**GameFramework**

**Les bases d’un casse-briques**

Une image contenant texte, équipement électronique, ordinateur, afficher

Description générée automatiquement

# Table des matières

[1 Table des matières 2](#_Toc119077960)

[2 Introduction 4](#_Toc119077961)

[2.1 But de ce tutorial 4](#_Toc119077962)

[3 Mise en place 4](#_Toc119077963)

[3.1 Créer un nouveau jeu 4](#_Toc119077964)

[3.2 Commencer le développement 4](#_Toc119077965)

[4 Ajouter un sprite à la scène 5](#_Toc119077966)

[4.1 Systèmes de coordonnées 8](#_Toc119077967)

[5 Déplacer un sprite 8](#_Toc119077968)

[5.1 Déplacement par la cadence 9](#_Toc119077969)

[5.2 Déplacement sur deux axes 10](#_Toc119077970)

[5.3 Améliorer le rebond 11](#_Toc119077971)

[5.4 Déplacer un sprite au clavier 12](#_Toc119077972)

[5.4.1 Déplacement simple 13](#_Toc119077973)

[5.4.2 Déplacement en cadence 14](#_Toc119077974)

[6 Détecter les collisions 16](#_Toc119077975)

[6.1 Vérifier après avoir déplacé 17](#_Toc119077976)

[6.2 Les rectangles d’encombrement 19](#_Toc119077977)

[6.3 Déboguer les collisions 20](#_Toc119077978)

[7 Supprimer un sprite 21](#_Toc119077979)

[7.1 Distinguer différents types de sprite 22](#_Toc119077980)

[7.1.1 Identifier un sprite par une donnée distinctive 22](#_Toc119077981)

[8 Fonctionnalités de jeu 24](#_Toc119077982)

[8.1 Les modes de jeu 24](#_Toc119077983)

[8.2 Écrans de début et de fin 26](#_Toc119077984)

[8.2.1 Affichage d’un message 27](#_Toc119077985)

[8.2.2 Recommencer une partie 29](#_Toc119077986)

[9 Agrémentations 32](#_Toc119077987)

[9.1 Programmer la destruction d’un sprite 32](#_Toc119077988)

[9.2 Changer le point de positionnement 33](#_Toc119077989)

[9.2.1 Décalage calculé 34](#_Toc119077990)

[9.2.2 Décalage du point de positionnement 34](#_Toc119077991)

[9.3 Cheat mode 36](#_Toc119077992)

[9.4 Fond d’écran 37](#_Toc119077993)

[10 Suite du travail 38](#_Toc119077994)

[10.1 Problèmes à solutionner 38](#_Toc119077995)

[10.2 Améliorations possibles 39](#_Toc119077996)

[11 Autres concepts 39](#_Toc119077997)

[11.1 Identifier un sprite par son type exact 39](#_Toc119077998)

[11.2 Orienté-objet ? Déplacement par un *tick handler* 43](#_Toc119077999)

[11.2.1 Le *tick handler* 43](#_Toc119078000)

[11.2.2 Balle posée sur la raquette 44](#_Toc119078001)

[11.2.3 Rebond 45](#_Toc119078002)

[11.2.4 Affectation d’un gestionnaire à un sprite 46](#_Toc119078003)

[11.2.5 Autres gestionnaires 52](#_Toc119078004)

[11.3 Vérifier avant de déplacer 52](#_Toc119078005)

[11.3.1 Vérifier le rebond avant de déplacer la balle 53](#_Toc119078006)

[11.4 Gestionnaire de tick et clavier 54](#_Toc119078007)

[12 Concepts avancés 57](#_Toc119078008)

[12.1 Adapter l’animation du déplacement 58](#_Toc119078009)

[12.1.1 Appliquer une rotation 58](#_Toc119078010)

[12.1.2 Calculer l’angle de rotation exact 59](#_Toc119078011)

[12.1.3 Appliquer une matrice de transformation 61](#_Toc119078012)

[12.1.4 Utiliser un jeu d’images différent 63](#_Toc119078013)

[12.2 Interrompre l’animation en cas d’arrêt 63](#_Toc119078014)

[12.3 Sprite sheet 64](#_Toc119078015)

[12.3.1 Chargement et découpage 65](#_Toc119078016)

[12.4 Scrolling 66](#_Toc119078017)

[12.5 Changement de sens animé 68](#_Toc119078018)

[12.6 Listes d’animations 68](#_Toc119078019)

[12.7 Gravité 69](#_Toc119078020)

[12.8 « Manger » du bitmap 69](#_Toc119078021)

[12.9 Dessiner dans un QWidget 69](#_Toc119078022)

[12.10 Quitter l’application 72](#_Toc119078023)

[12.10.1 Fermeture de la fenêtre par l’utilisateur 72](#_Toc119078024)

[12.10.2 Terminer l’application par le programme 72](#_Toc119078025)

[13 Annexes 72](#_Toc119078026)

[13.1 Diagramme de classes 72](#_Toc119078027)

[13.2 Diagrammes de séquence 72](#_Toc119078028)

[13.2.1 Démarrage du jeu 72](#_Toc119078029)

[13.2.2 Quitter l’application 73](#_Toc119078030)

[13.3 Propagation des événements du clavier 74](#_Toc119078031)

[13.4 Ordre de destruction lorsque le programme quitte 74](#_Toc119078032)

# Introduction

Le projet GameFramework est un framework développé en C++, basé sur Qt, qui fournit une base minimale pour le développement d’un jeu vidéo simple, basé sur les sprites.

Le projet peut être cloné à partir du dépît GitHub suivant :

<https://github.com/divtec-cejef/2018-JCO-GameFramework.git>

Il est livré avec différents exemples qui démontrent les possibilités d’animation et de contrôle de sprites pour la création d’un jeu 2D.

Le cœur du jeu doit être codé dans la classe GameCore (fichiers gamecore.h et gamecore.cpp).

La classe GameCore fournie contient le code des exemples.

## But de ce tutorial

Le but de ce tutorial est de découvrir les fonctionnalités de base du *GameFramework* en développant une version simple d’un casse-brique.

# Mise en place

## Créer un nouveau jeu

Pour créer un nouveau jeu à partir du projet GameFramework, il faut lancer le script generate\_new\_project.bat, dans le dossier scripts.

Une boîte de dialogue (rudimentaire) s’ouvre, permettant de spécifier le nom du projet et l’emplacement où les fichiers de base seront créés.

Une image contenant texte

Description générée automatiquement

Un nouveau projet avec l’arborescence suivante est créé :

Une image contenant table

Description générée automatiquement

Le dossier doc contient les fichiers sources de documentation, ainsi que la documentation générée par doxygen.

Le dossier res contient les ressources. Entre autres, un sous-dossier images contenant les images qui seront utilisées dans notre jeu.

Le dossier scripts contient les scripts qui permettront la génération et le déploiement automatique de notre jeu.

Le dossier src contient les fichiers de code source et de configuration du projet.

## Commencer le développement

Pour démarrer le développement de notre jeu, il faut ouvrir le projet (fichier pro situé dans le dossier src) au moyen de Qt Creator, valider la configuration et compiler le code existant. Si le projet a été correctement créé, il devrait compiler sans erreur et démarrer.

Seul un écran noir, vide s’affichage.

# Ajouter un sprite à la scène

Un sprite est un élément graphique (généralement en 2D) qui peut être déplacé sur la surface de jeu, appelée la **scène**.

Pour qu’un objet puisse être affiché par la scène, il doit être de type QGraphicsItem, ou évidemment d’une spécialisation de ce type.

La classe Sprite est une classe spécifique au *GameFramework*, qui hérite de la classe QObject et QGraphicsPixmapItem, qui sont des classes fournies par le framework *Qt*.

Elle implémente un certain nombre de fonctionnalités utiles pour un jeu en 2D.

Pour ajouter un sprite (Sprite) à la scène (GameScene), il faut tout d'abord l'instancier (par exemple dans le constructeur de GameCore) et l'ajouter à la scène.

La variable membre GameCore::m\_pScene pointe sur la scène active.

Lors de l'instanciation d'un sprite, on peut profiter de passer une chaîne de caractères contenant la localisation d'une image sur le disque, afin de spécifier l'apparence du sprite.

Pour pouvoir instancier un sprite, il ne faut pas oublier d'inclure l'entête sprite.h :

**gamecore.cpp** :

#include "sprite.h"

GameCore::**GameCore**(GameCanvas\* pGameCanvas, QObject\* pParent) : QObject(pParent) {

// ...

// Instancier et initialiser les sprites ici :

// ...

Sprite\* pBall = new Sprite(GameFramework::imagesPath() + "brickbreaker/orb\_1.png");

m\_pScene->addSpriteToScene(pBall);

// ...

m\_pGameCanvas->startTick();

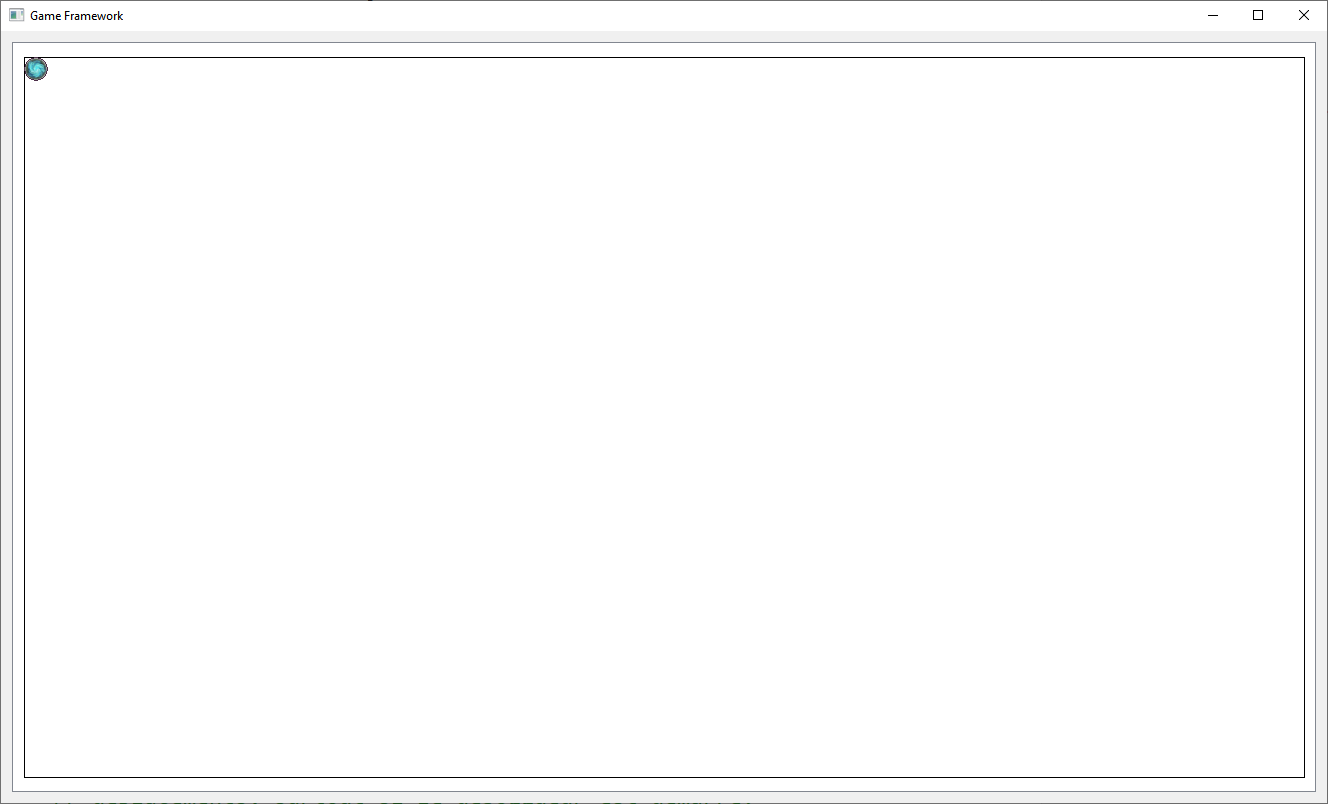
}

**Rappel** : contrairement à Java, le C++ ne **possède pas** (en standard) de **ramasse-miettes** ! (*garbage collector*)

C’est donc le **développeur** qui est **responsable** de **détruire** les objets et variables créés avec le mot-clé new, **sauf** si cette responsabilité est confiée à un autre objet.

C’est le cas ici : Au moment où notre sprite est ajouté à la scène (avec addSpriteToScene), c’est cette dernière qui se chargera de **détruire** le sprite lorsqu’elle sera elle-même **détruite**.

**Attention** : si un sprite est **retiré** de la scène, celle-ci s’affranchit alors de cette responsabilité et le développeur doit donc s’assurer que le sprite sera correctement détruit lorsqu’il ne sera plus utilisé, au moyen de l’instruction delete.



En C++, il est de la responsabilité du développeur qui instancie des objets, de s'assurer qu'ils soient détruits lorsqu'ils ne sont plus utilisés, afin de libérer la mémoire qui était occupée.

Toutefois, lorsqu'un sprite est ajouté une scène, celle-ci en devient la propriétaire et se chargera de détruire le sprite lorsque la scène sera détruite.

Par défaut, le sprite est positionné à la coordonnée (0 ; 0). Cette coordonnée représente le coin supérieur gauche de l’image. Au besoin, cela peut être modifié de même qu’il est possible de mettre l’image à l’échelle si elle est trop grande (ou trop petite).

Pour le positionner ailleurs, utiliser la méthode Sprite::setPos() :

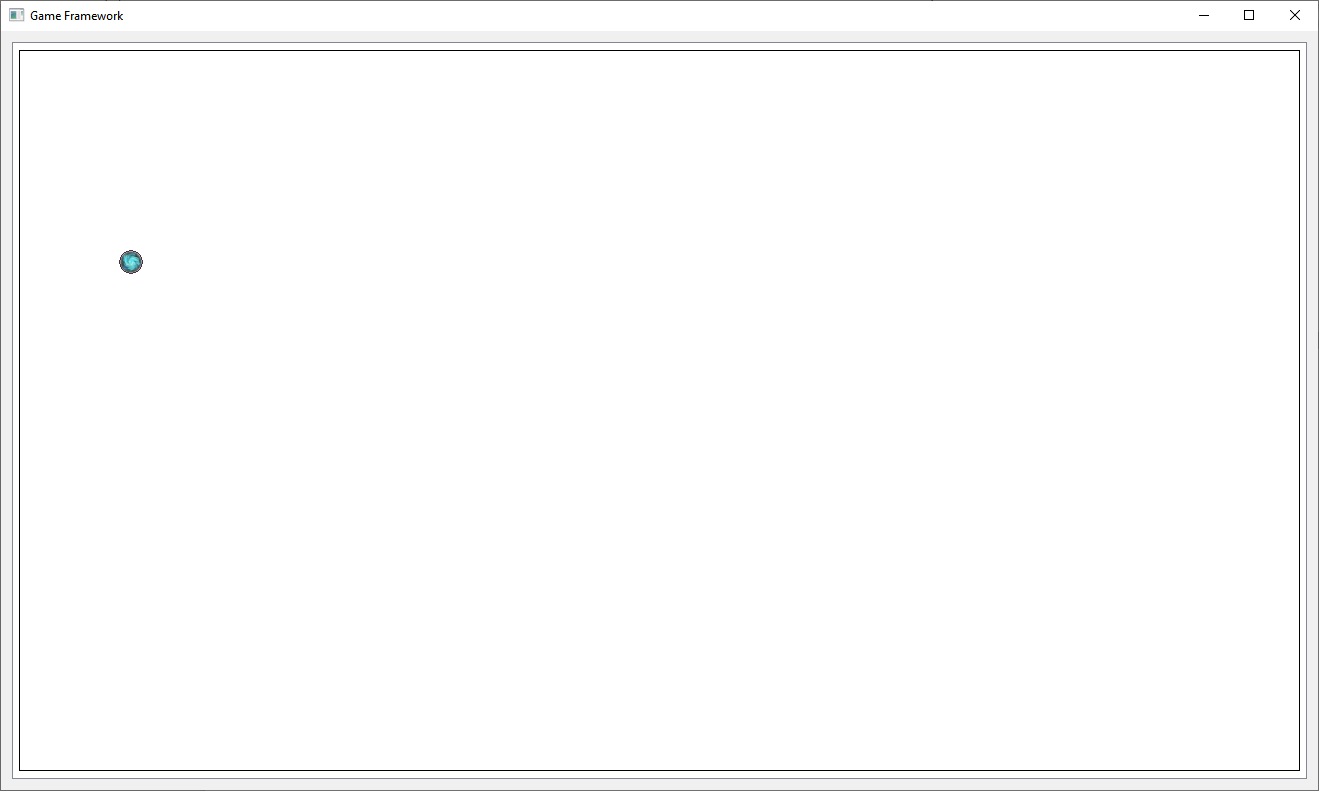
// Instancier et initialiser les sprites ici :

// ...

Sprite\* pBall = new Sprite(GameFramework::imagesPath() + "brickbreaker/orb\_1.png");

m\_pScene->addSpriteToScene(pBall);

pBall->setPos(100, 200);



Pour le positionner au milieu de la scène, il faut tenir compte des dimensions (largeur et hauteur) de la scène (avec les méthodes GameScene::width() et GameScene::height()) :

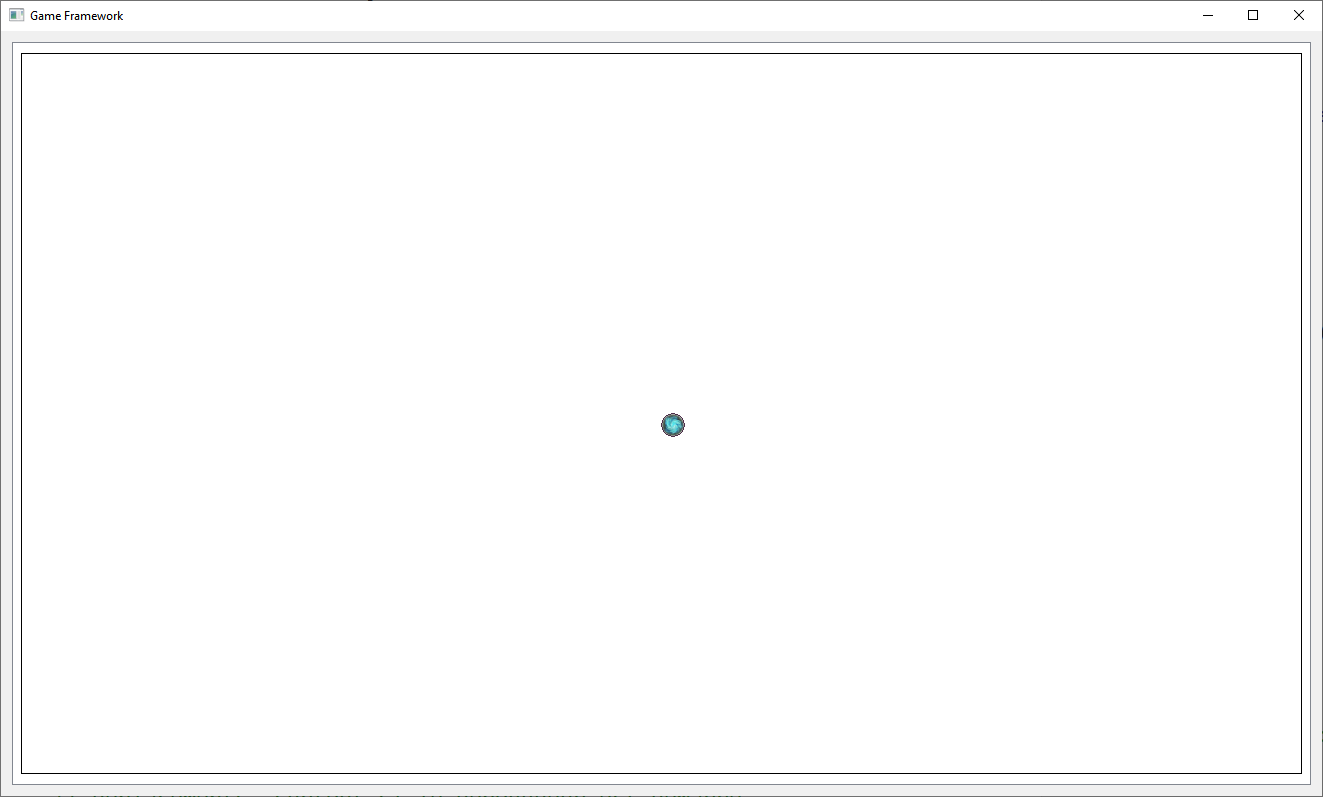
// Instancier et initialiser les sprites ici :

// ...

Sprite\* pBall = new Sprite(GameFramework::imagesPath() + " brickbreaker/orb\_1.png ");

m\_pScene->addSpriteToScene(pBall);

pBall->setPos(m\_pScene->width()/2.0, m\_pScene->height()/2.0);



Il est possible d’animer l’apparence du sprite en lui affectant plusieurs images grâce à la méthode Sprite::addAnimationFrame().

