Rapport du projet

JanJump

OUJDID Mohamed CHALKHA Achraf OULADDAHMAN Ilyass

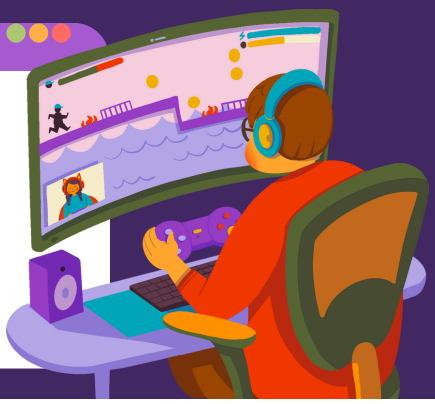


Table des matières :

Introduction

JanJump est un jeu de plateforme développé en C++ avec l'utilisation du framework Qt. Inspiré par le célèbre jeu Doodle Jump, ce projet vise à offrir une expérience similaire tout en servant d'exemple concret des concepts avancés de la programmation orientée objet (OOP) enseignés dans le cadre du cours. En recréant un jeu de plateforme dynamique et divertissant comme Doodle Jump, JanJump démontre l'application pratique de principes tels que l'encapsulation, l'héritage, le polymorphisme et la gestion des événements dans un environnement graphique interactif. Cette documentation fournit une vue d'ensemble détaillée du projet, y compris la structure du code source et les principales fonctionnalités implémentées.

Prérequis

Logiciels nécessaires :

- Qt 5.15 ou supérieur
- Un compilateur C++ (comme g++ ou clang)

Structure du projet

Répertoires et fichiers principaux :

src/ : Contient le code source du projet

- **game.h et game.cpp**
- **gamescene.h et gamescene.cpp**
- **view.h et view.cpp**

images/ : Contient les ressources graphiques

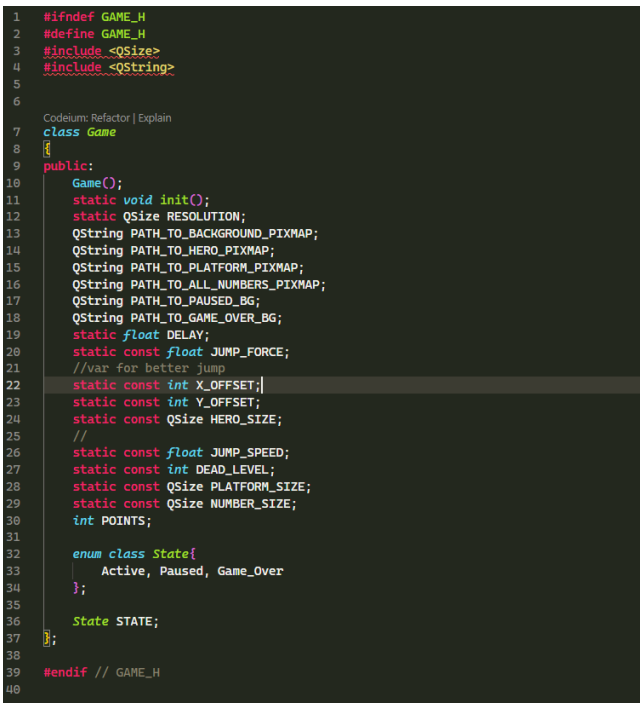
Explication des principaux fichiers

game.h et game.cpp

game.h :

