement d'augmenter ou de diminuer ce volume.

---

2. Gestion des états du jeu

La fonction `SetGameState()` permet de gérer les transitions entre les différents états du jeu (menu, partie en cours, victoire, défaite), et d'ajuster l'environnement sonore en conséquence.

➤ *Extrait de la gestion des états :*

```
void Engine::SetGameState(GameState newState) {
  if (m_gameState == newState) return;

  GameState oldState = m_gameState;
  m_gameState = newState;
  SDL_Log("Changing GameState from %d to %d", oldState,
newState);

  switch (newState) {
    case STATE_MAIN_MENU:
      if (!Mix_PlayingMusic()) {
        AudioManager::GetInstance()->PlayMusic("menu_music", -
1);
      }
      break;
    case STATE_PLAYING:
      if (oldState != STATE_START_SCREEN) {
        AudioManager::GetInstance()->StopMusic();
      }
      AudioManager::GetInstance()->PlayMusic("game_music", -
1);
      break;
    case STATE_GAME_OVER:
      AudioManager::GetInstance()->StopMusic();
      AudioManager::GetInstance()->PlaySound("lose", 0);
      break;
    case STATE_WIN:
      AudioManager::GetInstance()->StopMusic();
      AudioManager::GetInstance()->PlaySound("win", 0);
      break;
  }
}
```

- Cette méthode orchestre les transitions entre les états tels que : `STATE_MAIN_MENU`, `STATE_PLAYING`, `STATE_GAME_OVER`, et `STATE_WIN`.

- Chaque transition déclenche la lecture ou l'arrêt de pistes audio spécifiques.

---

3. Gestion du joueur – Mouvement et collisions

La classe `Player` prend en charge les déplacements de la moto, les changements de voie, la vitesse et les collisions avec les obstacles.

➤ *Mise à jour de la position et de la vitesse :*

```
void Player::update(float deltaTime) {
    if (!m_isSlowed) {
        const Uint8* keyState = SDL_GetKeyboardState(NULL);
        bool braking = keyState[SDL_SCANCODE_LEFT];

        if (braking) {
            m_speed -= m_braking * deltaTime;
        } else {
            m_speed += m_acceleration * deltaTime;
        }

        m_speed = std::max(m_minSpeed, std::min(m_speed,
m_maxSpeed));
    } else {
        m_speed = m_penaltySpeed;
    }

    if (m_numLanes > 0) {
        if (std::abs(m_currentY - m_targetY) > 0.5f) {
            float direction = (m_targetY > m_currentY) ? 1.0f
: -1.0f;
            float distanceToMove = m_laneChangeAnimSpeed *
deltaTime;
            m_currentY += direction * distanceToMove;

            if ((direction > 0 && m_currentY > m_targetY) ||
(direction < 0 && m_currentY < m_targetY)) {
                m_currentY = m_targetY;
            }
        } else {
            m_currentY = m_targetY;
        }
    }
}
```

- La vitesse est modifiée en fonction des entrées clavier : un appui sur la flèche gauche ralentit le joueur, sinon il accélère automatiquement.
- Le changement de voie est animé pour offrir une transition visuelle fluide entre les lignes.

➤ *Détection des collisions avec les obstacles :*

```
for (auto it = m_obstacles.begin(); it != m_obstacles.end() &&
!collisionProcessed; ) {
    it->collider.x -= static_cast<int>(scrollAmount);

    SDL_Rect playerCollisionBox = ...; // Calcul avec
réduction
    SDL_Rect obstacleCollisionBox = ...;

    if (SDL_HasIntersection(&playerCollisionBox,
&obstacleCollisionBox)) {
        AudioManager::GetInstance()->PlaySound("crash", 0);
        m_Player->ApplySpeedPenalty();
        it = m_obstacles.erase(it);
        collisionProcessed = true;
        break;
    }
}
```

- Les boîtes de collision sont volontairement réduites (via un *reductionFactor*) pour rendre les collisions plus précises et moins frustrantes pour le joueur.
- En cas de collision, un effet sonore est joué et la vitesse du joueur est pénalisée.