L’image affichée par le sprite peut ensuite être sélectionnée manuellement par son index (Sprite::setCurrentAnimationFrame()), ou être changée automatiquement, à une vitesse donnée, grâce à Sprite::startAnimation() :

// Instancier et initialiser les sprites ici :

Sprite\* pBall = new Sprite(GameFramework::imagesPath() + "brickbreaker/orb\_1.png");

m\_pScene->addSpriteToScene(*pBall*);

pBall->setPos(m\_pScene->width()/2.0, m\_pScene->height()/2.0);

pBall->addAnimationFrame(GameFramework::imagesPath() + "brickbreaker/orb\_2.png");

pBall->addAnimationFrame(GameFramework::imagesPath() + "brickbreaker/orb\_3.png");

pBall->addAnimationFrame(GameFramework::imagesPath() + "brickbreaker/orb\_4.png");

pBall->startAnimation(100);

Dans l’exemple ci-dessus, le sprite affichera tour à tour les quatre images (orb 1, orb 2, orb3 et orb4), toutes les 100 millisecondes.

## Systèmes de coordonnées



La position d’un sprite est donnée par rapport au système de coordonnées **global** de la scène.

L’origine de la scène (le point 0,0) correspond à son coin supérieur gauche. L’axe des X positifs va en direction de la **droite** de l’écran, l’axe des Y positifs va en direction du **bas** de l’écran.

Par défaut, la position donnée à un sprite indique la position de son coin **supérieur gauche**. C’est son point de positionnement (parfois appelé également *hotspot*).[[1]](#footnote-1)



Certaines manipulations du sprite s’effectuent dans le système de coordonnées **locale** du sprite.

Dans ce cas, l’origine est située sur son point de positionnement.

# Déplacer un sprite

Dans la section précédente, le sprite créé était manipulé par la variable locale pSprite, qui est en réalité un pointeur sur l’objet sprite créé. Cette variable est **détruite** à la fin de la méthode.

Pour pouvoir manipuler le sprite ultérieurement (donc dans une autre méthode de GameCore), il est nécessaire de **mémoriser** ce pointeur dans une **variable membre** (ou champ) privée. Il faut donc déclarer cette variable dans le fichier d’entête gamecore.h :

…

class **GameCore** : public QObject

{

Q\_OBJECT

public:

…

signals:

…

private:

GameCanvas\* **m\_pGameCanvas** = nullptr;

GameScene\* **m\_pScene** = nullptr;

Sprite\* **m\_pBall** = nullptr;

private slots:

};

La classe Sprite n'est **pas connue** dans gamecore.h. Comme cette classe, dans le fichier d’entête, n’est utilisée que pour mémoriser un **pointeur**, il n’est **pas nécessaire** d’inclure toute la classe (avec #include "sprite.h"). Il suffit de **déclarer** la classe afin que le compilateur sache qu’elle **existe**, à la suite des autres inclusions :

#include <QObject>

#include <QPointF>

class **GameCanvas**;

class **GameScene**;

class **Sprite**;

Il ne reste plus qu’à initialiser cette variable membre dans le constructeur de GameCore :

// Instancier et initialiser les sprites ici :

Sprite\* **pBall** = new Sprite(GameFramework::imagesPath() + "brickbreaker/orb\_1.png");

m\_pScene->addSpriteToScene(*pBall*);

pBall->setPos(m\_pScene->width()/2.0, m\_pScene->height()/2.0);

pBall->addAnimationFrame(GameFramework::imagesPath() + "brickbreaker/orb\_2.png");

pBall->addAnimationFrame(GameFramework::imagesPath() + "brickbreaker/orb\_3.png");

pBall->addAnimationFrame(GameFramework::imagesPath() + "brickbreaker/orb\_4.png");

pBall->startAnimation(100);

m\_pBall = pBall;

## Déplacement par la cadence

La dernière instruction du constructeur de GameCore démarre la cadence du jeu (tick).

m\_pGameCanvas->startTick();

Lorsque la cadence du jeu est démarrée, la méthode GameCore::tick() est **automatiquement appelée**, par défaut toutes les 20 millisecondes environ (ce qui correspond à environ 50 images/seconde).

//! Cadence.

//! Gère le déplacement de la Terre qui tourne en cercle.

//! \param elapsedTimeInMilliseconds Temps écoulé depuis le dernier appel.

void GameCore::**tick**(long long elapsedTimeInMilliseconds) {

}

Le paramètre elapsedTimeInMilliseconds indique combien de millisecondes se sont écoulées depuis le dernier tick (donc depuis le dernier appel de la méthode tick()). En théorie, il devrait toujours être exactement de 20 millisecondes, mais en réalité, cette valeur peut légèrement varier selon la charge du système d’exploitation.

Pour obtenir un mouvement le plus fluide possible, il faut tenir compte de cette légère variation. Le principe est de déterminer une vitesse de déplacement du sprite (ici, notre balle) en pixels par seconde. Cette vitesse est notée . Il s’agit ensuite de calculer par la formule ci-dessous la distance parcourue (, en pixels) pendant la durée du tick (, en secondes) :

const int BALL\_SPEED = 300 ; // vitesse de déplacement de la balle en pixels/s

…

void GameCore::**tick**(long long elapsedTimeInMilliseconds) {

double distance = PLAYER\_SPEED \* elapsedTimeInMilliseconds / 1000.0;

m\_pPlayer->setX(m\_pPlayer->x() + distance);

}

L’exemple simple ci-dessus se contente de déplacer le sprite sur la droite, à une vitesse constante.

On constate que désormais, tant que la cadence fonctionne, notre balle se déplace à droite et fini par sortir de la zone d’affichage.

Pour faire un aller-retour, il faut, par exemple, mémoriser dans une **variable membre supplémentaire**, la **direction** de déplacement :

**gamecore.h :**

private:

GameCanvas\* **m\_pGameCanvas** = nullptr;

GameScene\* **m\_pScene** = nullptr;

Sprite\* **m\_pBall** = nullptr;

int **m\_ballDirection** = 1;

**gamecore.cpp :**

void GameCore::**tick**(long long elapsedTimeInMilliseconds) {

// Détermine la distance parcourue en tenant compte de la direction

double **distance** = BALL\_SPEED \*

elapsedTimeInMilliseconds / 1000.0 \*

m\_ballDirection;

m\_pBall->setX(m\_pBall->x() + distance);

// Détecte si le joueur atteint un des bords de la scène

if (m\_pBall->right() > m\_pScene->width() ||

m\_pBall->left() < 0) {

m\_ballDirection \*= -1; // Change de direction

}

}

La méthode Sprite::right() indique la coordonnée en X du bord droit du sprite.

La méthode Sprite::left() indique la coordonnée en X du bord gauche du sprite.

## Déplacement sur deux axes

L’exemple précédent ne déplace le sprite que sur l’axe des X, en utilisant la méthode Sprite::setX().

Bien sûr il est également possible de déplacer le sprite sir l’axe des Y, en utilisant la méthode Sprite::setY().

Lorsqu’un sprite est amené à se déplacer sur les deux axes (horizontal et vertical), il devient plus intéressant de gérer la direction de son déplacement au moyen d’un **vecteur de déplacement**.

Ce vecteur permet de représenter la direction d’un déplacement par une composante le long de l’axe X et une autre composante le long de l’axe Y. Pour cela, la classe QPointF fait parfaitement l’affaire, car elle mémorise une valeur X et une valeur Y.

QPointF(1,0) : représente un déplacement à droite le long de l’axe X.

QPointF(-1,0) : représente un déplacement à gauche le long de l’axe X.

QPointF(0,-1) : représente un déplacement vers le haut le long de l’axe Y.

QPointF(0,1) : représente un déplacement vers le bas de long de l’axe Y.

Il est bien sûr possible de combiner différentes valeurs pour X et Y afin de produire des déplacements qui ne sont pas orthonormés.

QPointF(1,1) : représente un déplacement de 45° vers le bas à droite.

Pour implémenter cela, la variable membre m\_ballDirection doit changer de type afin d’être un QPointF.

QPointF m\_ballDirection = QPointF(1,1);

Il faut ensuite adapter le code de la méthode tick(), pour calculer le mouvement du sprite à partir du **vecteur** de déplacement et l’appliquer au sprite au moyen de la méthode Sprite::setPos().

Le bloc de code qui détecte si le sprite sort de la scène doit également être adapté pour tenir compte des limites supérieures et inférieures.

//! Cadence.

//! \param elapsedTimeInMilliseconds Temps écoulé depuis le dernier appel.

void GameCore::**tick**(long long **elapsedTimeInMilliseconds**) {

// Création d'un vecteur de déplacement du sprite.

QPointF **ballMovement** = m\_ballDirection \* BALL\_SPEED \* elapsedTimeInMilliseconds / 1000.0;

m\_pBall->setPos(m\_pBall->pos() + ballMovement);

// Détecte si la balle atteint un des bords de la scène

if (m\_pBall->right() > m\_pScene->width() ||

m\_pBall->left() < 0 ||

m\_pBall->top() < 0 ||

m\_pBall->bottom() > m\_pScene->height()) {

m\_ballDirection \*= -1; // Change de direction

}

}

Le code qui vient d’être implémenté déplace la balle en bas à droite. Lorsque le bord de la scène est atteint, la direction est inversée et la balle remonte en haut à gauche.

Elle ne rebondit donc pas vraiment, car les **deux composantes** du vecteur de déplacement, la composante X et la composante Y, sont **inversées**.

## Améliorer le rebond

En réalité, pour reproduire un effet de rebond, lorsque la balle touche un obstacle, seule une des deux composantes doit être inversée[[2]](#footnote-2).

Si la balle touche un obstacle horizontal (le long de l’axe X), c’est la composante Y du vecteur de déplacement qui doit être inversée.



Si la balle touche un obstacle vertical (le long de l’axe Y), c’est la composante X du vecteur de déplacement qui doit être inversée.



La méthode tick peut être modifiée ainsi pour améliorer l’effet de rebond :

//! Cadence.

//! \param elapsedTimeInMilliseconds Temps écoulé depuis le dernier appel.

void GameCore::**tick**(long long **elapsedTimeInMilliseconds**) {

// Création d'un vecteur de déplacement du sprite.

QPointF **ballMovement** = m\_ballDirection \* BALL\_SPEED \* elapsedTimeInMilliseconds / 1000.0;

m\_pBall->setPos(m\_pBall->pos() + ballMovement);

// Détecte si la balle atteint un des bords de la scène

if (m\_pBall->right() > m\_pScene->width() || m\_pBall->left() < 0) {

m\_ballDirection.setX(m\_ballDirection.x() \* -1);

} else if (m\_pBall->top() < 0 || m\_pBall->bottom() > m\_pScene->height()) {

m\_ballDirection.setY(m\_ballDirection.y() \* -1);

}

}

## Déplacer un sprite au clavier

La balle se déplace maintenant toute seule.

Ajoutons la raquette, qui sera déplacée au clavier. Cela se fait dans le constructeur de GameCore :

// Instancier et initialiser les sprites ici :

Sprite\* **pBall** = new Sprite(GameFramework::imagesPath() + "brickbreaker/orb\_1.png");

m\_pScene->addSpriteToScene(*pBall*);

pBall->setPos(m\_pScene->width()/2.0, m\_pScene->height()/2.0);

pBall->addAnimationFrame(GameFramework::imagesPath() + "brickbreaker/orb\_2.png");

pBall->addAnimationFrame(GameFramework::imagesPath() + "brickbreaker/orb\_3.png");

pBall->addAnimationFrame(GameFramework::imagesPath() + "brickbreaker/orb\_4.png");

pBall->startAnimation(100);

m\_pBall = pBall;

Sprite\* **pPaddle** = new Sprite(GameFramework::imagesPath() + "brickbreaker/paddle1.png");

m\_pScene->addSpriteToScene(*pPaddle*);

pPaddle->setPos(m\_pScene->width()/2.0, m\_pScene->height()-100);

pPaddle->addAnimationFrame(GameFramework::imagesPath() + "brickbreaker/paddle2.png");

pPaddle->addAnimationFrame(GameFramework::imagesPath() + "brickbreaker/paddle3.png");

pPaddle->startAnimation(100);

m\_pPaddle = pPaddle;

Il ne faut pas oublier d’ajouter la variable membre privée m\_pPaddle dans le fichier d’entête de GameCore :

class **GameCore** : public QObject

{

…

private:

GameCanvas\* **m\_pGameCanvas** = nullptr;

GameScene\* **m\_pScene** = nullptr;

Sprite\* **m\_pBall** = nullptr;

Sprite\* **m\_pPaddle** = nullptr;

QPointF **m\_ballDirection** = QPointF(1,1);

private slots:

};

Une image contenant texte

Description générée automatiquement

### Déplacement simple

Dans sa forme la plus simple, il s’agit, lorsqu’une touche est appuyée, de changer la position du sprite à déplacer, en l’occurrence la raquette.

Lorsqu'une touche est appuyée, la méthode GameCore::keyPressed() est automatiquement appelée.

Lorsqu'une touche est relâchée, la méthode GameCore::keyReleased() est automatiquement appelée.

Il est important de ne **pas remplacer** le **code existant**, mais d’y **ajouter** notre code de déplacement.

//! Traite la pression d'une touche.

//! \param key Numéro de la touche (voir les constantes Qt)

//!

void GameCore::**keyPressed**(int key) {

emit notifyKeyPressed(key);

switch(key) {

case Qt::Key\_Left:

m\_pPaddle->setX(m\_pPaddle->x() - 20); break;

case Qt::Key\_Right:

m\_pPaddle->setX(m\_pPaddle->x() + 20); break;

}

}

On constate que la **répétition** des touches est **désactivée** par défaut. Pour **enclencher** la répétition des touches, il faut modifier la méthode GameCanvas::keyPressed() et éventuellement GameCanvas::keyReleased().

//! Gère l'appui sur une touche du clavier.

//! Les répétitions automatiques sont ignorées.

void GameCanvas::**keyPressed**(QKeyEvent\* pKeyEvent) {

// Supprimer ce premier test si la répétition de touche doit être signalée.

/\* if (pKeyEvent->isAutoRepeat())

pKeyEvent->ignore();

else\*/ {

m\_pGameCore->keyPressed(pKeyEvent->key());

…

Cette façon simple de déplacer un sprite n’est pas optimale puisqu’elle provoque une animation **saccadée**, avec une vitesse qui dépend uniquement de la **fréquence de répétition** des touches et pas du tout de la cadence du jeu, ou d’une vitesse de déplacement de la raquette.

Une meilleure solution consiste, comme nous l’avons fait pour le déplacement de la balle, à déterminer une vitesse de déplacement de la raquette (en pixels/s) et d’utiliser le mécanisme de **cadence du jeu** (tick).

Par ailleurs, dans la situation actuelle, aucun contrôle n’est fait pour éviter que la raquette sorte des limites de la scène.

### Déplacement en cadence

Comme nous l’avons vu plus haut (§5.1), lorsque la cadence du jeu est démarrée, la méthode GameCore::tick() est **automatiquement appelée**, par défaut toutes les 20 millisecondes environ.

On peut calculer la distance de déplacement parcourue par la raquette en fonction de sa vitesse et du temps écoulé.

Pour la raquette, il n’y a pas besoin d’un vecteur de déplacement, puisqu’elle ne se déplace que sur l’axe X. Une simple direction suffit, en l’occurrence un nombre entier valant 1 dans le cas d’un déplacement à droite, -1 pour un déplacement à gauche et 0 lorsqu’il n’y a pas de déplacement.

L’appui sur une touche change la direction selon la touche appuyée.

Le relâchement d’une touche met la direction à zéro et la raquette s’arrête.

Ajouter au fichier **gamecore.h** la variable membre privée m\_paddleDirection.

private:

GameCanvas\* **m\_pGameCanvas** = nullptr;

GameScene\* **m\_pScene** = nullptr;

Sprite\* **m\_pBall** = nullptr;

Sprite\* **m\_pPaddle** = nullptr;

QPointF **m\_ballDirection** = QPointF(1,1);

int **m\_paddleDirection** = 0;

Ajouter au fichier gamecore.cpp une constante pour mémoriser la vitesse de déplacement de la raquette, puis compléter les méthodes keyPressed() et keyReleased() ainsi que la méthode tick().

// Constantes

const int **SCENE\_WIDTH** = 1280;

const int **BALL\_SPEED** = 300 ; // vitesse de déplacement de la balle en pixels/s

const int **PADDLE\_SPEED** = 500; // vitesse de déplacement de la raquette, en pixels/s

…

//! Traite la pression d'une touche.

//! \param key Numéro de la touche (voir les constantes Qt)

//!

void GameCore::**keyPressed**(int **key**) {

emit notifyKeyPressed(key);

switch(key) {

case Qt::Key\_Left: m\_paddleDirection = -1; break;

case Qt::Key\_Right: m\_paddleDirection = 1; break;

}

}

//! Traite le relâchement d'une touche.

//! \param key Numéro de la touche (voir les constantes Qt)

void GameCore::**keyReleased**(int **key**) {

emit notifyKeyReleased(key);

m\_paddleDirection = 0;

}

//! Cadence.

//! \param elapsedTimeInMilliseconds Temps écoulé depuis le dernier appel.

void GameCore::**tick**(long long **elapsedTimeInMilliseconds**) {

// Calcul de la distance de déplacement de la raquette

double **paddleMovement** = m\_paddleDirection \*

PADDLE\_SPEED \*

elapsedTimeInMilliseconds / 1000.0;

m\_pPaddle->setX(m\_pPaddle->x() + paddleMovement);

// Détecte si la raquette dépasse un des bords de la scène et la repositionne le cas échéant.

if (m\_pPaddle->right() > m\_pScene->width()) {

m\_pPaddle->setX(m\_pScene->width() - m\_pPaddle->width());

} else if (m\_pPaddle->left() < 0) {

m\_pPaddle->setX(0);

}

// Création d'un vecteur de déplacement de la balle.

QPointF **ballMovement** = m\_ballDirection \* BALL\_SPEED \* elapsedTimeInMilliseconds / 1000.0;

…

}

Le code ci-dessus vérifie également, après avoir déplacé la raquette, que celle-ci ne **sorte pas** des **limites** de la **scène** et la **repositionne** le cas échéant.

Si la **répétition automatique** des touches avait été rétablie dans GameCanvas::keyPressed() (§5.4.1) il faut la **désactiver** car elle n’est pas nécessaire.

Désormais, il est possible de déplacer la raquette au moyen des touches fléchées. Toutefois, on constate que si plusieurs touches sont pressées en même temps puis relâchées une à une, le comportement de la raquette n’est pas optimal. Cela fera l’objet d’une amélioration ultérieure.

# Détecter les collisions

Lors du déplacement d’un sprite, il s’agit parfois de détecter d’éventuelles collisions.

Pour illustrer cela, ajoutons des briques sur la scène.

Les lignes de code suivantes peuvent-être ajoutées au constructeur de GameCore, à la suite des lignes de code créant la balle.

// Instancier et initialiser les sprites ici :

Sprite\* **pBall** = new Sprite(GameFramework::imagesPath() + "brickbreaker/orb\_1.png");

m\_pScene->addSpriteToScene(*pBall*);

pBall->setPos(m\_pScene->width()/2.0, m\_pScene->height()/2.0);

pBall->addAnimationFrame(GameFramework::imagesPath() + "brickbreaker/orb\_2.png");

pBall->addAnimationFrame(GameFramework::imagesPath() + "brickbreaker/orb\_3.png");

pBall->addAnimationFrame(GameFramework::imagesPath() + "brickbreaker/orb\_4.png");

pBall->startAnimation(100);

m\_pBall = pBall;

// Ajout de briques

QPixmap **brickImage** = QPixmap(GameFramework::imagesPath() + "brickbreaker/brick1.png");

int **brickWidth** = brickImage.width() + 1;

int **brickHeight** = brickImage.height() + 1;

for (int **row** = 0; row < 5; row++) {

for (int **col** = 0; col < 10; col++) {

Sprite\* **pBrick1** = new Sprite(brickImage);

m\_pScene->addSpriteToScene(*pBrick1*, 60 + col \* brickWidth, 90 + row \* brickHeight);

}

}

Comme toutes les briques ont la même apparence, l’image est chargée une fois pour toutes sous forme d’un QPixmap et stockée dans la variable brickImage.

Une image contenant table

Description générée automatiquement

Comme il n’y a aucune détection de collision, la balle passe derrière les briques et la raquette.

## Vérifier après avoir déplacé

La scène permet de fournir la liste des sprites qui sont en collision avec un sprite donné grâce à la méthode collidingSprites() :

Depuis GameCore, la scène est accessible grâce à la variable membre m\_pScene.

Il est donc possible de savoir si le sprite de notre balle est en collision avec un autre sprite :

auto **collisions** = m\_pScene->collidingSprites(m\_pBall);

Le mot-clé auto permet de laisser au C++ décider du type de la variable en fonction de ce qui lui est affectée. Concrètement ici, cela permet de simplifier l’écriture. Sans ce mot-clé auto, il faudrait écrire :

QList<Sprite\*> **collisions** = m\_pScene->collidingSprites(m\_pBall);

Si cette liste est vide, c’est que la balle ne collisionne avec aucun autre sprite.

Il est possible qu’un sprite entre en collision avec plusieurs sprites. C’est rare, mais souvent dans ces cas-là, il est suffisant de ne traiter que la première collision.