- Le fichier d'en-tête **game.h** définit la classe Game ainsi que ses membres. Voici un aperçu de son contenu :
- **Déclaration de la classe** : La classe Game encapsule les propriétés et les méthodes du jeu.
- **Membres statiques**:
  - RESOLUTION et DELAY : Ces variables statiques sont partagées entre toutes les instances de la classe Game. RESOLUTION représente la taille de la fenêtre du jeu, et DELAY pourrait contrôler l'intervalle de mise à jour du jeu.
  - JUMP_FORCE, X_OFFSET, Y_OFFSET, JUMP_SPEED, DEAD_LEVEL, HERO_SIZE, PLATFORM_SIZE, NUMBER_SIZE : Ces constantes définissent diverses propriétés physiques et visuelles du jeu, telles que la force d'un saut, les dimensions du personnage et des plateformes, ainsi que les limites du jeu.
- **Membres non statiques**:
  - PATH_TO_BACKGROUND_PIXMAP, PATH_TO_HERO_PIXMAP, PATH_TO_PLATFORM_PIXMAP, PATH_TO_ALL_NUMBERS_PIXMAP, PATH_TO_PAUSED_BG, PATH_TO_GAME_OVER_BG : Ces chaînes de caractères contiennent les chemins d'accès aux ressources d'images utilisées dans le jeu.
  - POINTS : Suit le score du joueur.
  - STATE : Une énumération State qui indique l'état actuel du jeu (Actif, En pause, Game_Over).

- **Constructeur et méthodes**:
  - Game() : Constructeur qui initialise les chemins d'accès aux ressources d'images et initialise le score initial.



```
#ifndef GAME_H
#define GAME_H
#include <QSize>
#include <QString>


Codeium: Refactor | Explain
class Game
{
public:
    Game();
    static void init();
    static QSize RESOLUTION;
    QString PATH_TO_BACKGROUND_PIXMAP;
    QString PATH_TO_HERO_PIXMAP;
    QString PATH_TO_PLATFORM_PIXMAP;
    QString PATH_TO_ALL_NUMBERS_PIXMAP;
    QString PATH_TO_PAUSED_BG;
    QString PATH_TO_GAME_OVER_BG;
    static float DELAY;
    static const float JUMP_FORCE;
    //var for better jump
    static const int X_OFFSET;
    static const int Y_OFFSET;
    static const QSize HERO_SIZE;
    //
    static const float JUMP_SPEED;
    static const int DEAD_LEVEL;
    static const QSize PLATFORM_SIZE;
    static const QSize NUMBER_SIZE;
    int POINTS;

    enum class State{
        Active, Paused, Game_Over
    };

    State STATE;
};

#endif // GAME_H

```

game.cpp :

Le fichier source **game.cpp** implémente les méthodes et initialise les membres statiques déclarés dans **game.h**. Voici un résumé :

- **Initialisation des membres statiques** : Initialise les membres statiques comme RESOLUTION, DELAY et les constantes du jeu (JUMP_FORCE, X_OFFSET, etc.).
- **Constructeur**:
  - Game::Game() : Initialise les chemins d'accès aux différentes ressources d'images et initialise le score (POINTS = 0).

- **Implémentation des méthodes**:
  - void Game::init() : Définit les valeurs initiales pour RESOLUTION et DELAY.

```cpp
#include "game.h"

QSize Game::RESOLUTION;
float Game::DELAY;
const float Game::JUMP_FORCE = -10.0f;
const int Game::X_OFFSET = 20;
const int Game::Y_OFFSET = 70;
const float Game::JUMP_SPEED = 0.2f;
const int Game::DEAD_LEVEL = 500;;
const QSize Game::HERO_SIZE = QSize(34, 64);
const QSize Game::PLATFORM_SIZE = QSize(64, 16);
const QSize Game::NUMBER_SIZE = QSize(32, 32);

Codeium: Refactor | Explain | Generate Function Comment | ×
Game::Game()
{
    PATH_TO_BACKGROUND_PIXMAP = ":/images/background.png";
    PATH_TO_HERO_PIXMAP = ":/images/hero.png";
    PATH_TO_PLATFORM_PIXMAP = ":/images/platform.png";
    PATH_TO_ALL_NUMBERS_PIXMAP = ":/images/all_numbers.png";
    PATH_TO_PAUSED_BG = ":/images/bg_pause.png";
    PATH_TO_GAME_OVER_BG = ":/images/bg_gameover.png";
    POINTS = 0;
}

Codeium: Refactor | Explain | Generate Function Comment | ×
void Game::init()
{
    RESOLUTION = QSize(400, 533);
    DELAY = 0.2f;
}
```