---

4. Génération des obstacles – `SpawnObstacle`

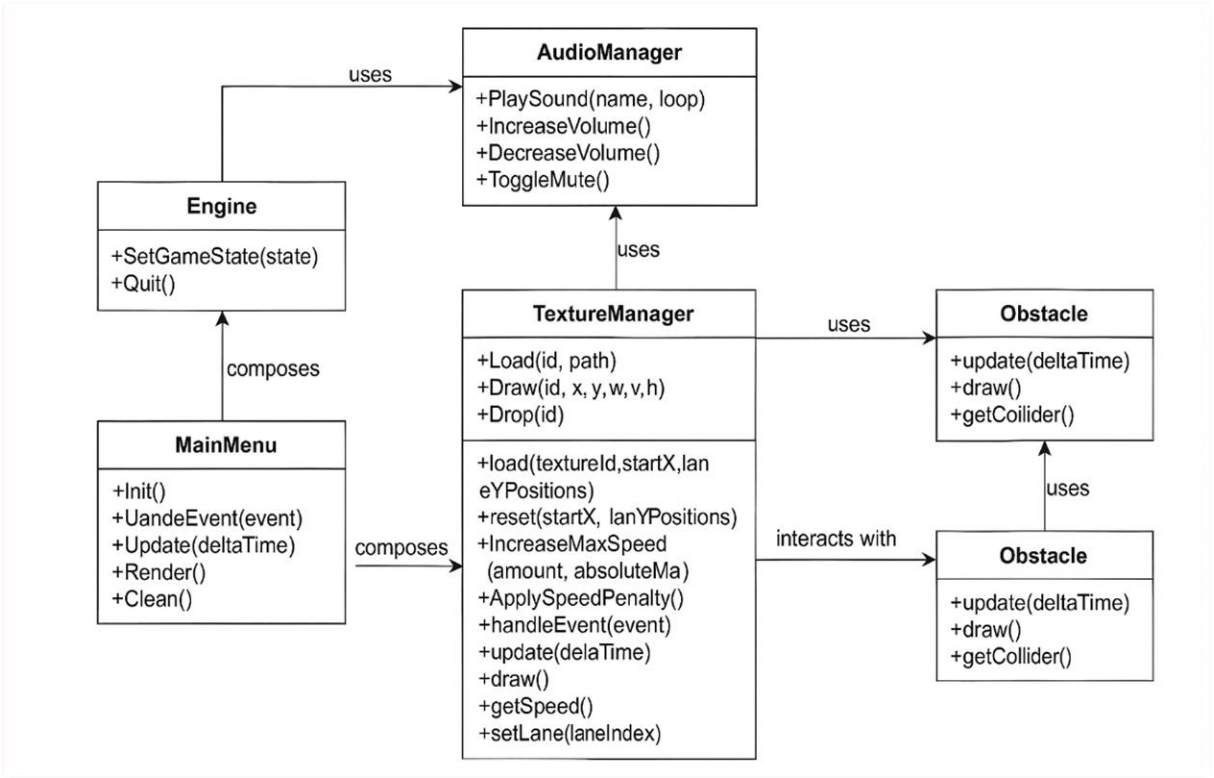
Les obstacles sont générés de façon contrôlée et aléatoire pour maintenir un bon équilibre entre difficulté et jouabilité.

➤ *Fonction de génération :*

```
void Engine::SpawnObstacle() {
    int activeCount = 0;
    for (const auto& obs : m_obstacles) {
        if (obs.isActive) ++activeCount;
    }
    const int maxOnScreen = 2;
    if (activeCount >= maxOnScreen) return;

    std::vector<int>
availableLaneIndices(m_laneYPositions.size());
    std::iota(availableLaneIndices.begin(),
availableLaneIndices.end(), 0);

    std::uniform_int_distribution<int> textureDist(0,
m_obstacleTextureIds.size() - 1);
    for (int i = 0; i < maxOnScreen - activeCount; ++i) {
        if (availableLaneIndices.empty()) break;
```