En cas de collision, avec une brique ou avec la raquette, il faut que notre balle rebondisse. Le problème, c’est qu’il faut pouvoir déterminer dans quelle direction elle doit rebondir. Juste savoir qu’il y a une collision n’est donc pas suffisant.

Les explications qui suivent se basent sur le cas d’une **collision** avec une **brique**, mais le **raisonnement** est exactement le **même** en cas de collision avec la **raquette**.

En cas de collision avec le **bord gauche** de la brique, que la balle vienne depuis le haut (figure 1) ou depuis le bas (figure 2), c’est la **composante X** qui doit être **inversée**, la composante Y reste inchangée.

|  |  |
| --- | --- |
|  |  |
| Figure 1  La balle collisionne le bord gauche en venant depuis le haut | Figure 2  La balle collisionne le bord gauche en venant depuis le bas |

En cas de collision avec le **bord droit** de la brique, que la balle vienne depuis le haut (figure 3) ou depuis le bas (figure 4), c’est **également** la **composante X** qui doit être **inversée**, la composante Y reste inchangée.

|  |  |
| --- | --- |
|  |  |
| Figure 3  La balle collisionne le bord droit en venant depuis le haut | Figure 4  La balle collisionne le bord droit en venant depuis le bas |

En cas de collision avec le **bord supérieur** de la brique, que la balle vienne depuis la gauche (figure 5) ou depuis la droite (figure 6), c’est la **composante Y** qui doit être **inversée**, la composante X reste inchangée.

|  |  |
| --- | --- |
|  |  |
| Figure 5  La balle collisionne le bord supérieur en venant depuis la gauche | Figure 6  La balle collisionne le bord supérieur en venant depuis la droite |

En cas de collision avec le **bord inférieur** de la brique, que la balle vienne depuis la gauche (figure 7) ou depuis la droite (figure 8), c’est **également** la **composante Y** qui doit être **inversée**, la composante X reste inchangée.

|  |  |
| --- | --- |
|  |  |
| Figure 7  La balle collisionne le bord inférieur en venant depuis la gauche | Figure 8  La balle collisionne le bord inférieur en venant depuis la droite |

En résumé, on constate que si la collision se fait sur un bord vertical, gauche ou droit, c’est la composante X qui doit être inversée.

Si la collision se fait sur un bord horizontal, supérieur ou inférieur, c’est la composante Y qui doit être inversée.

Il s’agit donc de trouver une astuce pour savoir si c’est une **collision** **verticale** ou **horizontale**. Pour cela, nous aurons besoin des **rectangles d’encombrement**.

## Les rectangles d’encombrement



Une idée simple est de se baser sur les rectangles d’encombrement.

Chaque sprite possède un **rectangle d’encombrement**, appelé également *boîte d’encombrement* ou en anglais *bounding rect* ou *bounding box*.

C’est le rectangle qui englobe le sprite. Il peut être obtenu par le méthode Sprite::globalBoundingRect().

En cas de collision entre la balle et une brique, il est possible de déterminer, en fonction des rectangles d’encombrement respectifs, un rectangle de collision :



On constate que le rectangle d’intersection (en violet translucide), en cas de collision sur le bord gauche (ou droit) et plus haut que large.

En cas de collision sur le bord supérieur ou inférieur, ce rectangle d’intersection (en violet translucide) est plus large que haut.

La classe QRectF nous met à disposition la méthode intersected() permettant de déterminer le rectangle obtenu par l’intersection d’un rectangle avec un autre.

Ici, il s’agit d’obtenir le rectangle d’intersection entre le rectangle d’encombrement de la balle (m\_pBall->globalBoundingRect()) et la brique prenant part à la collision, donc la première brique présente dans la liste des collisions (collisions.at(0)->globalBoundingRect()).

QRectF intersect = m\_pBall->globalBoundingRect()

.intersected(collisions.at(0)->globalBoundingRect());

Une fois ce rectangle obtenu, il faut vérifier s’il est plus large que haut (collision horizontale) ou non (collision verticale).

La méthode tick() de la classe GameCore peut être modifiée ainsi :

//! Cadence.

//! \param elapsedTimeInMilliseconds Temps écoulé depuis le dernier appel.

void GameCore::**tick**(long long **elapsedTimeInMilliseconds**) {

…

// Création d'un vecteur de déplacement du sprite.

QPointF **ballMovement** = m\_ballDirection \* BALL\_SPEED \* elapsedTimeInMilliseconds / 1000.0;

m\_pBall->setPos(m\_pBall->pos() + ballMovement);

QList<Sprite\*> **collisions** = m\_pScene->collidingSprites(m\_pBall);

if (!collisions.isEmpty()) {

// Collision avec une brique ou la raquette, on détermine le rectangle d'intersection

QRectF **intersect** = m\_pBall->globalBoundingRect()

.intersected(collisions.at(0)->globalBoundingRect());

if (intersect.width() < intersect.height())

// Rectangle plus haut que large : collision sur un flanc vertical

m\_ballDirection.setX(m\_ballDirection.x() \* -1);

else

// Rectangle plus large que haut : collision sur un flanc horizontal

m\_ballDirection.setY(m\_ballDirection.y() \* -1);

} else {

// Détecte si la balle atteint un des bords de la scène

if (m\_pBall->right() > m\_pScene->width() || m\_pBall->left() < 0) {

m\_ballDirection.setX(m\_ballDirection.x() \* -1);

} else if (m\_pBall->top() < 0 || m\_pBall->bottom() > m\_pScene->height()) {

m\_ballDirection.setY(m\_ballDirection.y() \* -1);

}

}

}

Évidemment, cette solution n’est **pas parfaite**, par exemple dans le cas où la balle collisionne avec le **coin** d’une brique, ce qui peut provoquer des **rebonds incohérents**.

De même, si la **vitesse** de la balle est **élevée**, elle peut potentiellement « s’enfoncer » dans la brique ou la raquette au point de provoquer également des **rebonds incohérents**. Dans certains cas, elle pourrait même **sortir** de la **scène**.

Ces différents cas devront être **solutionnés** plus tard.

## Déboguer les collisions

Lors du développement du jeu, il peut être pratique, lorsqu’il s’agit de gérer les collisions, de pouvoir visualiser le rectangle d’encombrement (*bounding rect*) d’un sprite et/ou sa forme (*shape*).

La méthode Sprite::setDebugModeEnabled() permet d’activer l’affichage en mode debug de façon **individuelle**.



Parfois, il peut être utile d’activer l’affichage en mode debug de **tous les sprites** présents sur la scène. Pour faire cela, le fichier d’entête de la classe Sprite possède des DEFINES qui sont commentés par défaut mais qui peuvent être décommentés.

// décommenter pour rendre le boundingrect de tous les sprites visible.

#define **DEBUG\_BRECT**

// décommenter pour rendre la shape de tous les sprites visible.

#define **DEBUG\_SHAPE**

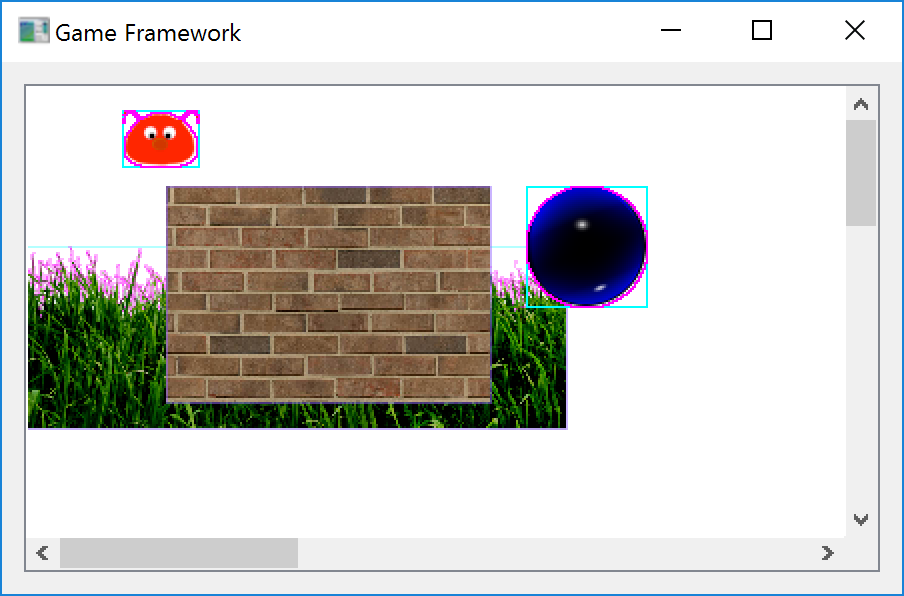
Ceci fait, chaque sprite dessiné est représenté avec son *bounding rect* en cyan et sa *shape* en magenta.

Au besoin, ces couleurs peuvent être modifiée dans la méthode Sprite::paint().

Une troisième DEFINE permet d’afficher dans la console de sortie de l’application le nombre de sprites instanciés.

// décommenter pour afficher dans la sortie de debug le nombre de sprites existants

#define DEBUG\_SPRITE\_COUNT



# Supprimer un sprite

Lorsque la balle entre en collision avec une brique, celle-ci doit être détruite.

**Attention** toutefois, la **suppression** d’un sprite doit être faite avec la plus grande **prudence**.

Idéalement, l’objet qui a **ajouté** le sprite à la **scène** devrait être celui qui le **retire** de la scène avec GameScene::removeSpriteFromScene() puis qui le détruit avec l’instruction delete. Dans notre cas, c’est GameCore.

// Exemple de suppression du sprite pointé par pSpriteToDelete

m\_pScene->removeSpriteFromScene(pSpriteToDelete);

delete pSpriteToDelete;

pSpriteToDelete = nullptr;

Nous verrons plus tard que ce n’est pas toujours possible.

Dans l’immédiat, adaptons notre code pour **supprimer** la **brique** avec laquelle notre balle est entrée en **collision**.

Le **problème** que nous avons est qu’en cas de **collision**, on sait quel **sprite** est en collision, mais on ne sait pas si ce sprite est une **brique** ou la **raquette**. Or, c’est évidemment important de le savoir car la brique doit être détruite, mais pas la raquette.

Il faut donc être capable de déterminer à quel type de sprite nous avons affaire.

## Distinguer différents types de sprite

Pour faire cette distinction, il y a **deux** méthodes. La **première** n’est pas typiquement orientée-objet et consiste à **ajouter** au sprite une **donnée** **distinctive** permettant de l’identifier. L’autre méthode, typiquement orientée-objet, consiste à déterminer le type exact du sprite au sein d’une hiérarchie de classes héritant de la classe Sprite.

Cette deuxième façon de faire sera expliquée ultérieurement.

### Identifier un sprite par une donnée distinctive

Un sprite est capable de mémoriser des données grâce à la méthode setData(), qui prend deux paramètres : une ***clé*** (un nombre entier) et une ***valeur*** qui lui est associée (n’importe quel type de donnée).

Par exemple, l’instruction pBrick->setData(0, "brique") mémorise pour la clé 0 la chaîne de caractères "brique".

Il est ensuite possible de lire cette données grâce à la méthode data() : pBrick->data(0).toString();.

Evidemment, le but est d’éviter les *magic numbers* et il serait mieux d’identifier un sprite avec un nombre plutôt qu’une chaîne de caractères.

Nous pourrions donc déclarer deux types énumérés pour pallier ces problèmes :

enum SpriteDataKey {

SPRITE\_TYPE\_KEY = 0

};

enum SpriteType {

BRICK,

PADDLE

};

Ainsi, le code ci-dessus deviendrait :

pBrick->setData(SPRITE\_TYPE\_KEY, BRICK);

pBrick->data(SPRITE\_TYPE\_KEY).toInt();

Adaptons notre programme afin qu’il implémente cette méthode.

Dans notre cas, ces enumérés ne seront utilisés que dans le fichier d’implémentation de GameCore. Ils peuvent donc être déclarés dans ce fichier **gamecore.cpp**.

// Constantes

const int **SCENE\_WIDTH** = 1280;

const int **BALL\_SPEED** = 300 ; // vitesse de déplacement de la balle en pixels/s

const int **PADDLE\_SPEED** = 500; // vitesse de déplacement de la raquette, en pixels/s

// Enumérés

enum **SpriteDataKey** {

**SPRITE\_TYPE\_KEY** = 0

};

enum **SpriteType** {

**BRICK**,

**PADDLE**,

**BALL**

};

Lors de la création des sprites, dans le constructeur de GameCore, il faut attribuer une donnée à chaque sprite.

GameCore::**GameCore**(GameCanvas\* **pGameCanvas**, QObject\* **pParent**) : QObject(*pParent*) {

…

// Instancier et initialiser les sprites ici :

Sprite\* **pBall** = new Sprite(GameFramework::imagesPath() + "brickbreaker/orb\_1.png");

m\_pScene->addSpriteToScene(*pBall*);

pBall->setPos(m\_pScene->width()/2.0, m\_pScene->height()/2.0);

pBall->addAnimationFrame(GameFramework::imagesPath() + "brickbreaker/orb\_2.png");

pBall->addAnimationFrame(GameFramework::imagesPath() + "brickbreaker/orb\_3.png");

pBall->addAnimationFrame(GameFramework::imagesPath() + "brickbreaker/orb\_4.png");

pBall->startAnimation(100);

pBall->setData(SPRITE\_TYPE\_KEY, BALL);

m\_pBall = pBall;

Sprite\* **pPaddle** = new Sprite(GameFramework::imagesPath() + "brickbreaker/paddle1.png");

m\_pScene->addSpriteToScene(*pPaddle*);

pPaddle->setPos(m\_pScene->width()/2.0, m\_pScene->height()-100);

pPaddle->addAnimationFrame(GameFramework::imagesPath() + "brickbreaker/paddle2.png");

pPaddle->addAnimationFrame(GameFramework::imagesPath() + "brickbreaker/paddle3.png");

pPaddle->startAnimation(100);

pPaddle->setData(SPRITE\_TYPE\_KEY, PADDLE);

m\_pPaddle = pPaddle;

// Ajout de briques

QPixmap **brickImage** = QPixmap(GameFramework::imagesPath() + "brickbreaker/brick1.png");

int **brickWidth** = brickImage.width() + 1;

int **brickHeight** = brickImage.height() + 1;

for (int **row** = 0; row < 5; row++) {

for (int **col** = 0; col < 10; col++) {

Sprite\* **pBrick1** = new Sprite(brickImage);

pBrick1->setData(SPRITE\_TYPE\_KEY, BRICK);

m\_pScene->addSpriteToScene(*pBrick1*, 60 + col \* brickWidth, 90 + row \* brickHeight);

}

}

…

}

On peut maintenant, dans la méthode GameCore::tick(), tenir compte du type de sprite en cas de collision :

…

if (!collisions.isEmpty()) {

// Collision avec une élément, on détermine le rectangle d'intersection

Sprite\* **pCollisionned** = collisions.at(0);

QRectF **intersect** = m\_pBall->globalBoundingRect()

.intersected(pCollisionned->globalBoundingRect());

if (intersect.width() < intersect.height())

// Rectangle plus haut que large : collision sur un flanc vertical

m\_ballDirection.setX(m\_ballDirection.x() \* -1);

else

// Rectangle plus large que haut : collision sur un flanc horizontal

m\_ballDirection.setY(m\_ballDirection.y() \* -1);

// Si collision avec une brique : on la détruit

if (pCollisionned->data(SPRITE\_TYPE\_KEY).toInt() == BRICK) {

m\_pScene->removeSpriteFromScene(*pCollisionned*);

delete pCollisionned;

pCollisionned = nullptr;

}

} else {

…

# Fonctionnalités de jeu

Maintenant que le fonctionnement général du jeu est mis en place, il reste à prendre en charge les événements de jeu les plus importants :

* Identifier la fin du niveau, toutes les briques sont détruites ;
* Identifier la sortie de la balle au-delà de la raquette ;
* Démarrer le jeu sur impulsion du joueur (par exemple lorsqu’il appuie sur la touche espace).

## Les modes de jeu

Le jeu peut donc se trouver concrètement dans les modes suivants :

* En attente de démarrage
* En cours de jeu
* Fin du jeu car toutes les briques sont détruites
* Fin du jeu car la balle est tombée au-delà de la raquette.

L’idéal pour représenter ces différents modes de jeu est donc d’utiliser une énumération et de mémoriser l’état du jeu dans une variable membre de la classe GameCore.

**gamecore.h**

private:

enum **GameMode** {

**WAITING\_FOR\_START**,

**RUNNING**,

**ENDED\_LOSE**,

**ENDED\_WIN**

};

…

GameCanvas\* **m\_pGameCanvas** = nullptr;

GameScene\* **m\_pScene** = nullptr;

Sprite\* **m\_pBall** = nullptr;

Sprite\* **m\_pPaddle** = nullptr;

QPointF **m\_ballDirection** = QPointF(1,1);

int **m\_paddleDirection** = 0;

GameMode **m\_gameMode** = WAITING\_FOR\_START;

…

Il s’agit ensuite principalement de faire les modifications suivantes :

* Prévoir une touche permettant de démarrer le jeu (passer du mode **WAITING\_FOR\_START** au mode **RUNNING**) ;
* Compter le nombre de briques restantes, afin de savoir quand la dernière brique est détruite, afin de détecter une fin de jeu victorieuse (**ENDED\_WIN**).
* Modifier la méthode GameCore::tick() afin d’adapter la gestion des déplacements selon le mode du jeu et vérifier si la balle tombe en dessous de la raquette afin de détecter une fin de jeu perdante (**ENDED\_LOSE**).

Tout d’abord, compléter le fichier d’entête pour ajouter une variable membre mémorisant le nombre de briques restantes.

GameMode **m\_gameMode** = WAITING\_FOR\_START;

int **m\_brickCount** = 0;

Au moment de la création du champ de briques (dans le constructeur de GameCore), mettre à jour ce compteur de briques restantes :

GameCore::**GameCore**(GameCanvas\* **pGameCanvas**, QObject\* **pParent**) : QObject(*pParent*) {

…

// Ajout de briques

QPixmap **brickImage** = QPixmap(GameFramework::imagesPath() + "brickbreaker/brick1.png");

int **brickWidth** = brickImage.width() + 1;

int **brickHeight** = brickImage.height() + 1;

for (int **row** = 0; row < 5; row++) {

for (int **col** = 0; col < 10; col++) {

Sprite\* **pBrick1** = new Sprite(brickImage);

pBrick1->setData(SPRITE\_TYPE\_KEY, BRICK);

m\_pScene->addSpriteToScene(*pBrick1*, 60 + col \* brickWidth, 90 + row \* brickHeight);

m\_brickCount++;

}

}

…

}

Détecter l’appui (ou plus exactement le relâchement) de la touche espace, pour démarrer le jeu si le jeu est en mode **WAITING\_FOR\_START**.

void GameCore::**keyReleased**(int **key**) {

emit notifyKeyReleased(key);

m\_paddleDirection = 0;

if (key == Qt::Key\_Space && m\_gameMode == WAITING\_FOR\_START) {

m\_gameMode = RUNNING;

}

}

Enfin, adapter la méthode tick() pour tenir compte du mode de jeu dans le déplacement des différents sprites.

void GameCore::**tick**(long long **elapsedTimeInMilliseconds**) {

// Calcul de la distance de déplacement de la raquette

double **paddleMovement** = m\_paddleDirection \*

PADDLE\_SPEED \*

elapsedTimeInMilliseconds / 1000.0;

m\_pPaddle->setX(m\_pPaddle->x() + paddleMovement);

// Détecte si la raquette dépasse un des bords de la scène et la repositionne le cas échéant.

if (m\_pPaddle->right() > m\_pScene->width()) {

m\_pPaddle->setX(m\_pScene->width() - m\_pPaddle->width());

} else if (m\_pPaddle->left() < 0) {

m\_pPaddle->setX(0);

}

if (m\_gameMode == WAITING\_FOR\_START) {

// En attente du départ, la balle est posée sur la raquette

m\_pBall->setPos(m\_pPaddle->x() + (m\_pPaddle->width() - m\_pBall->width()) / 2.0,

m\_pPaddle->top() - m\_pBall->height());

} else if (m\_gameMode == RUNNING) {

// Création d'un vecteur de déplacement de la balle.