Interaction entre les fichiers

- **Initialisation** : Lorsqu'une instance de la classe Game est créée, le constructeur initialise les chemins d'accès aux images et initialise POINTS à 0. La méthode statique init() est utilisée pour définir la résolution et le délai, probablement importants pour l'affichage et le timing du jeu.
- **Logique et propriétés du jeu :** Les constantes et membres statiques définissent des aspects importants du comportement du jeu, comme la hauteur du saut du personnage (JUMP_FORCE), la taille des éléments du jeu (HERO_SIZE, PLATFORM_SIZE) et la vitesse de mise à jour du jeu (DELAY).

Ensemble, ces fichiers posent les bases de la logique du jeu, gèrent son état et gèrent le rendu de ses composants graphiques.

gamescene.h et gamescene.cpp

Le fichier **gamescene.cpp** fournit l'implémentation de la classe GameScene, qui est responsable de la gestion de l'interface graphique du jeu et de l'entrée utilisateur. Voici un aperçu détaillé de ce que chaque partie fait :

Initialisation de classe :

Constructeur:

```
GameScene::GameScene(QObject *parent)
    : QGraphicsScene{parent}, m_iteration_value(1000.0f/60.0f),
      m_leftMove(false), m_rightMove(false), m_heroXpos(100), m_heroYpos(100),
      m_deltaX(3), m_deltaY(0.2f), m_height(200), m_facingRight(true), m_countOfPlatforms(10)
{
    Game::init();
    srand(time(0));
    setSceneRect(0, 0, m_game.RESOLUTION.width(), m_game.RESOLUTION.height());
    setBackgroundBrush(Qt::white);

    m_heroPixmap.load(m_game.PATH_TO_HERO_PIXMAP);
    m_heroItem = new QGraphicsPixmapItem(QPixmap(m_heroPixmap));
    m_heroTransform = m_heroItem->transform();
    addItem(m_heroItem);

    m_numberPixmap.load(m_game.PATH_TO_ALL_NUMBERS_PIXMAP);
    m_platformPixmap.load(m_game.PATH_TO_PLATFORM_PIXMAP);

    m_timer = new QTimer(this);
    connect(m_timer, &QTimer::timeout, this, &GameScene::update);
    m_timer->start(m_iteration_value);

    for (int i = 0; i < m_countOfPlatforms; i++) {
        m_platforms[i].x = rand() % Game::RESOLUTION.width();
        m_platforms[i].y = rand() % Game::RESOLUTION.height();
    }
}
```

- Initialisation : Met en place la scène de jeu, initialise les variables du jeu, charge les images et démarre un minuteur pour mettre à jour le jeu à un intervalle fixe (60 fois par seconde).

Key Press Events:

```
void GameScene::keyPressEvent(QKeyEvent *event)
{
    switch (event->key()) {
    case Qt::Key_Left:
        if (m_game.STATE == Game::State::Active) {
            m_leftMove = true;
        }
        break;
    case Qt::Key_Right:
        if (m_game.STATE == Game::State::Active) {
            m_rightMove = true;
        }
        break;
    case Qt::Key_P:
        if (m_game.STATE == Game::State::Active) {
            m_game.STATE = Game::State::Paused;
        } else if (m_game.STATE == Game::State::Paused) {
            m_game.STATE = Game::State::Active;
        }
        break;
    case Qt::Key_R:
        if (m_game.STATE == Game::State::Game_Over) {
            reset();
        }
        break;
    }
    QGraphicsScene::keyPressEvent(event);
}
```

- Gestion des touches : Détecte les pressions de touches pour déplacer le héros, mettre en pause le jeu, réinitialiser le jeu ou prendre une capture d'écran.

```
void GameScene::keyReleaseEvent(QKeyEvent *event)
{
    switch (event->key()) {
    case Qt::Key_Left:
    case Qt::Key_Right:
        m_rightMove = false;
        m_leftMove = false;
        break;
    }
    QGraphicsScene::keyReleaseEvent(event);
}
```

- Gestion de la libération des touches : Arrête le déplacement du héros lorsque les touches de direction sont relâchées.