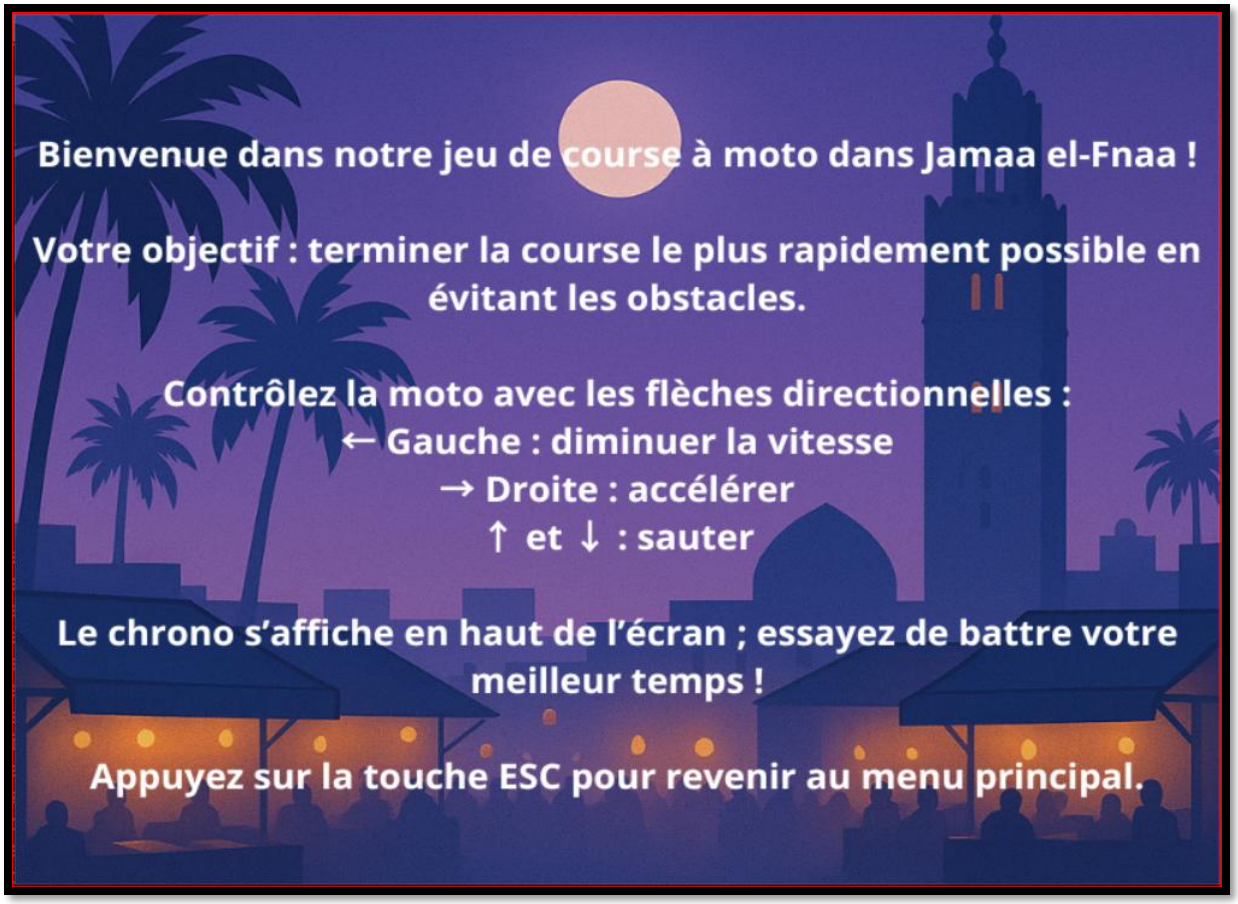
```
        std::uniform_int_distribution<int> laneDist(0,
availableLaneIndices.size() - 1);
        int laneIndex = availableLaneIndices[laneDist(m_rng)];

        Obstacle newObs;
        newObs.textureId =
m_obstacleTextureIds[textureDist(m_rng)];
        newObs.collider.x = SCREEN_WIDTH + 50;
        newObs.collider.y = m_laneYPositions[laneIndex];
        newObs.isActive = true;
        m_obstacles.push_back(newObs);


availableLaneIndices.erase(availableLaneIndices.begin() +
laneDist(m_rng));
    }
}
```