QPointF **ballMovement** = m\_ballDirection \* BALL\_SPEED \* elapsedTimeInMilliseconds / 1000.0;

m\_pBall->setPos(m\_pBall->pos() + ballMovement);

// Si la balle tombe en dessous de la raquette, fin du jeu

if (m\_pBall->top() > m\_pPaddle->bottom()) {

m\_gameMode = ENDED\_LOSE;

} else {

QList<Sprite\*> **collisions** = m\_pScene->collidingSprites(m\_pBall);

if (!collisions.isEmpty()) {

// Collision avec une élément, on détermine le rectangle d'intersection

Sprite\* **pCollisionned** = collisions.at(0);

QRectF **intersect** = m\_pBall->globalBoundingRect().intersected(pCollisionned->globalBoundingRect());

if (intersect.width() < intersect.height())

// Rectangle plus haut que large : collision sur un flanc vertical

m\_ballDirection.setX(m\_ballDirection.x() \* -1);

else

// Rectangle plus large que haut : collision sur un flanc horizontal

m\_ballDirection.setY(m\_ballDirection.y() \* -1);

// Si collision avec une brique : on la détruit

if (pCollisionned->data(SPRITE\_TYPE\_KEY).toInt() == BRICK) {

m\_pScene->removeSpriteFromScene(*pCollisionned*);

delete pCollisionned;

pCollisionned = nullptr;

m\_brickCount--;

if (m\_brickCount == 0)

m\_gameMode = ENDED\_WIN;

}

} else {

// Détecte si la balle atteint un des bords de la scène

if (m\_pBall->right() > m\_pScene->width() || m\_pBall->left() < 0) {

m\_ballDirection.setX(m\_ballDirection.x() \* -1);

} else if (m\_pBall->top() < 0 || m\_pBall->bottom() > m\_pScene->height()) {

m\_ballDirection.setY(m\_ballDirection.y() \* -1);

}

}

}

}

}

## Écrans de début et de fin

Avec les modifications apportées au paragraphe précédent, lorsque le jeu se termine, par une défaite (**ENDED\_LOSE**) ou par une victoire (**ENDED\_WIN**), rien de spécial ne se passe.

De même, lorsque le jeu est en attente que le joueur appuie sur la barre d’espace pour démarrer la partie (**WAITING\_FOR\_START**) rien de spécial n’est affiché pour que le joueur sache ce qu’il doit faire.

Pour clarifier cela, nous afficherons un message au-dessus de la surface de jeu, adapté à l’état dans lequel se trouve le jeu.

La mise en place de cette gestion des différents modes de jeu va également nous obliger à réorganiser notre code.

### Affichage d’un message



Nous allons commencer par implémenter deux méthodes, permettant respectivement d’afficher un message au centre de l’écran de jeu et de l’effacer.

Puisque ce message fera partie intégrante de la scène, ce sera un élément spécialisé de QGraphicsItem, en l’occurrence un QGraphicsSimpleTextItem.

Ce message sera affiché pendant un certain temps, puis il faudra l’effacer. Il est donc nécessaire de mémoriser un pointeur sur ce message.

Ce pointeur ne servant qu’à supprimer le message, il pourra être de type QGraphicsItem.

Le fichier d’entête **gamecore.h** peut être modifié ainsi :

…

class **GameCanvas**;

class **GameScene**;

class **Sprite**;

class **QGraphicsItem**;

…

private:

void **displayInformation**(const QString& **rMessage**);

void **clearInformation**();

enum **GameMode** {

**WAITING\_FOR\_START**,

**RUNNING**,

**ENDED\_LOSE**,

**ENDED\_WIN**

};

GameCanvas\* **m\_pGameCanvas** = nullptr;

GameScene\* **m\_pScene** = nullptr;

Sprite\* **m\_pBall** = nullptr;

Sprite\* **m\_pPaddle** = nullptr;

QGraphicsItem\* **m\_pDisplayedInformation** = nullptr;

QPointF **m\_ballDirection** = QPointF(1,1);

int **m\_paddleDirection** = 0;

…

Les méthodes displayInformation() et clearInformation() seront implémentées ainsi :

//! Affiche le message d'information donné au centre de l'espace de jeu.

//! Si un message est déjà affiché, il est remplacé par ce nouveau message.

//! \param rMessage Message à afficher.

void GameCore::**displayInformation**(const QString& **rMessage**) {

clearInformation();

// Affichage du message en gras

QGraphicsSimpleTextItem\* **pText** = m\_pScene->createText(QPointF(0,0),

rMessage,

50,

Qt::red);

QFont **boldFont** = pText->font();

boldFont.setBold(true);

pText->setFont(boldFont);

// Centrage du texte

pText->setPos((m\_pScene->width() - pText->*boundingRect*().width()) / 2,

m\_pScene->height()/2);

m\_pDisplayedInformation = pText;

}

//! Efface le message d'information.

void GameCore::**clearInformation**() {

if (m\_pDisplayedInformation != nullptr) {

delete m\_pDisplayedInformation;

m\_pDisplayedInformation = nullptr;

}

}

Il est maintenant possible d’afficher un message et de l’effacer selon le mode de jeu.

GameCore::**GameCore**(GameCanvas\* **pGameCanvas**, QObject\* **pParent**) : QObject(*pParent*) {

…

// Ajout de briques

QPixmap **brickImage** = QPixmap(GameFramework::imagesPath() + "brickbreaker/brick1.png");

int **brickWidth** = brickImage.width() + 1;

int **brickHeight** = brickImage.height() + 1;

for (int **row** = 0; row < 5; row++) {

for (int **col** = 0; col < 10; col++) {

Sprite\* **pBrick1** = new Sprite(brickImage);

m\_pScene->addSpriteToScene(*pBrick1*, 60 + col \* brickWidth, 90 + row \* brickHeight);

m\_brickCount++;

}

}

displayInformation("Appuyez sur espace pour commencer");

…

}

Lorsque le jeu démarre, ce message doit être effacé.

Détecter l’appui (ou plus exactement le relâchement) de la touche espace, pour démarrer le jeu si le jeu est en mode **WAITING\_FOR\_START**.

void GameCore::**keyReleased**(int **key**) {

emit notifyKeyReleased(key);

m\_paddleDirection = 0;

if (key == Qt::Key\_Space && m\_gameMode == WAITING\_FOR\_START) {

m\_gameMode = RUNNING;

clearInformation();

}

}

Il faut également afficher un message lorsque le joueur perd ou lorsqu’il gagne.

void GameCore::**tick**(long long **elapsedTimeInMilliseconds**) {

…

// Si la balle tombe en dessous de la raquette, fin du jeu

if (m\_pBall->top() > m\_pPaddle->bottom()) {

m\_gameMode = ENDED\_LOSE;

displayInformation("Perdu !");

} else {

….

// Si collision avec une brique : on la détruit

if (pCollisionned->data(SPRITE\_TYPE\_KEY).toInt() == BRICK) {

m\_pScene->removeSpriteFromScene(*pCollisionned*);

delete pCollisionned;

pCollisionned = nullptr;

m\_brickCount--;

if (m\_brickCount == 0) {

m\_gameMode = ENDED\_WIN;

displayInformation("Victoire !");

}

…

}

### Recommencer une partie

Dans l’état actuel de notre jeu, lorsqu’une partie est terminée (à la suite d’une défaite ou d’une victoire), il n’est pas possible de recommencer.

Pour pouvoir recommencer une partie, il faut mettre en place les éléments suivants :

* Être capable d’effacer les briques restantes et de recréer un niveau de briques ;
* Indiquer au joueur quoi faire pour recommencer une partie (par exemple, appuyer sur *espace*) ;
* Revenir au mode de jeu **WAITING\_FOR\_START**.

Cela implique de créer deux méthodes supplémentaires, permettant respectivement d’effacer les briques restantes à la suite d’une défaite et de créer un nouveau niveau de briques.

Il faut donc déclarer ces deux méthodes dans **gamecore.h** :

private:

void **clearLevel**();

void **createLevel**();

void **displayInformation**(const QString& **rMessage**);

void **clearInformation**();

Ces deux méthodes seront implémentées ainsi :

//! Efface toutes les briques restantes.

void GameCore::**clearLevel**() {

// On passe en revue tous les sprites de la scène. Si c'est une

// brique, elle est retirée de la scène et effacée.

for (Sprite\* **pSprite** : m\_pScene->sprites()) {

if (pSprite->data(SPRITE\_TYPE\_KEY).toInt() == BRICK) {

m\_pScene->removeSpriteFromScene(*pSprite*);

delete pSprite;

pSprite = nullptr;

}

}

m\_brickCount = 0;

}

//! Construit un niveau de briques.

void GameCore::**createLevel**() {

clearLevel();

QPixmap **brickImage** = QPixmap(GameFramework::imagesPath() + "brickbreaker/brick1.png");

int **brickWidth** = brickImage.width() + 1;

int **brickHeight** = brickImage.height() + 1;

for (int **row** = 0; row < 5; row++) {

for (int **col** = 0; col < 10; col++) {

Sprite\* **pBrick1** = new Sprite(brickImage);

pBrick1->setData(SPRITE\_TYPE\_KEY, BRICK);

m\_pScene->addSpriteToScene(*pBrick1*, 60 + col \* brickWidth, 90 + row \* brickHeight);

m\_brickCount++;

}

}

}

Pour éviter de dupliquer le code, il faut modifier le constructeur de GameCore pour qu’il utilise la méthode createLevel() :

…

// Ajout de briques

~~QPixmap~~ **~~brickImage~~** ~~= QPixmap(GameFramework::imagesPath() + "brickbreaker/brick1.png");~~

~~int~~ **~~brickWidth~~** ~~= brickImage.width() + 1;~~

~~int~~ **~~brickHeight~~** ~~= brickImage.height() + 1;~~

~~for (int~~ **~~row~~** ~~= 0; row < 5; row++) {~~

~~for (int~~ **~~col~~** ~~= 0; col < 10; col++) {~~

~~Sprite\*~~ **~~pBrick1~~** ~~= new Sprite(brickImage);~~

~~m\_pScene->addSpriteToScene(~~*~~pBrick1~~*~~, 60 + col \* brickWidth, 90 + row \* brickHeight);~~

~~m\_brickCount++;~~

~~}~~

~~}~~

createLevel();

…

Il faut maintenant adapter les messages affichés lorsque le jeu se termine, afin d’expliquer au joueur qu’il faut appuyer sur espace pour recommencer. C’est la méthode GameCore::tick() qui est concernée.

…

if (m\_pBall->top() > m\_pPaddle->bottom()) {

m\_gameMode = ENDED\_LOSE;

displayInformation("Perdu ! Appuyez sur espace pour recommencer");

…

if (m\_brickCount == 0) {

m\_gameMode = ENDED\_WIN;

displayInformation("Victoire ! Appuyez sur espace pour recommencer");

}

…

Finalement, il faut compléter la méthode GameCore::keyReleased() pour tenir compte de façon plus complète du mode du jeu lorsque la touche espace est appuyée.

//! Traite le relâchement d'une touche.

//! \param key Numéro de la touche (voir les constantes Qt)

void GameCore::**keyReleased**(int **key**) {

emit notifyKeyReleased(key);

m\_paddleDirection = 0;

if (key == Qt::Key\_Space) {

switch (m\_gameMode) {

case WAITING\_FOR\_START:

m\_gameMode = RUNNING;

clearInformation();

break;

case ENDED\_LOSE:

case ENDED\_WIN:

m\_gameMode = WAITING\_FOR\_START;

clearInformation();

createLevel();

break;

}

}

Nous obtenons ainsi un jeu très simple, mais fonctionnel.

### Mettre le jeu en pause

La classe GameCanvas gère la cadence du jeu et se charge de propager cette cadence aux classes GameCore et GameScene. Pour mettre le jeu en pause, il y a principalement deux solutions :

* Stopper la cadence au moyen de la méthode GameCanvas::stopTick(), c’est la solution la plus simple.
* Gérer la propagation de la cadence aux classes GameCore et GameScene en modifiant le code de GameCanvas::onTick().

La première solution est la plus simple et peut être mise en œuvre dans GameCore::keyReleased().

//! Traite le relâchement d'une touche.

//! \param key Numéro de la touche (voir les constantes Qt)

void GameCore::**keyReleased**(int **key**) {

emit notifyKeyReleased(key);

m\_paddleDirection = 0;

if (key == Qt::Key\_Space) {

switch (m\_gameMode) {

case WAITING\_FOR\_START:

m\_gameMode = RUNNING;

clearInformation();

break;

case ENDED\_LOSE:

case ENDED\_WIN:

m\_gameMode = WAITING\_FOR\_START;

clearInformation();

createLevel();

break;

case RUNNING:

if (m\_pGameCanvas->isTicking()) {

m\_pGameCanvas->stopTick();

displayInformation("Press space to continue");

} else {

clearInformation();

m\_pGameCanvas->startTick();

}

break;

}

}

}

# Agrémentations

Nous pouvons encore ajouter quelques fioritures à notre jeu, ce qui permettra d’aborder quelques facettes supplémentaires du GameFramework.

## Programmer la destruction d’un sprite

On pourrait vouloir ajouter, lorsque la balle passe en dessous de la raquette, une animation d’explosion.

Cette explosion sera un sprite éphémère, qui prendra la position de la balle, affichera une suite d’images représentant l’explosion, puis devra être détruit.

Cela implique de réaliser les opérations suivantes :

* Lorsque la partie est perdue, créer le sprite d’explosion ;
* Positionner le sprite à la place de la balle et afficher l’animation ;
* Détecter la fin de l’animation afin de détruire ce sprite éphémère.

Un sprite peut émettre le signal animationFinished lorsque son animation est terminée.

De plus, un objet Qt (qui hérite de QObject et implémente la macro Q\_OBJECT) peut être placé dans une « liste » d’objets à détruire par Qt lorsqu’un cycle de gestion d’événements est terminé. C’est le slot deleteLater qui fait ça.

L’astuce est donc de connecter la signal animationFinished au slot deleteLater. Ainsi, dès que l’animation de l’explosion est terminée, le sprite sera automatiquement détruit.

L’idée est donc de créer une méthode privée createExplosion() dans la classe GameCore, qui prend en paramètre la position où doit s’afficher cette explosion.

//! Affiche une explosion à la position donnée.

//! Cette méthode démontre comment créer un sprite qui affiche une animation et

//! qui se détruit automatiquement dès que l'animation est terminée.

//! \param pos Position de l'explosion.

void GameCore::**createExplosion**(QPointF **pos**) {

Sprite\* **pExplosion** = new Sprite();

for (int **imageNumber** = 1; imageNumber <= 7; imageNumber++) {

pExplosion->addAnimationFrame(GameFramework::imagesPath() +

QString("brickbreaker/blue\_explosion%1.png").arg(imageNumber));

}

pExplosion->setAnimationSpeed(25);

pExplosion->setPos(pos);

// Faire en sorte qu'à la fin de l'animation, le sprite est automatiquement détruit.

pExplosion->setEmitSignalEndOfAnimationEnabled(true);

connect(pExplosion, &Sprite::animationFinished, pExplosion, &Sprite::deleteLater);

// Démarrage de l'animation

m\_pScene->addSpriteToScene(*pExplosion*);

pExplosion->startAnimation();

}

On constate les éléments suivants en lisant ce code :

* Comme l’animation est constituée de sept images, on crée une boucle qui va lire les sept images et les ajouter à l’animation du sprite ;
* Par défaut, le signal animationFinished n’est pas émis, il faut donc enclencher cette mission.

Désormais, lorsque le joueur perd, il faut cacher le sprite de la balle et afficher l’explosion. Cela se fait dans GameCore::tick() :

…

// Si la balle tombe en dessous de la raquette, fin du jeu

if (m\_pBall->top() > m\_pPaddle->bottom()) {

m\_gameMode = ENDED\_LOSE;

displayInformation("Perdu ! Appuyez sur espace pour recommencer");

m\_pBall->hide();

createExplosion(m\_pBall->pos());

} else {

…

Finalement, lorsqu’une partie recommence, il ne faut pas oublier de rendre la balle visible. Cela se fait dans GameCore::keyReleased() :

…

case ENDED\_WIN:

m\_gameMode = WAITING\_FOR\_START;

clearInformation();

createLevel();

m\_pBall->show();

break;

…

## Changer le point de positionnement

On constate que l’animation fonctionne, mais il subsiste un petit problème : elle ne semble pas se produire parfaitement là où se trouvait la balle.

Ceci est dû au point de positionnement, qui est situé par défaut sur le coin supérieur gauche du sprite (§4.1) :

Pour que l’explosion semble se produire exactement là où se trouvait la balle, on ne peut pas utiliser telle quelle la position de la balle pour la position de l’explosion, il faut tenir compte d’un décalage pour que le centre de l’explosion soit situé sur le centre de la balle.

Ce décalage peut être mis en place de deux façons.

### Décalage calculé



Il est assez facile de calculer ce décalage : le long de l’axe X, il correspond à la moitié de la différence de largeur et le long de l’axe Y, il correspond à la moitié de la différence de hauteur.

Ce décalage pourrait donc être calculé au moment de positionner l’explosion. Le problème, c’est que lorsqu’on appelle la méthode createExplosion(), la position de la balle ainsi que ses dimensions sont connues, mais pas encore les dimensions de l’explosion puisque le sprite n’existe pas encore.

Et une fois dans la méthode explosion, on peut connaître les dimensions de l’explosion, mais sans avoir accès aux dimensions de la balle.

Une des solutions à ce problème serait de passer à la méthode createExplosion() un sprite sur lequel l’explosion doit se positionner plutôt qu’une position.

//! Affiche une explosion à l'emplacement du sprite donné.

//! Cette méthode démontre comment créer un sprite qui affiche une animation et

//! qui se détruit automatiquement dès que l'animation est terminée.

//! \param pSprite Sprite sur lequel l'explosion doit se positionner.

void GameCore::**createExplosion**(Sprite\* **pSprite**) {

Sprite\* **pExplosion** = new Sprite();

for (int **imageNumber** = 1; imageNumber <= 7; imageNumber++) {

pExplosion->addAnimationFrame(GameFramework::imagesPath() +

QString("brickbreaker/blue\_explosion%1.png").arg(imageNumber));

}

pExplosion->setAnimationSpeed(25);

// Positionne le centre de l'explosion sur le centre du sprite qui explose

pExplosion->setPos(pSprite->x()-(pExplosion->width()-pSprite->width())/2,

pSprite->y()-(pExplosion->height()-pSprite->height())/2);

// Faire en sorte qu'à la fin de l'animation, le sprite est automatiquement détruit.

pExplosion->setEmitSignalEndOfAnimationEnabled(true);

connect(pExplosion, &Sprite::animationFinished, pExplosion, &Sprite::deleteLater);

// Démarrage de l'animation

m\_pScene->addSpriteToScene(*pExplosion*);

pExplosion->startAnimation();

}

Il ne reste plus qu’à adapter l’appel à la méthode createExplosion() depuis GameCore::tick() :

…

createExplosion(*m\_pBall*);

…

Cette solution est raisonnable lorsque le calcul ne doit être effectué qu’à un seul endroit (ou éventuellement à quelques endroits). Mais si ce genre de calcul doit être répété et dupliqué à de nombreux endroits dans le code, alors cette solution est à proscrire.

### Décalage du point de positionnement

Une autre solution est de décaler le point de positionnement pour qu’il soit placé au centre du sprite, aussi bien pour la balle que pour l’explosion.

Concrètement, ce n’est pas le point de positionnement qui va être déplacé, mais un décalage qui sera ajouté au sprite, de sorte que son point de positionnement corresponde à son centre.



Lors de la construction de la balle (dans le constructeur de GameCore), le décalage sera mis en place grâce à la méthode setOffset().

…

// Instancier et initialiser les sprites ici :

Sprite\* **pBall** = new Sprite(GameFramework::imagesPath() + "brickbreaker/orb\_1.png");

m\_pScene->addSpriteToScene(*pBall*);

pBall->setOffset(-pBall->width()/2, -pBall->height()/2);

pBall->setPos(-m\_pScene->width()/2.0, -m\_pScene->height()/2.0);

pBall->addAnimationFrame(GameFramework::imagesPath() + "brickbreaker/orb\_2.png");

…

La méthode createExplosion() peut revenir à sa signature précédente qui prend en paramètre un point de positionnement, et mettre en place le décalage de l’explosion.

//! Affiche une explosion à la position donnée.

//! Cette méthode démontre comment créer un sprite qui affiche une animation et

//! qui se détruit automatiquement dès que l'animation est terminée.

//! \param pos Position de l'explosion.

void GameCore::**createExplosion**(QPointF **pos**) {

Sprite\* **pExplosion** = new Sprite();

for (int **imageNumber** = 1; imageNumber <= 7; imageNumber++) {

pExplosion->addAnimationFrame(GameFramework::imagesPath() +

QString("brickbreaker/blue\_explosion%1.png").arg(imageNumber));

}

pExplosion->setAnimationSpeed(25);

pExplosion->setOffset(-pExplosion->width()/2, -pExplosion->height()/2);

pExplosion->setPos(pos);

// Faire en sorte qu'à la fin de l'animation, le sprite est automatiquement détruit.

pExplosion->setEmitSignalEndOfAnimationEnabled(true);

connect(pExplosion, &Sprite::animationFinished, pExplosion, &Sprite::deleteLater);

// Démarrage de l'animation

m\_pScene->addSpriteToScene(*pExplosion*);

pExplosion->startAnimation();

}

Enfin il faut revenir à la version précédente de l’appel à createExplosion() dans GameCore::tick().

…

// Si la balle tombe en dessous de la raquette, fin du jeu

if (m\_pBall->top() > m\_pPaddle->bottom()) {

m\_gameMode = ENDED\_LOSE;

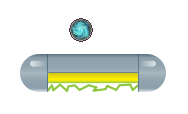
displayInformation("Perdu ! Appuyez sur espace pour recommencer");

m\_pBall->hide();

createExplosion(m\_pBall->pos());

} else {

…



Le décalage du point de positionnement induit cependant un autre problème : a façon dont la balle est positionnée sur la raquette lorsque le jeu est en attente du démarrage d’une partie.

En effet, la position calculée, qui correspondait au coin supérieur gauche de la balle, correspond maintenant à son centre.