Clamp X Position:

```
void GameScene::clampXpos()
{
    if (m_heroTransform.m11() == -1) { // Is Turn Left
        if (m_heroXpos - m_heroItem->boundingRect().width() < 0) {
            m_heroXpos = m_heroItem->boundingRect().width();
        }
    } else { // Is Turn Right
        if (m_heroXpos + m_heroItem->boundingRect().width() > Game::RESOLUTION.width()) {
            m_heroXpos = Game::RESOLUTION.width() - m_heroItem->boundingRect().width();
        }
    }
}
```

- Limite de position en X : Assure que le héros ne sort pas de l'écran horizontalement.

Draw Score:

```
void GameScene::drawScore()
{
    QString scoreText = QString::number(m_game.POINTS);
    int unityPartVal = 0;
    int decimalPartValue = 0;
    int hendredthPartValue = 0;

    if (scoreText.length() == 1) { // 0 - 9
        unityPartVal = scoreText.toInt();
    } else if (scoreText.length() == 2) { // 10 - 99
        unityPartVal = scoreText.last(1).toInt();
        decimalPartValue = scoreText.first(1).toInt();
    } else if (scoreText.length() == 3) { // 100 - 999
        unityPartVal = scoreText.last(1).toInt();
        hendredthPartValue = scoreText.first(1).toInt();
        QString copyVal = scoreText;
        copyVal.chop(1);
        decimalPartValue = copyVal.last(1).toInt();
    }

    QGraphicsPixmapItem* unityPartScoreItem = new QGraphicsPixmapItem(m_numberPixmap.copy(
    unityPartScoreItem->moveBy(Game::RESOLUTION.width() - Game::NUMBER_SIZE.width(), 0);
    addItem(unityPartScoreItem);

    QGraphicsPixmapItem* decimalPartScoreItem = new QGraphicsPixmapItem(m_numberPixmap.copy
    decimalPartScoreItem->moveBy(Game::RESOLUTION.width() - 2 * Game::NUMBER_SIZE.width(),
    addItem(decimalPartScoreItem);

    QGraphicsPixmapItem* hundrethPartScoreItem = new QGraphicsPixmapItem(m_numberPixmap.co
    hundrethPartScoreItem->moveBy(Game::RESOLUTION.width() - 3 * Game::NUMBER_SIZE.width()
    addItem(hundrethPartScoreItem);
}
```

- Affichage du score : Affiche le score à l'écran en extrayant les chiffres et en affichant les images correspondantes des chiffres

Reset Game:

```
void GameScene::reset()
{
    m_heroXpos = 100;
    m_heroYpos = 100;
    m_height = 200;
    for (int i = 0; i < m_countOfPlatforms; i++) {
        m_platforms[i].x = rand() % Game::RESOLUTION.width();
        m_platforms[i].y = rand() % Game::RESOLUTION.height();
    }

    m_game.STATE = Game::State::Active;
    m_game.POINTS = 0;
}
```

- Réinitialisation du jeu : Réinitialise la position du héros, les plateformes, l'état du jeu et le score pour commencer une nouvelle partie.

view.h et view.cpp

view.h :

```h
#ifndef VIEW_H
#define VIEW_H

#include <QGraphicsView>
class GameScene;

Codeium: Refactor | Explain
class View : public QGraphicsView
{
    Q_OBJECT
public:
    explicit View();

signals:

private:
    GameScene* m_gameScene;

    // QWidget interface
protected:
    virtual void keyPressEvent(QKeyEvent *event) override;
};

#endif // VIEW_H

```

- **Gardes d'inclusion :**

#ifndef VIEW_H et #define VIEW_H au début et #endif à la fin empêchent les inclusions multiples de ce fichier d'en-tête.

- **Inclusions :**

#include <QGraphicsView> : Inclut la classe Qt nécessaire pour afficher une scène graphique.