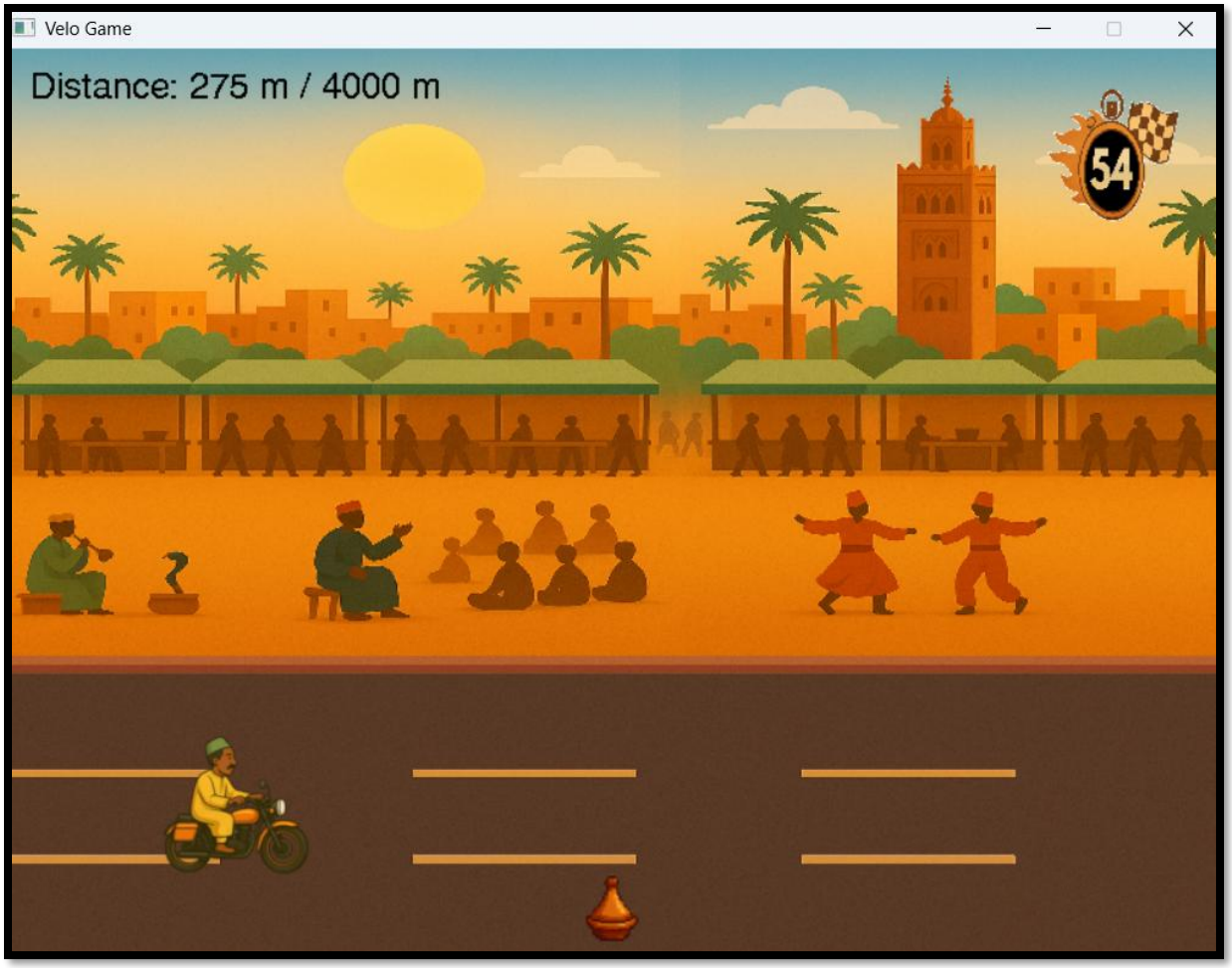
- La génération des obstacles est limitée à un maximum de deux simultanés à l'écran.
- Chaque obstacle est affecté à une voie libre choisie aléatoirement, garantissant une répartition équilibrée.



Captures d’écran


Velo Game
JEUX DE MOTO
à JEMAA EL FNA!
JOUER
À PROPOS
QUITTER


Bienvenue dans notre jeu de course à moto dans Jamaa el-Fnaa !
Votre objectif : terminer la course le plus rapidement possible en évitant les obstacles.
Contrôlez la moto avec les flèches directionnelles :
← Gauche : diminuer la vitesse
→ Droite : accélérer
↑ et ↓ : sauter
Le chrono s'affiche en haut de l'écran ; essayez de battre votre meilleur temps !
Appuyez sur la touche ESC pour revenir au menu principal.


Velo Game
Distance: 275 m / 4000 m
54


GAME OVER
LA PARTIE EST TERMINÉE, DÉPÊCHEZ-VOUS...!! ET REJOUEZ
GAME OVER

Difficultés rencontrées

Oussama et Ahmed ont rencontré un problème de linkage entre SDL2 et MinGW, notamment entre les versions 32 bits et 64 bits. Pour y remédier, ils ont aligné le compilateur en utilisant mingw32 et ont choisi les DLL compatibles 32 bits de SDL2 (notamment SDL2.dll, SDL2_image.dll, SDL2_ttf.dll, SDL2_mixer.dll, libgcc_s_seh-1.dll, libwinpthread-1.dll), toutes placées dans le dossier bin/Debug.

Ahmed a traité un souci lié au redimensionnement non proportionnel des sprites d'obstacles. Il a mis en place un calcul dynamique du ratio en utilisant SDL_QueryTexture() : la hauteur a été fixée à un sixième de la hauteur de la fenêtre, tandis que la largeur a été ajustée automatiquement en respectant le ratio largeur/hauteur d'origine (srcW/srcH).