Il faut donc adapter le code qui se charge du suivi de la balle lorsque la raquette bouge, dans GameCore::tick().

…

if (m\_gameMode == WAITING\_FOR\_START) {

// En attente du départ, la balle est posée sur la raquette

m\_pBall->setPos(m\_pPaddle->x() + m\_pPaddle->width()/ 2,

m\_pPaddle->top() - m\_pBall->height()/2);

…

## Cheat mode

Lorsqu’on développe un jeu il est souvent utile, pour pouvoir tester correctement toutes les facettes du jeu, d’avoir un mode de triche (*cheat mode*).

Nous allons rapidement en implémenter un, qui s’activera très simplement lorsque la touche c sera appuyée et qui fera en sorte que la raquette suive automatiquement la balle.

Pour commencer, il faut pouvoir mémoriser si le *cheat mode* est enclenché, nous aurons besoin pour cela d’ajouter une variable membre privée à GameCore :

bool **m\_cheatModeEnabled** = false;

Ensuite lorsque la touche c est relâchée, on bascule le mode triche. La méthode keyReleased() est remaniée pour utiliser des instructions switch :

void GameCore::**keyReleased**(int **key**) {

emit notifyKeyReleased(key);

m\_paddleDirection = 0;

switch(key) {

case Qt::Key\_Space:

switch (m\_gameMode) {

case WAITING\_FOR\_START:

m\_gameMode = RUNNING;

clearInformation();

break;

case ENDED\_LOSE:

case ENDED\_WIN:

m\_gameMode = WAITING\_FOR\_START;

clearInformation();

createLevel();

m\_pBall->show();

break;

case RUNNING:

if (m\_pGameCanvas->isTicking()) {

m\_pGameCanvas->stopTick();

displayInformation("Press space to continue");

} else {

clearInformation();

m\_pGameCanvas->startTick();

}

}

break;

case Qt::Key\_C:

m\_cheatModeEnabled = !m\_cheatModeEnabled;

break;

}

}

Enfin dans GameCore::tick(), il faut tenir compte du *cheat mode* pour ajuster la position de la raquette :

…

} else if (m\_gameMode == RUNNING) {

// Création d'un vecteur de déplacement de la balle.

QPointF **ballMovement** = m\_ballDirection \* BALL\_SPEED \* elapsedTimeInMilliseconds / 1000.0;

m\_pBall->setPos(m\_pBall->pos() + ballMovement);

// CHEAT MODE

if (m\_cheatModeEnabled)

m\_pPaddle->setX(m\_pBall->x() - m\_pPaddle->width()/2);

// Si la balle tombe en dessous de la raquette, fin du jeu

if (m\_pBall->top() > m\_pPaddle->bottom()) {

…

## Fond d’écran

La méthode GameScene::setBackgroundImage() permet de définir une image qui sera utilisée comme motif de fond de la scène.

Cette image peut être définie dans le constructeur de GameCore :

m\_pScene->setBackgroundImage(QImage(GameFramework::imagesPath()+"brickbreaker/space.jpg"));

# Déploiement

Pour qu’une application puisse s’exécuter, il ne suffit pas d’avoir le fichier exécutable. Plusieurs autres fichiers sont nécessaires pour son bon fonctionnement. Entre autres :

* Les bibliothèques utilisées par l’application (sous Windows, les fameuses *dll*s) ;
* Les ressources (images, sons) ;
* Les éventuels fichiers de configuration ;
* Les éventuels autres outils utilisés (base de données, drivers, etc.).

De plus, il faut s’assurer que notre application soit capable de localiser ces fichiers sur le disque.

Sur votre poste de développement, tous ces outils sont à disposition, installés et configurés. Le système d’exploitation est configuré pour que l’accès à tous les fichiers nécessaires soit possible.

Malheureusement, si notre application est installée sur un poste de travail qui n’a pas été forcément prévu pour le développement, il est fort probable que tous les fichiers requis ne soient pas présents ou correctement configurés.

L’étape de déploiement consiste donc à s’assurer que lors de l’installation de notre application sur un poste quelconque, tous les outils, fichiers et ressources nécessaires au bon fonctionnement de l’application soient présents et accessibles.

Qt nous fournit la commande windeployqt.exe pour aider au déploiement d’application pour Windows.

L’annexe 14.5 revient en détail sur la problématique du déploiement.

## Script de déploiement

En général, le déploiement est pris en charge de façon automatique par un script, qui se charge des tâches suivantes :

* Compiler et générer l’exécutable (en mode **release**) ;
* Rassembler dans un dossier spécifique l’exécutable, les ressources et les bibliothèques (dlls) requises ;
* Générer un installer (ou créer un fichier zip).

Le projet GameFramework contient le script generate\_release.bat qui se charge de réaliser ces principales étapes, et rassemble tous les fichiers nécessaires dans un dossier release.

Il peut facilement être adapté à votre jeu.

# Suite du travail

## Problèmes à solutionner

Dans son état actuel, les bases du jeu sont posées, mais il reste un certain nombre de petits bugs qui devraient être solutionnés :

* Si le *tick* dure trop longtemps (par exemple lors d’un redimensionnement de la fenêtre), la balle va faire d’un coup une longue distance et peut sortir de la scène pour ne plus jamais y revenir ;
* Si la balle va trop vite, la collision peut être trop avancée dans le sens où la balle se retrouve complètement dans la brique. Il n’y aura donc plus moyen de savoir si c’est une collision sur un flanc vertical ou horizontal ;
* En cas de collision avec le coin d’une brique, le rectangle d’intersection est aussi large que haut. Cette situation est-elle véritablement problématique ?
* La balle peut potentiellement toucher deux briques simultanément ;
* L’appui sur plusieurs touches est mal géré car la raquette peut se bloquer alors qu’une touche de déplacement est encore enfoncée ;
* En cas de défaite, il faudrait réinitialiser le vecteur de déplacement de la balle, sans quoi elle repart avec sa dernière direction (et donc à priori vers le bas).
* Actuellement, la balle se déplace toujours selon un angle de 45°, sa vitesse est donc constante. Mais si cet angle est appelé à varier, il faudra faire quelques calculs pour garantir une vitesse constante quelle que soit le vecteur de déplacement :



## Améliorations possibles

* Modifier l’angle de rebond de la balle selon l’emplacement du point d’impact sur la raquette ;
* Ajouter un certain nombre de vies ;
* Créer différents types de briques (incassables, cassable en plusieurs chocs) ;
* Augmentation de la vitesse de la balle ;
* Briques laissant tomber des bonus à attraper, par exemple :
  + Agrandissement temporaire de la raquette ;
  + Suivi temporaire automatique de la balle ;
  + Ajout d’une balle supplémentaire ;
  + Balle temporaire destructrice (qui détruit toutes les briques sur son passage et ne rebondit que sur les murs et la raquette ;
  + Filet d’un filet de protection temporaire ;
  + Etc.
* Briques laissant tomber des malus, par exemple :
  + Rapetissement de la raquette ;
  + Accélération de la balle ;
  + Apparition de nouvelles briques
  + Etc.
* Ajout d’un système de score et de meilleur score ;
* Ajout de niveaux ;
* Ajout d’un système de lecture de niveaux à partir d’un fichier de configuration plutôt qu’en les codant en dur ;
* Etc.

# Autres concepts

## Identifier un sprite par son type exact

C’est généralement en cas de collision qu’il est intéressant de connaître le type exact d’un sprite. Nous avons vu une première façon de le faire en utilisant une donnée distinctive (§7.1.1).

Une autre solution, plus proche de la philosophie orienté-objet, repose sur la spécialisation de la classe Sprite et l’utilisation de l’instruction de conversion dynamic\_cast<>().

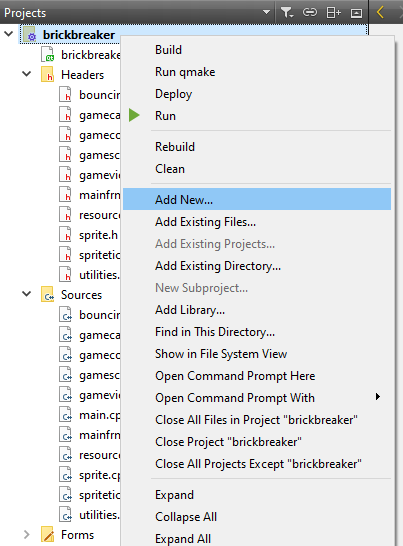
Elle est donc particulièrement utile pour les projets qui tirent parti de l’héritage d’un Sprite.

Imaginons que nous voulions faire la hiérarchie de classes suivante, afin de distinguer les briques des autres sprites, par exemple pour y ajouter certaines propriétés spécifiques aux briques.



Il faut donc ajouter à notre projet une nouvelle classe Brick, qui hérite de Sprite.

Cela se fait facilement avec Qt Creator en choisissant Add New.. dans le menu contextuel de notre projet, puis en indiquant que l’on souhaite créer une nouvelle classe.

 Une image contenant texte

Description générée automatiquement

Il ne reste alors plus qu’à indiquer le nom de notre classe et de préciser qu’elle hérite de la classe Sprite.

Une image contenant texte

Description générée automatiquement

Pour l’instant, exception faite du constructeur, notre classe peut rester vide, seul son type nous intéresse :

**brick.h :**

#ifndef BRICK\_H

#define **BRICK\_H**

#include "sprite.h"

class **Brick** : public Sprite

{

public:

**Brick**(const QPixmap& **rPixmap**, QGraphicsItem\* **pParent** = nullptr);

};

#endif // BRICK\_H

**brick.cpp :**

#include "brick.h"

Brick::**Brick**(const QPixmap& **rPixmap**, QGraphicsItem\* **pParent**) : Sprite(rPixmap, *pParent*)

{

}

Le nouveau type Brick étant créé, il peut être utilisé dans GameCore lors de l’instanciation des briques :

//! Construit un niveau de briques.

void GameCore::**createLevel**() {

clearLevel();

QPixmap **brickImage** = QPixmap(GameFramework::imagesPath() + "brickbreaker/brick1.png");

int **brickWidth** = brickImage.width() + 1;

int **brickHeight** = brickImage.height() + 1;

for (int **row** = 0; row < 5; row++) {

for (int **col** = 0; col < 10; col++) {

~~Sprite\*~~ **~~pBrick1~~** ~~= new Sprite(brickImage);~~

Sprite\* **pBrick1** = new Brick(brickImage);

pBrick1->setData(SPRITE\_TYPE\_KEY, BRICK);

m\_pScene->addSpriteToScene(*pBrick1*, 60 + col \* brickWidth, 90 + row \* brickHeight);

m\_brickCount++;

}

}

}

Pour que cette modification fonctionne, il ne faut pas oublier d’inclure le fichier brick.h dans GameCore :

#include "brick.h"

Nous pouvons ensuite utiliser l’instruction dynamic\_cast<>() pour tenter de convertir le pointeur sur le sprite en collision en un pointeur sur une brique. Si cette conversion fonctionne, nous obtenons un pointeur valide (donc différent de nullptr), ce qui nous permet de garantir que le sprite en collision est bel et bien une brique.

void GameCore::**tick**(long long **elapsedTimeInMilliseconds**) {

…

// Si collision avec une brique : on la détruit

~~if (pCollisionned->data(SPRITE\_TYPE\_KEY).toInt() == BRICK) {~~

if (dynamic\_cast<Brick\*>(pCollisionned) != nullptr) {

m\_pScene->removeSpriteFromScene(*pCollisionned*);

delete pCollisionned;

pCollisionned = nullptr;

m\_brickCount--;

if (m\_brickCount == 0) {

m\_gameMode = ENDED\_WIN;

displayInformation("Victoire ! Appuyez sur espace pour recommencer");

}

…

}

Cette solution nous permet de faire du nettoyage dans le code de la classe GameCore afin de supprimer tout ce qui concerne les données de sprite.

**gamecore.cpp :**

~~…~~

~~// Enumérés~~

~~enum~~ **~~SpriteDataKey~~** ~~{~~

**~~SPRITE\_TYPE\_KEY~~** ~~= 0~~

~~};~~

~~enum~~ **~~SpriteType~~** ~~{~~

**~~BRICK~~**~~,~~

**~~PADDLE~~**~~,~~

**~~BALL~~**

~~};~~

~~…~~

GameCore::**GameCore**(GameCanvas\* **pGameCanvas**, QObject\* **pParent**) : QObject(*pParent*) {

…

~~pBall->setData(SPRITE\_TYPE\_KEY, BALL);~~

…

~~pPaddle->setData(SPRITE\_TYPE\_KEY, PADDLE);~~

…

}

…

//! Efface toutes les briques restantes.

void GameCore::**clearLevel**() {

// On passe en revue tous les sprites de la scène. Si c'est une

// brique, elle est retirée de la scène et effacée.

for (Sprite\* **pSprite** : m\_pScene->sprites()) {

if (dynamic\_cast<Brick\*>(pSprite) != nullptr) {

~~if (pSprite->data(SPRITE\_TYPE\_KEY).toInt() == BRICK) {~~

m\_pScene->removeSpriteFromScene(*pSprite*);

delete pSprite;

pSprite = nullptr;

}

}

m\_brickCount = 0;

}

…

//! Construit un niveau de briques.

void GameCore::**createLevel**() {

clearLevel();

QPixmap **brickImage** = QPixmap(GameFramework::imagesPath() + "brickbreaker/brick1.png");

int **brickWidth** = brickImage.width() + 1;

int **brickHeight** = brickImage.height() + 1;

for (int **row** = 0; row < 5; row++) {

for (int **col** = 0; col < 10; col++) {

Sprite\* **pBrick1** = new Brick(brickImage);

~~pBrick1->setData(SPRITE\_TYPE\_KEY, BRICK);~~

m\_pScene->addSpriteToScene(*pBrick1*, 60 + col \* brickWidth, 90 + row \* brickHeight);

m\_brickCount++;

}

}

}

## Orienté-objet ? Déplacement par un *tick handler*

Il est parfois délicat d’appliquer à un jeu les règles de l’art de la programmation orientée-objets.

Dans l’état actuel de notre application, mis à part les différents éléments graphiques (GameScene, GameCanvas, Sprite), le code que nous avons écrit dans GameCore n’est pas véritablement orienté-objet puisque toute la logique du jeu se trouve sous forme procédurale (impérative) dans une seule et unique classe et plus particulièrement dans la méthode tick() dont on pourrait reprocher la longueur.

Toutefois la particularité d’un jeu fait qu’on a souvent besoin d’une classe qui joue le rôle de centre de contrôle et qui doit connaître et gérer l’intégralité du jeu. Il est donc parfois difficile (et souvent au prix d’une performance moindre) de déléguer les responsabilités de GameCore à d’autres classes.

Il est toutefois possible, dans certains cas simples de déporter la logique du déplacement d’un sprite dans une classe dédiée.

Cette façon de faire est plus proche de la philosophie orientée-objet, permet dans une certaine mesure la réutilisation de code, mais ajoute une complexité dans la communication entre les objets et le partage des responsabilités qui n’est pas toujours adaptée aux spécificités d’un jeu.

Nous allons voir comment adapter notre jeu pour en faire usage.

### Le *tick handler*

L’idée est de confier la responsabilité du déplacement d’un sprite à une classe dédiée, appelée ici un *tick handler*, ou *gestionnaire de tick*.

Le GameFramework propose une classe de base, SpriteTickHandler, qui peut être spécialisée pour gérer un déplacement selon une **stratégie** particulière, puis **affectée** à un sprite pour que celui-ci se déplace selon cette stratégie.

Lorsqu'un gestionnaire est affecté à un sprite (Sprite::setTickHandler()), la fonction init() du gestionnaire est appelée une **unique** fois, puis, lorsque la cadence démarre, la fonction tick() du gestionnaire est **automatiquement** appelée.

Depuis le gestionnaire, il est possible d'accéder au sprite en question avec l'attribut m\_pParentSprite.

Concrètement, dans notre jeu, nous avons trois stratégies de déplacement :

* Déplacement de la balle par rebond
* Déplacement de la balle lorsqu’elle est posée sur la raquette
* Déplacement de la raquette

Il s’agit donc de créer pour chaque stratégie de déplacement une nouvelle classe, puis d’affecter au sprite la stratégie de déplacement adaptée.

### Balle posée sur la raquette



Il faut créer une nouvelle classe, nommée LinkedTickHandler, qui hérite de SpriteTickHandler et qui aura pour rôle de s’assurer que le sprite qu’elle contrôle (m\_pParentSprite) soit toujours posé sur un sprite de référence (m\_pRefSprite) passé au constructeur du gestionnaire.

Afin d’éviter de dupliquer du code, ce positionnement sera réalisé dans la méthode linkPosition().

**linkedtickhandler.h :**

#ifndef LINKEDTICKHANDLER\_H

#define **LINKEDTICKHANDLER\_H**

#include "spritetickhandler.h"

class **LinkedTickHandler** : public SpriteTickHandler

{

public:

**LinkedTickHandler**(Sprite\* **pRefSprite**, Sprite\* **pParentSprite** = nullptr);

virtual void ***init***() override;

virtual void ***tick***(long long **elapsedTimeInMilliseconds**) override;

private:

void **linkPosition**();

Sprite\* **m\_pRefSprite**;

};

#endif // LINKEDTICKHANDLER\_H

**linkedtickhandler.cpp :**

#include "linkedtickhandler.h"

#include "sprite.h"

//! Constructeur.

//! \param pRefSprite Sprite sur lequel reposera le sprite géré.

//! \param pParentSprite Sprite dont le déplacement doit être géré.

//!

LinkedTickHandler::**LinkedTickHandler**(Sprite\* **pRefSprite**,

Sprite\* **pParentSprite**) : SpriteTickHandler(*pParentSprite*) {

m\_pRefSprite = pRefSprite;

}

//! Ajuste la position initial du sprite géré.

void LinkedTickHandler::***init***() {

linkPosition();

}

//! Cadence

void LinkedTickHandler::***tick***(long long **elapsedTimeInMilliseconds**) {

linkPosition();

}

//! Déplace le sprite géré afin qu'il soit posé sur le sprite de référence.

void LinkedTickHandler::**linkPosition**() {

m\_pParentSprite->setPos(m\_pRefSprite->x() + m\_pRefSprite->width()/ 2,

m\_pRefSprite->top() - m\_pParentSprite->height()/2);

}

### Rebond



Il faut maintenant créer une nouvelle classe, nommée BouncingTickHandler, qui hérite de SpriteTickHandler et qui aura pour rôle de s’assurer que le sprite qu’elle contrôle (m\_pParentSprite) rebondisse sur les sprites avec lesquelles il collisionne.

Le code du déplacement du sprite placé dans la méthode tick() reprend en grande partie le code de déplacement écrit précédemment dans GameCore::tick(), mais sans les vérifications de fin de jeu (par défaite ou par victoire).

**bouncingtickhandler.h :**

#ifndef BOUNCINGTICKHANDLER\_H

#define **BOUNCINGTICKHANDLER\_H**

#include "spritetickhandler.h"

#include <QPointF>

//! \brief Classe qui gère le déplacement d'un sprite.

//!

class **BouncingTickHandler** : public SpriteTickHandler

{

public:

B**ouncingTickHandler**(Sprite\* **pParentSprite** = nullptr);

virtual void ***tick***(long long **elapsedTimeInMilliseconds**) override;

private:

QPointF **m\_spriteDirection**;

};

#endif // BOUNCINGTICKHANDLER\_H

**bouncingtickhandler.cpp :**

#include "bouncingtickhandler.h"

#include "sprite.h"

#include "gamescene.h"

const int **INITIAL\_VELOCITY** = 600;

//! Constructeur.

//! \param pParentSprite Sprite dont le déplacement doit être géré.

BouncingTickHandler::**BouncingTickHandler**(Sprite\* **pParentSprite**) : SpriteTickHandler (*pParentSprite*) {

}

//! Cadence : détermine le mouvement que fait le sprite durant le temps écoulé,

//! vérifie si il doit rebondir et le positionne à son nouvel emplacement.

void BouncingTickHandler::***tick***(long long **elapsedTimeInMilliseconds**) {

QPointF **spriteMovement** = m\_spriteDirection \* INITIAL\_VELOCITY \*

elapsedTimeInMilliseconds / 1000.0;

// Création d'un vecteur de déplacement de la balle.

m\_pParentSprite->setPos(m\_pParentSprite->pos() + spriteMovement);

QList<Sprite\*> **collisions** = parentScene()->collidingSprites(m\_pParentSprite);

if (collisions.isEmpty()) {

// Pas de collision : on s'assure que le sprite reste dans les limites

// de la scène.

if (m\_pParentSprite->right() > parentScene()->width()) {

m\_spriteDirection.setX(-1);

m\_pParentSprite->setX(parentScene()->width()-m\_pParentSprite->width());

} else if (m\_pParentSprite->left() < 0) {

m\_spriteDirection.setX(1);

m\_pParentSprite->setX(0);

}

if (m\_pParentSprite->bottom() > parentScene()->height()) {

m\_spriteDirection.setY(-1);

m\_pParentSprite->setY(parentScene()->height()-m\_pParentSprite->height());

} else if (m\_pParentSprite->top() < 0) {

m\_spriteDirection.setY(1);

m\_pParentSprite->setY(0);

}

} else {

// Collision avec une élément, on détermine le rectangle d'intersection

Sprite\* **pCollisionned** = collisions.at(0);

QRectF **intersect** = m\_pParentSprite->globalBoundingRect()

.intersected(pCollisionned->globalBoundingRect());

if (intersect.width() < intersect.height())

// Rectangle plus haut que large : collision sur un flanc vertical

m\_spriteDirection.setX(m\_spriteDirection.x() \* -1);

else

// Rectangle plus large que haut : collision sur un flanc horizontal

m\_spriteDirection.setY(m\_spriteDirection.y() \* -1);

}

}

### Affectation d’un gestionnaire à un sprite

Nous avons maintenant à disposition les deux gestionnaires permettant de gérer le déplacement de la balle. Nous allons donc adapter GameCore afin de les utiliser.