- **Déclaration anticipée :**

class GameScene; : Déclaration anticipée de GameScene pour éviter les problèmes de dépendances circulaires.

- **Classe View :**

class View : public QGraphicsView : Hérite de QGraphicsView, qui est utilisé pour afficher la scène du jeu.

- **Constructeur public :**

explicit View(); : Constructeur pour la classe View.

- **Section Signaux :**

Vide pour le moment, mais réservé pour les signaux Qt qui peuvent être ajoutés plus tard si nécessaire.

- **Membres privés :**

GameScene* m_gameScene; : Pointeur vers une instance de GameScene.

- **Méthodes protégées :**

virtual void keyPressEvent(QKeyEvent *event) override; : Réimplémentation de keyPressEvent pour gérer des événements clavier personnalisés.

view.cpp :

```cpp
#include "view.h"
#include "gamescene.h"
#include <QApplication>
#include <QKeyEvent>

Codeium: Refactor | Explain | Generate Function Comment | X
View::View()
    : QGraphicsView(), m_gameScene(new GameScene(this))
{
    setScene(m_gameScene);
    resize(Game::RESOLUTION.width()+2, Game::RESOLUTION.height()+2);
    setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
    setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);

}

Codeium: Refactor | Explain | Generate Function Comment | X
void View::keyPressEvent(QKeyEvent *event)
{
    if(!event->isAutoRepeat())
    {
        switch (event->key()) {
            case Qt::Key_Escape:
                QApplication::instance()->quit();
            break;
        }
    }
    QGraphicsView::keyPressEvent(event);
}
```

- **Inclusions :**Inclut les fichiers d'en-tête pour view, gamescene, QApplication et QKeyEvent.
- **Constructeur View::View :**
- **QGraphicsView() :** Appelle le constructeur de la classe de base.
- **m_gameScene(new GameScene(this)) :** Initialise m_gameScene avec une nouvelle instance de GameScene, en passant this comme parent.
- **setScene(m_gameScene) :** Définit la scène à afficher dans cette vue.
- **resize(Game::RESOLUTION.width() + 2, Game::RESOLUTION.height() + 2) :** Redimensionne la vue pour correspondre à la résolution du jeu. setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff)

- **setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff) :** Désactive les barres de défilement pour une interface plus propre.
- **Gestionnaire d'événements de pression de touche View::keyPressEvent:** Vérifie si l'événement n'est pas une répétition automatique. Si la touche Échap est enfoncée, il quitte l'application en utilisant QApplication::instance()->quit(). Appelle l'implémentation de la classe de base pour gérer tout autre événement de touche.

Fonctionnalités du jeu

Principales fonctionnalités :

- Déplacement du personnage principal
- Saut et collision avec les plateformes
- Comptage des points
- États du jeu (Actif, Pause, Game Over)

Instructions pour jouer

Commandes du clavier :

- Flèche gauche : Déplacer à gauche
- Flèche droite : Déplacer à droite
- P : Pause/Continuer le jeu
- R : Réinitialiser après Game Over
- Échap : Quitter le jeu

Conclusion

En conclusion, cette exploration de JanJump s'est avérée être une expérience particulièrement enrichissante. Nous avons pu observer de près comment les concepts de la POO sont appliqués dans le domaine des jeux vidéo, sans pour autant revendiquer la création de l'application. En nous familiarisant avec des technologies telles que C++, Qt, Git, et en explorant les principes fondamentaux de la programmation orientée objet, nous avons approfondi notre compréhension des méthodes utilisées pour développer des applications interactives et dynamiques.

Ce projet nous a permis d'apprécier l'importance de l'organisation du code et de la gestion des ressources visuelles comme les images, tout en soulignant l'efficacité de Git pour la collaboration et la gestion de versions. En résumé, JanJump a représenté bien plus qu'une simple exploration technologique : il nous a offert une plongée immersive dans les rouages complexes du développement logiciel appliqué aux jeux, enrichissant notre parcours d'apprentissage et ouvrant de nouvelles perspectives pour nos futurs projets.