Concernant la gestion des collisions, Oussama a constaté qu'elles étaient trop généreuses, couvrant parfois des zones non pertinentes. Il a donc réduit la taille du SDL_Rect utilisé pour la détection, en appliquant un facteur de réduction de 60 % centré sur le sprite, afin de gagner en précision.

Pour la synchronisation entre l'audio et le compte à rebours, Ahmed a utilisé SDL_GetTicks() afin d'assurer une mise à jour cohérente du chronomètre et une lecture conditionnelle des effets sonores, évitant ainsi les superpositions sonores non désirées.

Un autre problème concernait les définitions multiples de WINDOW_WIDTH et WINDOW_HEIGHT dans différents fichiers. Oussama a résolu cela en centralisant ces constantes dans un seul fichier (Engine.h) et en supprimant les redéfinitions dans les autres headers.

Ahmed a également corrigé un défaut d'affichage où le hover des boutons apparaissait tronqué. Il a modifié les textures PNG des boutons, notamment en étendant la largeur de celui de « Jouer », puis a généré une nouvelle version de l'image avec des dimensions de 817×253 pixels.

Enfin, Oussama a réglé un problème de chargement de police TTF. Il a vérifié le chemin relatif du fichier de police PixelifySans-Regular.ttf, s'assurant qu'il soit bien référencé depuis l'exécutable, et a aussi nettoyé les surfaces intermédiaires (FreeSurface) pour éviter les fuites de mémoire.

Structuration des tâches

Oussama s'est chargé de l'installation de la bibliothèque SDL et de la configuration de l'environnement MinGW. Cela a impliqué le téléchargement de la version 32 bits de SDL2, l'ajustement des variables d'environnement (notamment le PATH) ainsi que la copie manuelle des fichiers DLL nécessaires dans les répertoires d'exécution.

Ahmed a pris en main la génération des planches d'icônes destinées à l'interface graphique. Il a réalisé les designs sous Inkscape et les a exportés au format PNG avec deux résolutions spécifiques : 817×253 pixels pour les boutons principaux et 184×157 pixels pour d'autres éléments interactifs.

La structure du projet via CMake et Code::Blocks a été établie par Oussama. Il a créé le fichier projet .cbp et organisé les répertoires de manière claire, notamment src/ pour le code source et bin/ pour les exécutables.

Ahmed a implémenté le cœur du programme avec la classe Engine et la boucle principale. Il a codé les fonctions clés Engine::Init(), Run() et Clean(), ainsi que la gestion des différents états du jeu.

L'écran de menu principal et les interactions utilisateur ont été développés par Oussama. Il a assuré le chargement des textures, la détection de la position de la souris, ainsi que l'animation visuelle des boutons au survol (hover).

Ahmed a aussi travaillé sur le système de jeu lié au joueur et aux obstacles, avec la création des classes Player et Obstacle, la mise à jour des positions, la génération dynamique d'obstacles, et la gestion des collisions physiques.

Pour la gestion sonore, Oussama a mis en place un gestionnaire audio sous forme de singleton (AudioManager). Il a configuré l'ouverture du système audio via Mix_OpenAudio() et intégré des fonctions comme PlaySound() pour les effets sonores.

Ahmed a également mené des tests et du débogage liés à l'utilisation de DLL tierces, en s'aidant de l'outil *Dependency Walker* afin d'analyser les dépendances et corriger les éventuels problèmes de linkage croisé.

Enfin, la rédaction du rapport PDF a été assurée conjointement par Oussama et Ahmed. Ensemble, ils ont structuré le document, rédigé les différentes sections, et soigné la mise en page finale.

Conclusion

Le projet de jeu "Course à Moto à Jamaa-el-Fnaa" a été développé en utilisant des technologies modernes et des pratiques de développement robuste. Le jeu propose une expérience immersive, avec un gameplay stimulant, des graphismes dynamiques, des obstacles variés, et une gestion audio de haute qualité. En ajoutant des fonctionnalités comme la gestion de la musique et des effets sonores, le changement de voies animé, et la possibilité de muter le son, le jeu offre une expérience fluide et agréable.

Les perspectives d'évolution incluent l'ajout de niveaux supplémentaires avec des obstacles à thèmes variés, l'intégration d'animations via des spritesheets, et la possibilité de porter le jeu sur mobile via la compatibilité de SDL2 avec Android.