Tout d’abord, inclure les fichiers d’entête de ces deux gestionnaires :

…

#include "bouncingtickhandler.h"

#include "brick.h"

#include "gamescene.h"

#include "gamecanvas.h"

#include "linkedtickhandler.h"

…

Dans le constructeur, il faut assigner à la balle le gestionnaire initial, c’est-à-dire celui qui pose la balle sur la raquette. Mais pour qu’un gestionnaire fonctionne, il a besoin de la cadence de jeu.

Par défaut, un sprite n’est pas connecté à la cadence du jeu. Pour que le gestionnaire de tick affecté à un sprite fonctionne, il faut indiquer à la scène que le sprite en question doit recevoir la cadence du jeu :

m\_pScene->registerSpriteForTick(*pSprite*);

Le sprite peut également s’enregistrer lui-même auprès de la scène s’il a été préalablement ajouté à la scène, grâce à la méthode Sprite::registerForTick().

…

GameCore::**GameCore**(GameCanvas\* **pGameCanvas**, QObject\* **pParent**) : QObject(*pParent*) {

…

Paddle\* **pPaddle** = new Paddle(GameFramework::imagesPath() + "brickbreaker/paddle1.png");

m\_pScene->addSpriteToScene(*pPaddle*);

pPaddle->setPos(m\_pScene->width()/2.0, m\_pScene->height()-100);

pPaddle->addAnimationFrame(GameFramework::imagesPath() + "brickbreaker/paddle2.png");

pPaddle->addAnimationFrame(GameFramework::imagesPath() + "brickbreaker/paddle3.png");

pPaddle->startAnimation(100);

m\_pPaddle = pPaddle;

// Au lancement du jeu, la balle est posée sur la raquette

m\_pBall->setTickHandler(new LinkedTickHandler(*m\_pPaddle*));

m\_pBall->registerForTick();

…

}

Lorsque la touche espace est relâchée, le mode de jeu change. Cela implique de changer le mode de déplacement de la balle.

void GameCore::**keyReleased**(int **key**) {

emit notifyKeyReleased(key);

m\_paddleDirection = 0;

switch(key) {

case Qt::Key\_Space:

switch (m\_gameMode) {

case WAITING\_FOR\_START:

m\_gameMode = RUNNING;

clearInformation();

m\_pBall->setTickHandler(new BouncingTickHandler);

break;

case ENDED\_LOSE:

case ENDED\_WIN:

m\_gameMode = WAITING\_FOR\_START;

clearInformation();

createLevel();

m\_pBall->show();

m\_pBall->setTickHandler(new LinkedTickHandler(*m\_pPaddle*));

break;

case RUNNING:

if (m\_pGameCanvas->isTicking()) {

m\_pGameCanvas->stopTick();

displayInformation("Press space to continue");

} else {

clearInformation();

m\_pGameCanvas->startTick();

}

}

break;

case Qt::Key\_C:

m\_cheatModeEnabled = !m\_cheatModeEnabled;

break;

}

}

La méthode GameCore::tick() n’a plus à gérer le déplacement de la balle, mais est toujours responsable du déplacement de la raquette.

void GameCore::**tick**(long long **elapsedTimeInMilliseconds**) {

// Calcul de la distance de déplacement de la raquette

double **paddleMovement** = m\_paddleDirection \* PADDLE\_SPEED \*

elapsedTimeInMilliseconds / 1000.0;

m\_pPaddle->setX(m\_pPaddle->x() + paddleMovement);

// Détecte si la raquette dépasse un des bords de la scène et la repositionne le cas échéant.

if (m\_pPaddle->right() > m\_pScene->width()) {

m\_pPaddle->setX(m\_pScene->width() - m\_pPaddle->width());

} else if (m\_pPaddle->left() < 0) {

m\_pPaddle->setX(0);

}

// CHEAT MODE

if (m\_cheatModeEnabled)

m\_pPaddle->setX(m\_pBall->x() - m\_pPaddle->width()/2);

~~if (m\_gameMode == WAITING\_FOR\_START) {~~

~~// En attente du départ, la balle est posée sur la raquette~~

~~//m\_pBall->setPos(m\_pPaddle->x() + m\_pPaddle->width()/ 2,~~

~~m\_pPaddle->top() - m\_pBall->height()/2);~~

~~} else if (m\_gameMode == RUNNING) {~~

~~// Création d'un vecteur de déplacement de la balle.~~

~~QPointF ballMovement = m\_ballDirection \* BALL\_SPEED \* elapsedTimeInMilliseconds / 1000.0;~~

~~m\_pBall->setPos(m\_pBall->pos() + ballMovement);~~

~~// CHEAT MODE~~

~~if (m\_cheatModeEnabled)~~

~~m\_pPaddle->setX(m\_pBall->x() - m\_pPaddle->width()/2);~~

~~// Si la balle tombe en dessous de la raquette, fin du jeu~~

~~if (m\_pBall->top() > m\_pPaddle->bottom()) {~~

~~m\_gameMode = ENDED\_LOSE;~~

~~m\_pBall->hide();~~

~~displayInformation("Perdu ! Appuyez sur espace pour recommencer");~~

~~createExplosion(m\_pBall->pos());~~

~~} else {~~

~~QList<Sprite\*> collisions = m\_pScene->collidingSprites(m\_pBall);~~

~~if (!collisions.isEmpty()) {~~

~~// Collision avec une élément, on détermine le rectangle d'intersection~~

~~Sprite\* pCollisionned = collisions.at(0);~~

~~QRectF intersect = m\_pBall->globalBoundingRect().intersected(pCollisionned->globalBoundingRect());~~

~~if (intersect.width() < intersect.height())~~

~~// Rectangle plus haut que large : collision sur un flanc vertical~~

~~m\_ballDirection.setX(m\_ballDirection.x() \* -1);~~

~~else~~

~~// Rectangle plus large que haut : collision sur un flanc horizontal~~

~~m\_ballDirection.setY(m\_ballDirection.y() \* -1);~~

~~// Si collision avec une brique : on la détruit~~

~~if (dynamic\_cast<Brick\*>(pCollisionned) != nullptr) {~~

~~m\_pScene->removeSpriteFromScene(pCollisionned);~~

~~delete pCollisionned;~~

~~pCollisionned = nullptr;~~

~~m\_brickCount--;~~

~~if (m\_brickCount == 0) {~~

~~m\_gameMode = ENDED\_WIN;~~

~~displayInformation("Victoire ! Appuyez sur espace pour recommencer");~~

~~}~~

~~}~~

~~} else {~~

~~if (m\_pBall->right() > m\_pScene->width()) {~~

~~m\_ballDirection.setX(-1);~~

~~m\_pBall->setX(m\_pScene->width()-m\_pBall->width());~~

~~} else if (m\_pBall->left() < 0) {~~

~~m\_ballDirection.setX(1);~~

~~m\_pBall->setX(0);~~

~~}~~

~~if (m\_pBall->bottom() > m\_pScene->height()) {~~

~~m\_ballDirection.setY(-1);~~

~~m\_pBall->setY(m\_pScene->height()-m\_pBall->height());~~

~~} else if (m\_pBall->top() < 0) {~~

~~m\_ballDirection.setY(1);~~

~~m\_pBall->setY(0);~~

~~}~~

~~}~~

~~}~~

~~}~~

}

Dans l’état actuel de notre projet, la balle se déplace désormais de façon autonome, sans que GameCore ait à s’en préoccuper dans la méthode tick(). Cette dernière s’en voit donc largement simplifiée.

Toutefois ces changements ont provoqué plusieurs régressions :

* Les briques touchées par la balle ne sont plus détruites ;
* Le jeu ne se termine pas lorsque la balle tombe sous la raquette ;
* Le jeu ne se termine pas lorsqu’il n’y a plus de briques.

Pour corriger le premier problème, il faut malheureusement sacrifier le potentiel de réutilisation du gestionnaire BouncingTickHandler, afin de le rendre plus spécifique à notre jeu de sorte que lorsqu’une brique est touchée, elle est détruite.

**bouncingtickhandler.cpp :**

#include "bouncingtickhandler.h"

#include "brick.h"

#include "sprite.h"

#include "gamescene.h"

…

void BouncingTickHandler::***tick***(long long **elapsedTimeInMilliseconds**) {

…

} else {

// Collision avec une élément, on détermine le rectangle d'intersection

Sprite\* **pCollisionned** = collisions.at(0);

QRectF **intersect** = m\_pParentSprite->globalBoundingRect().intersected(pCollisionned->globalBoundingRect());

if (intersect.width() < intersect.height())

// Rectangle plus haut que large : collision sur un flanc vertical

m\_spriteDirection.setX(m\_spriteDirection.x() \* -1);

else

// Rectangle plus large que haut : collision sur un flanc horizontal

m\_spriteDirection.setY(m\_spriteDirection.y() \* -1);

// Si collision avec une brique : on la détruit

if (dynamic\_cast<Brick\*>(pCollisionned) != nullptr) {

parentScene()->removeSpriteFromScene(*pCollisionned*);

delete pCollisionned;

pCollisionned = nullptr;

}

}

}

Les briques sont maintenant détruites, mais subsiste le problème que GameCore doit garder à jour le compteur de briques (m\_brickCount) afin de détecter une fin de jeu victorieuse.

Heureusement, chaque fois qu’un sprite est retiré de la scène, la classe GameScene émet le signal spriteRemovedFromScene avec en paramètre le sprite qui est retiré.

Il suffit donc que GameCore se connecte à ce signal afin d’être informé chaque fois qu’un sprite est enlevé, et si ce sprite est une brique, le compteur peut être mis à jour.

Pour se connecter à un signal, il faut que GameCore implémente un slot prévu à cet effet.

**gamecore.h :**

…

private slots:

void **onSpriteRemovedFromScene**(Sprite\* **pSprite**);

…

Dans le constructeur de GameCore, il faut établir la connexion entre le signal émis par la scène et ce slot, puis implémenter le slot. Si toutes les briques ont été détruites, on enlève le gestionnaire de tick de la balle pour qu’elle arrête de rebondir.

**gamecore.cpp :**

GameCore::**GameCore**(GameCanvas\* **pGameCanvas**, QObject\* **pParent**) : QObject(*pParent*) {

…

// Au lancement du jeu, la balle est posée sur la raquette

m\_pBall->setTickHandler(new LinkedTickHandler(*m\_pPaddle*));

m\_pBall->registerForTick();

// Ajout de briques

createLevel();

displayInformation("Appuyez sur espace pour commencer");

connect(m\_pScene, &GameScene::spriteRemovedFromScene,

this, &GameCore::onSpriteRemovedFromScene);

}

//! Traite le retrait d'un sprite de la scène.

//! Si c'est une brique et que le jeu est en cours, on vérifie si le niveau est gagné.

//! \param pSprite Sprite qui a été retiré de la scène.

void GameCore::**onSpriteRemovedFromScene**(Sprite\* **pSprite**) {

if (dynamic\_cast<Brick\*>(pSprite) != nullptr && m\_gameMode == RUNNING) {

m\_brickCount--;

if (m\_brickCount == 0) {

m\_gameMode = ENDED\_WIN;

displayInformation("Victoire ! Appuyez sur espace pour recommencer");

m\_pBall->removeTickHandler();

}

}

}

Il ne reste maintenant qu’à vérifier si la balle passe au dessous de la raquette. Cela peut être fait dans la méthode GameCore::tick().

void GameCore::**tick**(long long **elapsedTimeInMilliseconds**) {

// Calcul de la distance de déplacement de la raquette

double **paddleMovement** = m\_paddleDirection \* PADDLE\_SPEED \*

elapsedTimeInMilliseconds / 1000.0;

m\_pPaddle->setX(m\_pPaddle->x() + paddleMovement);

// Détecte si la raquette dépasse un des bords de la scène et la repositionne le cas échéant.

if (m\_pPaddle->right() > m\_pScene->width()) {

m\_pPaddle->setX(m\_pScene->width() - m\_pPaddle->width());

} else if (m\_pPaddle->left() < 0) {

m\_pPaddle->setX(0);

}

// CHEAT MODE

if (m\_cheatModeEnabled)

m\_pPaddle->setX(m\_pBall->x() - m\_pPaddle->width()/2);

// Si la balle tombe en dessous de la raquette, fin du jeu

if (m\_gameMode == RUNNING && m\_pBall->top() > m\_pPaddle->bottom()) {

m\_gameMode = ENDED\_LOSE;

m\_pBall->hide();

displayInformation("Perdu ! Appuyez sur espace pour recommencer");

createExplosion(m\_pBall->pos());

m\_pBall->removeTickHandler();

}

}

### Autres gestionnaires

À noter que le projet de base GameFramework propose différents gestionnaires de tick pour illustrer différentes façons de gérer le déplacement d’un sprite : BouncingSpriteHandler, KeyTickHandler, RandomMoveTickHandler.

## Vérifier avant de déplacer

Lors du déplacement d’un sprite, nous avons vu comment détecter d’éventuelles collisions (§6).

Nous avons procédé selon la méthode suivante : **Déplacer** le sprite, **puis** **vérifier** s’il collisionne avec un autre sprite.

Cette méthode n’est pas toujours idéale et il serait parfois préférable de vérifier, **avant** de **déplacer**, si une collision **aurait** lieu.

Pour faire cela, il faut tout d’abord déterminer son prochain emplacement, et **vérifier**, **avant de faire un déplacement**, si ce prochain emplacement entre en collision avec un autre sprite et alors, par exemple, renoncer à déplacer le sprite.

L’idée de cette méthode de détection des collisions est de récupérer le rectangle qui englobe le sprite (rectangle d’encombrement, *bounding rect* ou *boîte d’encombrement*) avec la méthode Sprite::globalBoundingRect() et de déplacer ce **rectangle** (méthode translated()) **plutôt** que le **sprite** lui-même.



Ensuite, il faut vérifier si ce rectangle collisionne avec un ou plusieurs sprite. Si ce n’est pas le cas, le sprite lui-même peut être déplacé.

La scène permet de fournir la liste des sprites qui sont en collision avec un rectangle donné (en l’occurrence un *bounding rect*) grâce à la méthode collidingSprites() :

Depuis GameCore, la scène est accessible grâce à la variable membre m\_pScene.

**Attention** : dans la plupart des cas, le rectangle **collisionne** avec le **sprite** qui va être **déplacé** lui-même. Il est donc important d’ignorer cette collision-là, par exemple en retirant explicitement de la liste obtenue, le sprite qui se déplace, grâce à la méthode removeAll().

Pour vérifier si un rectangle est intégralement contenu au sein de la scène, la classe GameScene met à disposition la méthode isInsideScene(). Cela permet de s’assurer que le sprite ne sorte pas de la scène.

La solution d’utiliser le *bounding rect* pour détecter une collision a l’avantage d’être assez rapide, mais selon les besoins, peut manquer de précision puisqu’elle est une approximation.

Si une détection de collision au pixel près est nécessaire, il ne faut plus utiliser le *bounding rect*, mais la forme exacte du sprite, avec la méthode Sprite::globalShape() à la place de la méthode Sprite::globalBoundingRect().

### Vérifier le rebond avant de déplacer la balle

Nous allons modifier le gestionnaire BouncingTickHandler afin qu’il vérifie la collision avant de déplacer la balle. Nous verrons que cela élimine de nombreux petits bugs de déplacement par rapport à la façon actuelle de faire.

void BouncingTickHandler::***tick***(long long **elapsedTimeInMilliseconds**) {

QPointF **spriteMovement** = m\_spriteDirection \* INITIAL\_VELOCITY \*

elapsedTimeInMilliseconds / 1000.0;

// Détermine la prochaine position du sprite

QRectF **nextSpriteRect** = m\_pParentSprite->globalBoundingRect().translated(spriteMovement);

// Récupère tous les sprites de la scène que toucherait ce sprite à sa prochaine position

auto **collisions** = m\_pParentSprite->parentScene()->collidingSprites(nextSpriteRect);

// Supprimer le sprite lui-même, qui collisionne toujours avec sa boundingbox

collisions.removeAll(*m\_pParentSprite*);

if (collisions.isEmpty()) {

// Pas de collision : on s'assure que le sprite reste dans les limites

// de la scène.

bool **moveSprite** = true;

if (nextSpriteRect.right() > parentScene()->width()) {

m\_spriteDirection.setX(-1);

moveSprite = false;

} else if (nextSpriteRect.left() < 0) {

m\_spriteDirection.setX(1);

moveSprite = false;

}

if (nextSpriteRect.bottom() > parentScene()->height()) {

m\_spriteDirection.setY(-1);

moveSprite = false;

} else if (nextSpriteRect.top() < 0) {

m\_spriteDirection.setY(1);

moveSprite = false;

}

// Si le mouvement du sprite est possible, il est déplacé

if (moveSprite)

m\_pParentSprite->setPos(m\_pParentSprite->pos() + spriteMovement);

} else {

// Collision avec un élément, on détermine le rectangle d'intersection

Sprite\* **pCollisionned** = collisions.at(0);

QRectF **intersect** = m\_pParentSprite->globalBoundingRect()

.intersected(pCollisionned->globalBoundingRect());

if (intersect.width() < intersect.height())

// Rectangle plus haut que large : collision sur un flanc vertical

m\_spriteDirection.setX(m\_spriteDirection.x() \* -1);

else

// Rectangle plus large que haut : collision sur un flanc horizontal

m\_spriteDirection.setY(m\_spriteDirection.y() \* -1);

// Si collision avec une brique : on la détruit

if (dynamic\_cast<Brick\*>(pCollisionned) != nullptr) {

parentScene()->removeSpriteFromScene(*pCollisionned*);

delete pCollisionned;

pCollisionned = nullptr;

}

}

}

## Gestionnaire de tick et clavier

Une image contenant intérieur, sombre

Description générée automatiquement

Nous avons vu comment gérer le déplacement de la balle au moyen d’un tick handler (§12.2.1).

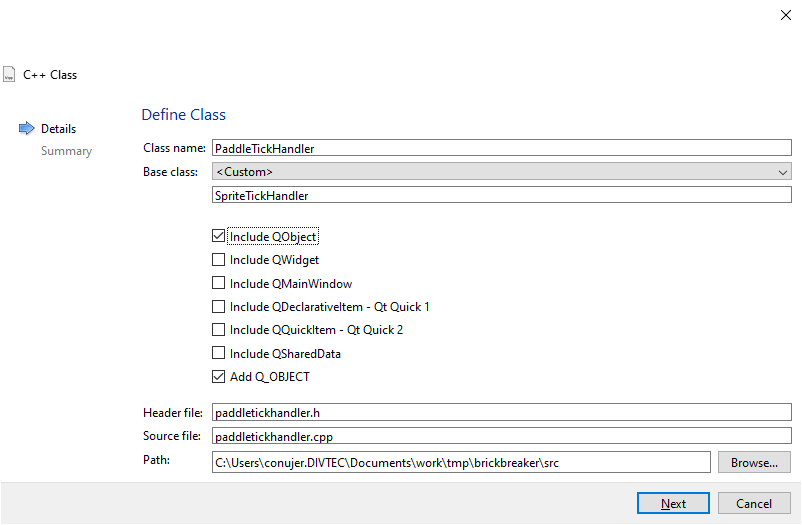
Si nous voulons faire de même pour la raquette, il faut parvenir à gérer, depuis un tick handler, les touches du clavier.

Pour qu’un gestionnaire de tick puisse tenir compte des touches du clavier, il faut qu’il puisse se connecter aux signaux émis par la classe GameCore :

* notifyKeyPressed(int key) et
* notifyKeyReleased(int key).

Ce mécanisme est mis en place par le gestionnaire KeyTickHandler, fourni dans le projet de base de *GameFramework*, mais est néanmoins décrit ici de façon **simplifiée**.

Pour ce faire, nous allons créer une classe PaddleTickHandler qui hérite de SpriteTickHandler. De plus, pour que notre gestionnaire de tick puisse se connecter à un signal, il faut qu’il hérite (en premier) de la classe QObject (ce qui nécessite d’inclure QObject) et que la macro Q\_OBJECT soit activée. Cela se fait dans le fichier d’entête.



La deuxième étape est de créer deux **slots** qui seront connectés aux deux signaux qui nous intéressent, émis par GameCore.

**paddletickhandler.h :**

#ifndef PADDLETICKHANDLER\_H

#define **PADDLETICKHANDLER\_H**

#include "spritetickhandler.h"

#include <QObject>

class **PaddleTickHandler** : public QObject, public SpriteTickHandler

{

Q\_OBJECT

public:

**PaddleTickHandler**(Sprite\* **pParentSprite** = nullptr);

virtual void ***init***() override;

virtual void ***tick***(long long **elapsedTimeInMilliseconds**) override;

public slots:

void **onKeyPressed**(int **key**);

void **onKeyReleased**(int **key**);

};

#endif // PADDLETICKHANDLER\_H

**Attention** : la macro Q\_OBJECT nécessite que le compilateur génère un fichier temporaire supplémentaire, appelé fichier moc. Si elle n’est pas directement indiquée au moment de l’ajout de la classe au projet, il faut manuellement exécuter la commande qmake afin de mettre à jour les fichiers du projet, en allant dans le menu Compiler de *Qt Creator* et en sélectionnant l’entrée Exécuter qmake. Si cela n’est pas fait, des messages d’erreur de cette nature seront émis lors de la compilation :

erreur : undefined reference to `vtable for PlayerTickHandler'

Il faut ensuite mémoriser la vitesse de déplacement de la raquette et sa direction. La vitesse initiale sera passée au constructeur par paramètre.

class **PaddleTickHandler** : public QObject, public SpriteTickHandler

{

Q\_OBJECT

public:

**~~PaddleTickHandler~~**~~(Sprite\*~~ **~~pParentSprite~~** ~~= nullptr);~~

**PaddleTickHandler**(double **paddleSpeed**, Sprite\* **pParentSprite** = nullptr);

virtual void ***init***() override;

virtual void ***tick***(long long **elapsedTimeInMilliseconds**) override;

public slots:

void **onKeyPressed**(int **key**);

void **onKeyReleased**(int **key**);

private:

double **m\_paddleSpeed** = 0;

int **m\_paddleDirection** = 0;

};

L’implémentation est relativement simple :

**paddletickhandler.cpp :**

#include "paddletickhandler.h"

#include "gamescene.h"

#include "sprite.h"

//! Constructeur

//! \param paddleSpeed Vitesse de déplacement du paddle.

//! \param pParentSprite Sprite dont le déplacement doit être géré.

PaddleTickHandler::**PaddleTickHandler**(double **paddleSpeed**, Sprite\* **pParentSprite**) : SpriteTickHandler(*pParentSprite*) {

m\_paddleSpeed = paddleSpeed;

}

//! Initialisation du gestionnaire.

void PaddleTickHandler::***init***() {

}

//! Cadence : détermine le mouvement que fait le sprite durant le temps écoulé.

void PaddleTickHandler::***tick***(long long **elapsedTimeInMilliseconds**) {

// Calcul de la distance de déplacement de la raquette

double **paddleMovement** = m\_paddleDirection \* m\_paddleSpeed \* elapsedTimeInMilliseconds / 1000.0;

m\_pParentSprite->setX(m\_pParentSprite->x() + paddleMovement);

// Détecte si la raquette dépasse un des bords de la scène et la repositionne le cas échéant.

if (m\_pParentSprite->right() > parentScene()->width()) {

m\_pParentSprite->setX(parentScene()->width() - m\_pParentSprite->width());

} else if (m\_pParentSprite->left() < 0) {

m\_pParentSprite->setX(0);

}

}

//! Une touche a été appuyée.

//! \param key Code de la touche appuyée.

void PaddleTickHandler::**onKeyPressed**(int **key**) {

switch(key) {

case Qt::Key\_Left: m\_paddleDirection = -1; break;

case Qt::Key\_Right: m\_paddleDirection = 1; break;

}

}

//! Une touche a été relâchée.

//! \param key Code de la touche relâchée.

void PaddleTickHandler::**onKeyReleased**(int **key**) {

m\_paddleDirection = 0;

}

Le problème de l’implémentation proposée ci-dessus survient lorsque plusieurs touches sont appuyées simultanément, mais qu’une seule est relâchée : le sprite ne bouge plus alors qu’une touche est encore appuyée.

Le gestionnaire de tick KeyTickHandler propose une solution pour gérer ce problème.

Finalement, il faut encore mettre à jour l’implémentation de GameCore pour réaliser les tâches suivantes :

* Affecter à la raquette un gestionnaire de tick PaddleTickHandler ;
* Connecter les signaux GameCore::notifyKeyPressed et GameCore::notifyKeyReleased aux slots correspondants du PaddleTickHandler ;
* Supprimer le code de déplacement de la raquette, désormais inutile.

GameCore::**GameCore**(GameCanvas\* **pGameCanvas**, QObject\* **pParent**) : QObject(*pParent*) {

…

// Au lancement du jeu, la balle est posée sur la raquette

m\_pBall->setTickHandler(new LinkedTickHandler(*m\_pPaddle*));

m\_pBall->registerForTick();

PaddleTickHandler\* **pPaddleHandler** = new PaddleTickHandler(PADDLE\_SPEED);

m\_pPaddle->setTickHandler(*pPaddleHandler*);

m\_pPaddle->registerForTick();

// connection des événements claviers au gestionnaire de tick

connect(this, &GameCore::notifyKeyPressed,

pPaddleHandler, &PaddleTickHandler::onKeyPressed);

connect(this, &GameCore::notifyKeyReleased,

pPaddleHandler, &PaddleTickHandler::onKeyReleased);

// Ajout de briques

createLevel();

…

}

void GameCore::**keyPressed**(int **key**) {

emit notifyKeyPressed(key);

~~switch(key) {~~

~~case Qt::Key\_Left: m\_paddleDirection = -1; break;~~

~~case Qt::Key\_Right: m\_paddleDirection = 1; break;~~

~~}~~

}

void GameCore::**keyReleased**(int **key**) {

emit notifyKeyReleased(key);

~~m\_paddleDirection = 0;~~

…

}

void GameCore::**tick**(long long **elapsedTimeInMilliseconds**) {

~~// Calcul de la distance de déplacement de la raquette~~

~~double paddleMovement = m\_paddleDirection \* PADDLE\_SPEED \*~~

~~elapsedTimeInMilliseconds / 1000.0;~~

~~m\_pPaddle->setX(m\_pPaddle->x() + paddleMovement);~~

~~// Détecte si la raquette dépasse un des bords de la scène et la repositionne le cas échéant.~~

~~if (m\_pPaddle->right() > m\_pScene->width()) {~~

~~m\_pPaddle->setX(m\_pScene->width() - m\_pPaddle->width());~~

~~} else if (m\_pPaddle->left() < 0) {~~

~~m\_pPaddle->setX(0);~~

~~}~~

// CHEAT MODE

if (m\_cheatModeEnabled)

m\_pPaddle->setX(m\_pBall->x() - m\_pPaddle->width()/2);

…

}

# Concepts avancés

**Remarque préliminaire :** les exemples de code présentés dans cette section se basent sur un projet *GameFramework* vide et sont indépendants du jeu de casse-briques développés dans les chapitres précédents.

Ils utilisent néanmoins la classe KeyTickHandler fournie par la version de démonstration de *GameFramework*, qu’il faut donc ajouter au projet, en copiant les fichiers keytickhandler.h et keytickhandler.cpp de *GameFramework* dans le dossier src du nouveau projet et en ajoutant ces deux fichiers au projet *Qt* en sélectionnant le menu contextuel de *Qt Creator* Add existing files….

## Adapter l’animation du déplacement

Comme déjà vu au tout début de ce tutoriel, il est possible d’animer un sprite en lui ajoutant plusieurs images grâce à la méthode addAnimationFrame() puis de passer d’une image à une autre en boucle, en utilisant la méthode startAnimation().

|  |  |  |  |
| --- | --- | --- | --- |
| C:\Users\conujer.DIVTEC\AppData\Local\Microsoft\Windows\INetCache\Content.Word\pacman_1.png | C:\Users\conujer.DIVTEC\AppData\Local\Microsoft\Windows\INetCache\Content.Word\pacman_2.png | C:\Users\conujer.DIVTEC\AppData\Local\Microsoft\Windows\INetCache\Content.Word\pacman_1.png | C:\Users\conujer.DIVTEC\AppData\Local\Microsoft\Windows\INetCache\Content.Word\pacman_3.png |
| image 1 | image 2 | image 1 | image 3 |

La vitesse d’animation peut être définie comme un paramètre de startAnimation() ou au moyen de la méthode setAnimationSpeed().

La mise en place du sprite dans le constructeur de GameCore peut être réalisée ainsi :

// Instancier et initialiser les sprites ici :

Sprite\* **pSprite** = new Sprite(GameFramework::imagesPath() + "tutorial/pacman\_1.png");

pSprite->addAnimationFrame(GameFramework::imagesPath() + "tutorial/pacman\_2.png");

pSprite->addAnimationFrame(GameFramework::imagesPath() + "tutorial/pacman\_1.png");

pSprite->addAnimationFrame(GameFramework::imagesPath() + "tutorial/pacman\_3.png");

m\_pScene->addSpriteToScene(*pSprite*);

KeyTickHandler\* **pKeyHandler** = new KeyTickHandler;

// connection des événements claviers au gestionnaire de tick

connect(this, &GameCore::notifyKeyPressed,

pKeyHandler, &KeyTickHandler::onKeyPressed);

connect(this, &GameCore::notifyKeyReleased,

pKeyHandler, &KeyTickHandler::onKeyReleased);

pSprite->setTickHandler(*pKeyHandler*);

pSprite->registerForTick();

pSprite->startAnimation(200); // Démarre l'animation et à l'image suivante toutes

// les 200 millisecondes

Pour que le code ci-dessus fonctionne, il ne faut pas oublier d’inclure sprite.h et keytickhandler.h au fichier gamecore.cpp.

### Appliquer une rotation

Pour rendre l’animation plus conforme au déplacement, il peut être intéressant de changer les images de l’animation selon que le personnage se déplace à gauche, à droite, en haut ou en bas.

Une première solution simple, acceptable dans les cas où l’image est **symétrique**, est de garder les mêmes images, mais de leur appliquer une **rotation**.

Cette rotation peut être appliquée dans le slot qui réagit à la pression sur les touches du clavier, dans notre classe KeyTickHandler :

//! Une touche a été appuyée.

//! \param key Code de la touche appuyée.

void KeyTickHandler::**onKeyPressed**(int **key**) {

// Le switch ne peut pas être utilisé car les valeurs à tester ne sont pas

// des constantes.

if (key == m\_upKey) {

m\_keyUpPressed = true;

m\_pParentSprite->setRotation(-90);

} else if (key == m\_downKey) {

m\_keyDownPressed = true;

m\_pParentSprite->setRotation(90);

} else if (key == m\_rightKey) {

m\_keyRightPressed = true;

m\_pParentSprite->setRotation(0);

} else if (key == m\_leftKey) {

m\_keyLeftPressed = true;

m\_pParentSprite->setRotation(180);

}

updateDirectionVector();

}

Le problème avec cette solution est qu’il faut faire attention au **point de transformation** du sprite, car c’est lui qui sera utilisé comme **centre de rotation**.

Par défaut, le point de transformation est placé en haut à gauche du rectangle d’encombrement (*bounding rect*) du sprite. La rotation s’effectue donc autour de ce point, ce qui produit un effet indésirable.

Ce point peut être déplacé au moyen de la méthode setTransformOriginPoint(). Dans notre cas, il faut le déplacer au centre du rectangle d’encombrement.

Le positionnement de ce point de transformation peut être fait dans la méthode init() de notre gestionnaire de tick KeyTickHandler.

//! Initialisation du gestionnaire.

void KeyTickHandler::***init***() {

// Déplacement du point d'application des transformations au centre du sprite

m\_pParentSprite->setTransformOriginPoint(m\_pParentSprite->width()/2,

m\_pParentSprite->height()/2);

}

### Calculer l’angle de rotation exact

Cette solution est maintenant satisfaisante tant que le sprite se déplace de façon **orthonormée**, c’est-à-dire uniquement dans les quatre directions de base, mais pas en diagonale.

Si l’on veut déterminer l’angle de rotation **exact** selon un **vecteur de déplacement** défini par une composante de déplacement sur l’axe des X et une autre composante sur l’axe des Y, comme c’est le cas avec la variable membre m\_directionVector de la classe KeyTickHandler, il faut faire un peu de trigonométrie.

La tangente de l’angle α se calcule par la formule .

L’angle α se calcule donc par la fonction réciproque *arctangente* (tangente inverse) : .

La fonction std::atan2() de la bibliothèque mathématique cmath calcule l’angle en **radians**. Il faut donc convertir cet angle en **degrés**, par exemple avec la fonction fournie qRadiansToDegrees() par Qt.

Le calcule et l’application de cet angle de rotation sur le sprite peut être centralisé dans la fonction updateDirectionVector() :

//! Met à jour le vecteur de déplacement du sprite en fonction des touches

//! qui sont appuyées.

void KeyTickHandler::**updateDirectionVector**() {

double **XVelocity** = 0;

double **YVelocity** = 0;

if (m\_keyUpPressed) YVelocity = -1.0 ;

if (m\_keyDownPressed) YVelocity = 1.0;

if (m\_keyRightPressed) XVelocity = 1.0;

if (m\_keyLeftPressed) XVelocity = -1-0;

m\_directionVector = QPointF(XVelocity, YVelocity);

// Applique l'angle de rotation sur le sprite selon la direction

m\_pParentSprite->setRotation(qRadiansToDegrees(std::atan2(YVelocity, XVelocity)));

}

Pourquoi std::atan2() plutôt que std::atan() pour calculer l’arctangente ?

La fonction atan() prend en paramètre la tangente de l’angle et ne permet pas de faire la distinction entre des directions diamétralement opposées, ce qui est embêtant dans notre situation.

La fonction atan2() prend en paramètre les composantes x et y ce qui lui permet de prendre en compte le signe des composantes du vecteur et de placer l’angle dans le bon quadrant, ce qui est exactement ce dont on a besoin.

Cela permet de simplifier la fonction onKeyPressed() :

//! Une touche a été appuyée.

//! \param key Code de la touche appuyée.

void KeyTickHandler::**onKeyPressed**(int **key**) {

if (key == m\_upKey) {

m\_keyUpPressed = true;

~~m\_pParentSprite->setRotation(-90);~~

} else if (key == m\_downKey) {

m\_keyDownPressed = true;

~~m\_pParentSprite->setRotation(90);~~

} else if (key == m\_rightKey) {

m\_keyRightPressed = true;

~~m\_pParentSprite->setRotation(0);~~

} else if (key == m\_leftKey) {

m\_keyLeftPressed = true;

~~m\_pParentSprite->setRotation(180);~~

}

updateDirectionVector();

}

**Attention** : lorsque le sprite est en **diagonale**, son rectangle d’encombrement (§6.2) **global** (globalBoundingRect() ) s’agrandi, ce qui peut être surprenant si on n’y prend pas garde, par exemple lors de la détection des collisions.



### Appliquer une matrice de transformation

Parfois, en particulier lorsque le personnage à animer n’est **pas symétrique**, appliquer une rotation ne fait pas l’affaire.

Pour illustrer cela, modifions notre programme pour utiliser comme personnage un bonhomme qui marche à droite ou à gauche. Les images que nous utiliserons représentent le personnage marchant à droite.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| C:\Users\conujer.DIVTEC\AppData\Local\Microsoft\Windows\INetCache\Content.Word\marche1.png |  | C:\Users\conujer.DIVTEC\AppData\Local\Microsoft\Windows\INetCache\Content.Word\marche2.png | Une image contenant texte  Description générée automatiquement | C:\Users\conujer.DIVTEC\AppData\Local\Microsoft\Windows\INetCache\Content.Word\marche4.png | C:\Users\conujer.DIVTEC\AppData\Local\Microsoft\Windows\INetCache\Content.Word\marche6.png |  | C:\Users\conujer.DIVTEC\Documents\work\GameFrameWork_pour_tutorial\res\images\marche7.png |
| marche1.png | marche2.png | marche3.png | marche4.png | marche5.png | marche6.png | marche7.png | marche8.png |

La mise en place du sprite dans le constructeur de GameCore peut être modifiée ainsi :

// Instancier et initialiser les sprites ici :

~~Sprite\* pSprite = new Sprite(GameFramework::imagesPath() + "tutorial/pacman\_1.png");~~

~~pSprite->addAnimationFrame(GameFramework::imagesPath() + "tutorial/pacman\_2.png");~~

~~pSprite->addAnimationFrame(GameFramework::imagesPath() + "tutorial/pacman\_1.png");~~

~~pSprite->addAnimationFrame(GameFramework::imagesPath() + "tutorial/pacman\_3.png");~~

Sprite\* **pSprite** = new Sprite;

for (int **imageIndex** = 1; imageIndex <= 8; imageIndex++) {

pSprite->addAnimationFrame(GameFramework::imagesPath() +

QString("demo/marche%1.png").arg(imageIndex));

}

pSprite->setPos(20,20);

m\_pScene->addSpriteToScene(*pSprite*);

KeyTickHandler\* **pKeyHandler** = new KeyTickHandler;

// connection des événements claviers au gestionnaire de tick

connect(this, &GameCore::notifyKeyPressed,

pKeyHandler, &KeyTickHandler::onKeyPressed);

connect(this, &GameCore::notifyKeyReleased,

pKeyHandler, &KeyTickHandler::onKeyReleased);

pSprite->setTickHandler(*pKeyHandler*);

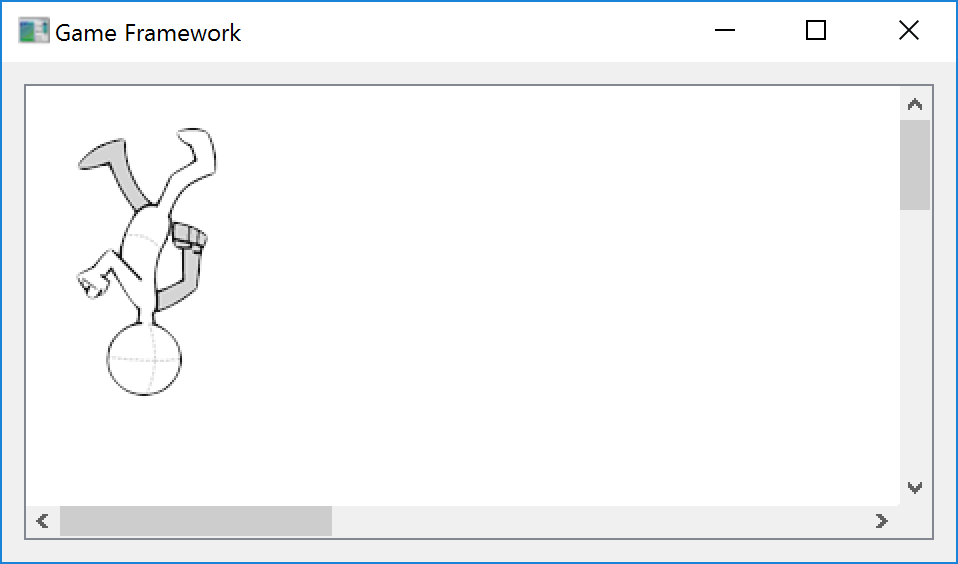
pSprite->registerForTick();

~~pSprite->startAnimation(200); // Démarre l'animation et à l'image suivante toutes//~~

~~// les 200 millisecondes~~

pSprite->startAnimation(50);

On voit bien que lorsque le personnage va à gauche, appliquer une rotation de 180 degrés ne fait pas ce que l’on souhaite obtenir. En l’occurrence, ce n’est pas une rotation dont nous avons besoin mais d’un **effet miroir**.



Qt permet d’appliquer à un élément graphique une **matrice de transformation** (ou plus exactement une liste de matrices de transformation) pour modifier son apparence. Il est possible de changer sa taille (QGraphicsScale) et son angle de rotation (QGraphicsRotation). Ces deux classes héritent de QGraphicsTransform.

L’astuce dans notre cas est de créer une matrice de transformation qui va mettre à l’échelle l’axe X uniquement, avec pour valeur -1. Il suffira d’appliquer cette matrice de transformation lorsque le personnage se déplace à gauche et de la retirer lorsqu’il se déplace à droite.

La matrice de transformation peut être créée une fois pour toute dans la classe KeyTickHandler :

**keytickhandler.h :**

#ifndef KEYTICKHANDLER\_H

#define **KEYTICKHANDLER\_H**

…

class **QGraphicsTransform**;

class **KeyTickHandler** : public QObject, public SpriteTickHandler

{

…

int **m\_rightKey** = Qt::Key\_Right;

int **m\_leftKey** = Qt::Key\_Left;

int **m\_upKey** = Qt::Key\_Up;

int **m\_downKey** = Qt::Key\_Down;

QList<QGraphicsTransform\*> **m\_leftDirectionTransform**;

};

#endif // KEYTICKHANDLER\_H

La méthode init() de KeyTickHandler est un bon emplacement pour créer la matrice de transformation :

//! Initialisation du gestionnaire.

void KeyTickHandler::***init***() {

// Déplacement du point d'application des transformations au centre du sprite

m\_pParentSprite->setTransformOriginPoint(m\_pParentSprite->width()/2,

m\_pParentSprite->height()/2);

// Création d'une matrice de transformation horizontale pour la direction gauche

QGraphicsScale\* **pHorizontalTransform** = new QGraphicsScale(*m\_pParentSprite*);

pHorizontalTransform->setOrigin(QVector3D(m\_pParentSprite->width()/2,0,0));

pHorizontalTransform->setXScale(-1);

// Configuration de la liste de transformations pour le déplacement à gauche

m\_leftDirectionTransform << pHorizontalTransform;

}

Il suffit, lorsque le personnage change de sens de déplacement, de lui affecter la liste de transformations adéquate.

//! Met à jour le vecteur de déplacement du sprite en fonction des touches

//! qui sont appuyées.

void KeyTickHandler::**updateDirectionVector**() {

double **XVelocity** = 0;

double **YVelocity** = 0;

if (m\_keyUpPressed) YVelocity = -1.0 ;

if (m\_keyDownPressed) YVelocity = 1.0;

if (m\_keyRightPressed) {

XVelocity = 1.0;

m\_pParentSprite->setTransformations(QList<QGraphicsTransform\*>());

}

if (m\_keyLeftPressed) {

XVelocity = -1.0;

m\_pParentSprite->setTransformations(m\_leftDirectionTransform);

}

m\_directionVector = QPointF(XVelocity, YVelocity);

~~// Applique l'angle de rotation sur le sprite selon la direction~~

~~m\_pParentSprite->setRotation(qRadiansToDegrees(std::atan2(YVelocity, XVelocity)));~~

}

Il ne faut pas oublier de retirer le code qui applique une rotation !

### Utiliser un jeu d’images différent

Dans d’autres situations plus complexes, les deux solutions présentées ci-dessus ne conviennent pas, car chaque sens de déplacement nécessite son propre jeu d’images. Il faut alors utiliser une autre technique, présentée au paragraphe 13.5 (Listes d’animation).

## Interrompre l’animation en cas d’arrêt

Si l’on veut que l’animation du personnage ne se produise qu’en cas de mouvement, on peut adapter notre code afin que l’animation ne démarre (avec Sprite::startAnimation()) que lorsqu’une touche est appuyée et s’arrête (avec Sprite::stopAnimation()) dès qu’une touche est relâchée.

Lorsqu’on arrête l’animation d’un sprite, on peut choisir de l’arrêter immédiatement (IMMEDIATE\_STOP), ou de terminer le cycle d’images avant de l’arrêter (END\_OF\_CYCLE\_STOP).

Lors de la création du sprite dans le constructeur de gamecore.cpp, le code doit être adapté ainsi afin que l’animation ne soit pas démarrée automatiquement :

GameCore::**GameCore**(GameCanvas\* **pGameCanvas**, QObject\* **pParent**) : QObject(*pParent*) {

…

pSprite->setTickHandler(*pKeyHandler*);

pSprite->registerForTick();

pSprite->setAnimationSpeed(50); // Changement d'image toutes les 50 ms

pSprite->setCurrentAnimationFrame(0); // Affiche la première image du sprite

~~pSprite->startAnimation(50);~~

…

}

Le gestionnaire de tick peut ensuite être modifié ainsi au niveau de la gestion du vecteur de déplacement :

void KeyTickHandler::**updateDirectionVector**() {

double **XVelocity** = 0;

double **YVelocity** = 0;

if (m\_keyUpPressed) YVelocity = -1.0 ;

if (m\_keyDownPressed) YVelocity = 1.0;

if (m\_keyRightPressed) {

XVelocity = 1.0;

m\_pParentSprite->setTransformations(QList<QGraphicsTransform\*>());

}

if (m\_keyLeftPressed) {

XVelocity = -1.0;

m\_pParentSprite->setTransformations(m\_leftDirectionTransform);

}

m\_directionVector = QPointF(XVelocity, YVelocity);

if (m\_directionVector.isNull())

m\_pParentSprite->stopAnimation(Sprite::END\_OF\_CYCLE\_STOP);

else if (!m\_pParentSprite->isAnimationRunning()) {

m\_pParentSprite->startAnimation();

}

}

Cette solution simple n’est pas parfaite (le personnage ne se déplace plus alors que l’animation se termine, si une touche est appuyée pendant que l’animation se termine, l’animation ne redémarre pas, etc.) mais permet néanmoins de démontrer le principe de base.

## Sprite sheet

Une sprite sheet (feuille de sprites) est une image bitmap contenant plusieurs plus petites images agencées au sein d’une grille.

Ces différentes images constituent souvent des animations et/ou les différents éléments de décor ou autre.

Dans un jeu classique, regrouper de petites images au sein d’une plus grande améliore les performances du jeu, réduit l’utilisation de la mémoire et accélère le démarrage du jeu.

Toutefois, dans le cas de *GameFramework*, le système graphique utilisé fait que le seul gain est d’éviter la prolifération des images.

|  |  |
| --- | --- |
|  |  |
| explosion.png | spinningwheel.png |

### Chargement et découpage

Lorsqu’une image est chargée sous forme d’un QPixmap, celle-ci est difficilement manipulable. Il est beaucoup plus pratique de la transformer en une QImage afin d’obtenir davantage de possibilités de manipulation, comme par exemple la méthode copy() qui permet de copier une portion de l’image. Il est ensuite facile de convertir la QImage obtenue en QPixmap.

// Instancier et initialiser les sprites ici :

Sprite\* pSprite = new Sprite;

// Données spécifiques à la spritesheet

const int FRAME\_COUNT = 15; // nombre d'images de l'animation

const int COLUMN\_COUNT = 4; // nombre de colonnes

const int ROW\_COUNT = 4; // nombre de lignes

// Chargement de la spritesheet

QImage spriteSheet(GameFramework::imagesPath() + "explosion.png");

int frameWidth = spriteSheet.width() / COLUMN\_COUNT;

int frameHeight = spriteSheet.height() / ROW\_COUNT;

// Découpage de la spritesheet

for (int frameIndex = 0; frameIndex < FRAME\_COUNT; frameIndex++) {

QImage spriteImage = spriteSheet.copy((frameIndex % COLUMN\_COUNT) \* frameWidth,

(frameIndex / ROW\_COUNT) \* frameHeight,

frameWidth, frameHeight);

pSprite->addAnimationFrame(QPixmap::fromImage(spriteImage));

}

// Mise en place du sprite sur la scène

pSprite->setAnimationSpeed(50);

pSprite->setCurrentAnimationFrame(0);

pSprite->setPos(0,0);

m\_pScene->addSpriteToScene(*pSprite*);

pSprite->startAnimation();

## Scrolling

La scène de jeu (GameScene), dans laquelle évoluent les sprites, est en réalité une surface dont la taille ne dépend pas de l’écran. Elle est affichée au travers d’une vue, implémentée par la classe GameView.

Si la surface de la scène dépasse la surface affichée par la vue, seule une portion de la scène est visible.



La largeur de la scène, en pixels, est mémorisée dans la constante SCENE\_WIDTH, définie dans gamecore.cpp.

Au moment de la création de la scène (dans le constructeur de GameCore), sa hauteur est automatiquement calculée afin que ses proportions soient adaptées à celle de l’écran :

// Créé la scène de base et indique au canvas qu'il faut l'afficher.

m\_pScene = pGameCanvas->createScene(0, 0, SCENE\_WIDTH,

SCENE\_WIDTH / GameFramework::screenRatio());

pGameCanvas->setCurrentScene(*m\_pScene*);

Il est bien sûr possible de donner à la scène des dimensions toutes autres et il n’y a aucune obligation de respecter les proportions de l’écran.

Par exemple, modifions le fichier gamecore.cpp pour que la scène ait une largeur de 2660 pixels.

const int SCENE\_WIDTH = 2560;

En plus du sprite déplacé par les touches, déjà présent sur la scène, nous en profitons pour ajouter quelques éléments de décor afin de pouvoir observer l’effet du scrolling.

// Instancier et initialiser les sprites ici :

Sprite\* pSprite = new Sprite(GameFramework::imagesPath() + "player\_m1.png");

pSprite->setPos(20,20);

m\_pScene->addSpriteToScene(*pSprite*);

pSprite->setTickHandler(new PlayerTickHandler*(*this*)*);

pSprite->registerForTick();

// Ajout d'une lignée de 5 murs de briques

QPixmap WallPixmap(GameFramework::imagesPath() + "brique.jpg");

for (int i = 0; i < 5; i++){

Sprite\* pWall = new Wall(WallPixmap);

m\_pScene->addSpriteToScene(*pWall*, i\*WallPixmap.width(), 250);

}

À ce stade, le problème est que le sprite, s’il sort de la surface affichée par la vue, n’est plus visible.

L’astuce est de centrer la vue sur le sprite, afin qu’il soit toujours visible. Pour cela, la méthode GameScene::centerViewOn() peut être utilisée.

Pour cela, il faut adapter le PlayerTickHandler :

void PlayerTickHandler::***tick***(long long elapsedTimeInMilliseconds) {

// Création d'un vecteur de déplacement du sprite.

QPointF spriteMovement = m\_playerDirection \* PLAYER\_SPEED \* elapsedTimeInMilliseconds / 1000.0;

// Détermine la prochaine position du sprite

QRectF nextSpriteRect = m\_pParentSprite->globalBoundingRect().translated(spriteMovement);

if (m\_pParentSprite->parentScene()->isInsideScene(nextSpriteRect)) {

m\_pParentSprite->setPos(m\_pParentSprite->pos() + spriteMovement);

// On centre la vue sur le sprite

m\_pParentSprite->parentScene()->centerViewOn(m\_pParentSprite);

}

}

D’autre part, il peut être souhaitable que les ascenseurs, affichées automatiquement par GameView lorsque la scène est plus grande que la surface affichée, et qui permettent de se déplacer dans la scène, soient retirés. Cela se fait dans le constructeur de MainFrm :

//! Construit la fenêtre principale.

MainFrm::**MainFrm**(QWidget \***pParent**) : QWidget(*pParent*), ui(new Ui::MainFrm) {

ui->setupUi(this);

m\_pGameCanvas = new GameCanvas(*ui*->*grvGame*);

// Pour que la vue adapte sa taille à celle de la fenêtre

//ui->grvGame->setFitToScreenEnabled(true);

// Pour supprimer les marges de la zone de jeu

// ui->verticalLayout->setContentsMargins(QMargins(0,0,0,0));

// Désactive l'affichage des ascenseurs de la vue

ui->grvGame->setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);

ui->grvGame->setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);

}

### HUD

Attention : ma solution pose problème lorsque la scène est redimensionnée pour s’adapter à la taille de l’écran ou lorsque la scène est plus petite que la surface affichée par GameView !

## Changement de sens animé

## Listes d’animations

Nous avons vu au chapitre §13.1 deux façons assez simples pour adapter l’apparence du sprite à son sens de déplacement, tout en limitant la quantité d’images nécessaires.

Parfois, il est néanmoins nécessaire d’avoir une animation différente, spécifique à chaque sens de déplacement.

// Instancier et initialiser les sprites ici :

Sprite\* pSprite = new Sprite;

// Données spécifiques à la spritesheet

const int FRAME\_COUNT = 8; // nombre d'images de l'animation

const int COLUMN\_COUNT = 8; // nombre de colonnes

const int ROW\_COUNT = 8; // nombre de lignes

// Chargement de la spritesheet

QImage spriteSheet(GameFramework::imagesPath() + "girl\_walking\_spritesheet.png");

int frameWidth = spriteSheet.width() / COLUMN\_COUNT;

int frameHeight = spriteSheet.height() / ROW\_COUNT;

pSprite->clearAnimations();

// Découpage de la spritesheet et création des animations

for (int rowIndex = 0; rowIndex < ROW\_COUNT; rowIndex++) {

pSprite->addAnimation();

pSprite->setActiveAnimation(rowIndex);

for (int colIndex = 0; colIndex < COLUMN\_COUNT; colIndex++) {

QImage spriteImage = spriteSheet.copy(colIndex \* frameWidth,

rowIndex \* frameHeight,

frameWidth, frameHeight);

pSprite->addAnimationFrame(QPixmap::fromImage(spriteImage));

}

}

// Mise en place du sprite sur la scène

pSprite->setAnimationSpeed(100);

pSprite->setCurrentAnimationFrame(0);

pSprite->setPos(0,0);

m\_pScene->addSpriteToScene(*pSprite*);

pSprite->setTickHandler(new PlayerTickHandler*(*this*)*);

pSprite->registerForTick();

playertickhandler.cpp

enum **SpriteAnimation** {

WALK\_SOUTH = 0,

WALK\_SE,

WALK\_EAST,

WALK\_NE,

WALK\_NORTH,

WALK\_NW,

WALK\_WEST,

WALK\_SW

};

//! Une touche a été appuyée.

//! \param key Code de la touche appuyée.

void PlayerTickHandler::**onKeyPressed**(int key) {

switch (key) {

case Qt::Key\_Left: m\_playerDirection = QPointF(-1,0);

m\_pParentSprite->setActiveAnimation(WALK\_WEST);

break;

case Qt::Key\_Right: m\_playerDirection = QPointF(1,0);

m\_pParentSprite->setActiveAnimation(WALK\_EAST);

break;

case Qt::Key\_Up: m\_playerDirection = QPointF(0,-1);

m\_pParentSprite->setActiveAnimation(WALK\_NORTH);

break;

case Qt::Key\_Down: m\_playerDirection = QPointF(0,1);

m\_pParentSprite->setActiveAnimation(WALK\_SOUTH);

break;

}

if (!m\_pParentSprite->isAnimationRunning()) {

m\_pParentSprite->startAnimation();

}

}

## Gravité

## « Manger » du bitmap

Parfois il pourrait être utile de modifier le bitmap d’un sprite en y appliquant une opération de dessin.

## Dessiner dans un QWidget

## Utiliser le système de gestion des ressources de Qt

Par défaut, GameFramework charge les ressources dont il a besoin (images, icônes, sons) à partir de fichiers externes placés dans le répertoire res.

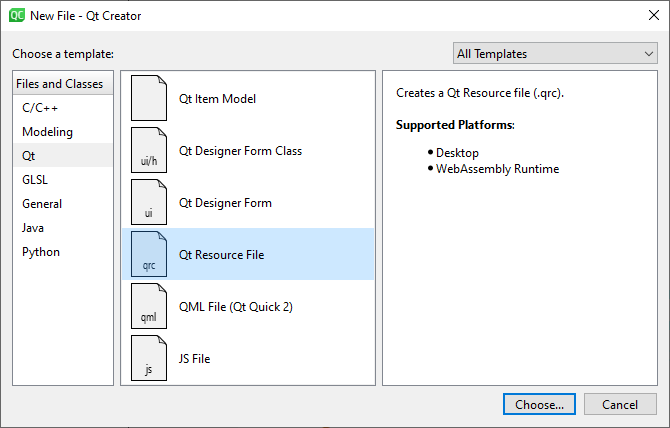
Si ce répertoire n’est pas présent, ou si son contenu est incomplet, certaines ressources peuvent être introuvables.

Pour remédier à ce problème, il est possible d’intégrer des ressources à l’exécutable généré par le compilateur, au prix d’un impact évident sur la taille de l’application.

Pour faire cela, il faut créer un fichier de ressources (extension .qrc) et le placer à l’emplacement des ressources. Dans le cas de GameFramework, dans le dossier res. Il faut également ajouter ce fichier de ressources au projet.

Une image contenant table

Description générée automatiquement



Une image contenant texte

Description générée automatiquement

Cela fait, il apparait dans la structure du projet.

Une image contenant texte

Description générée automatiquement

Il est ensuite possible d’ouvrir ce fichier dans l’éditeur avec le menu contextuel Open in editor… :

Pour ajouter des ressources, il faut créer un préfixe (qui peut se résumer à /) puis ajouter des fichiers. Il est cependant conseillé d’utiliser le système des préfixes, par exemple pour distinguer les images des sons.

Afin d’éviter des chemins à rallonge, il est également conseillé de donner un alias à chaque ressource.

Une image contenant texte

Description générée automatiquement

Le fichier .qrc obtenu est les suivant et peut être modifié à la main :

<RCC>

<qresource prefix="/images">

<file alias="cartoon-tree.png">images/demo/cartoon-tree.png</file>

<file>images/demo/landscape\_background.jpg</file>

</qresource>

<qresource prefix="/sounds">

<file>sounds/demo/coin.wav</file>

</qresource>

</RCC>

Le chemin pour accéder à une ressource sans alias : :/prefix/chemin\_complet/ressource. Par exemple :

m\_pScene->setBackgroundImage(QImage(":/images/images/demo/landscape\_background.jpg"));

Le chemin pour accéder à une ressource avec alias : :/prefix/alias. Par exemple :

Sprite\* **pRightTree** = new Sprite(":/images/cartoon-tree.png");

Parfois, il faut obtenir une URL (par exemple pour les sons). Le préfixe qrc est alors nécessaire : qrc:/prefix/chemin\_complet/ressource. Par exemple :

m\_bipSoundEffect.setSource(QUrl("qrc:/sounds/demo/coin.wav"));

L’utilisation du système des ressources pour certaines ressources n’empêche pas de charger les autres de façon classique, à partir du disque.

### Taille des ressources

Si la taille des ressources à intégrer dans l’exécutable devient trop important, le compilateur va générer une erreur.

Il faut alors changer la méthode de gestion des ressources en ajoutant la configuration suivante au projet (fichier .pro) :

CONFIG += resources\_big

## Quitter l’application

### Fermeture de la fenêtre par l’utilisateur

Lorsque l’utilisateur quitte l’application en fermant la fenêtre, cela arrête l’application et détruit l’objet MainFrm, ce qui provoque tour à tour la destruction de tous les éléments du jeu.

Le diagramme de séquence §14.2.2 illustre la succession d’appels provoquée par la fermeture de l’application.

### Terminer l’application par le programme

Pour provoquer de façon programmée la fermeture de l’application, il faut émettre le signal GameCanvas::requestToCloseApp().

# Annexes

## Diagramme de classes



## Diagrammes de séquence

### Démarrage du jeu



### Quitter l’application



## Propagation des événements du clavier

L’interface graphique donne le focus à l’instance de GameView. C’est donc elle qui reçoit les différents événements claviers.

En temps normal, GameView, qui hérite de QGraphicsView, laisse cette dernière gérer les événements, qui sont alors normalement transmis à la scène.

Le *GameFramework* installe un filtre à événement sur GameView, afin de prendre en charge les événements clavier dans GameCanvas.

Cela évite certains comportements standards des touches du clavier qui ne seraient pas adaptés à un jeu vidéo. Par exemple, que les flèches fassent scroller la vue.

Les autres événements (en particulier ceux de la souris) continuent d’être gérés par QGraphicsView, qui les transmet à QGraphicsScene, où ils sont convertis en QGraphicsSceneMouseEvent afin de les adapter au système de coordonnées de la scène.

Le *GameFramework* installe également un filtre à événement sur chaque instance de GameScene, afin de prendre en charge les événements souris dans GameCanvas.

## Ordre de destruction lorsque le programme quitte

## Déploiement

Pour qu’une application puisse s’exécuter, il ne suffit pas d’avoir le fichier exécutable. Plusieurs autres fichiers sont nécessaires pour son bon fonctionnement. Entre autres :

* Les bibliothèques utilisées par l’application (sous Windows, les fameuses dlls) ;
* Les ressources (images, sons) ;
* Les éventuels fichiers de configuration ;
* Les éventuels autres outils utilisés (base de données, drivers, etc.).

De plus, il faut s’assurer que notre application soit capable de localiser ces fichiers sur le disque.

Sur le poste de développement, tous ces outils sont à disposition, installés et configurés. Le système d’exploitation est configuré pour que l’accès à tous les fichiers nécessaires soit possible et l’IDE se charge également de démarrer l’application avec le bon répertoire de travail.

Malheureusement, si notre application est installée sur un poste de travail qui n’a pas été forcément prévu pour le développement, il est fort probable que tous les fichiers et ressources dont elle a besoin ne soient pas présents ou correctement configurés.

L’étape de déploiement consiste donc à s’assurer que lors de l’installation de notre application sur un poste quelconque, tous les outils, fichiers et ressources nécessaires au bon fonctionnement de l’application soient présents et accessibles.

### Emplacement des ressources

La structures des répertoires d’un projet de développement ressemble à cela :

├───build-brickbreaker-Desktop\_Qt\_6\_3\_0\_MinGW\_64\_bit-Debug

│ ├───debug

│ │ brickbreaker.exe

│ └───release

├───build-brickbreaker-Desktop\_Qt\_6\_3\_0\_MinGW\_64\_bit-Release

│ ├───debug

│ └───release

│ brickbreaker.exe

├───doc

├───res

│ ├───images

│ └───sounds

├───scripts

└───src

Lorsque l’application est compilée et démarrée depuis Qt Creator, le fichier exécutable est placé dans le sous-dossier debug (si compilée en mode *debug*) ou release (si compilée en mode *release*). Le chemin de travail de l’application est donc celui-ci[[3]](#footnote-3).

Les ressources sont donc situées, relativement au chemin de l’application, à l’emplacement suivant : ..\..\res.

Le problème est que lorsque l’application est installée sur un poste final, le fichier exécutable est normalement situé directement dans le dossier de l’application, au même niveau que les ressources :

├───brickbreaker.exe

│

├───res

│ ├───images

│ └───sounds

Les ressources sont donc situées, relativement au chemin de l’application, à l’emplacement suivant : .\res.

Cela implique donc que dans notre programme, selon qu’il est démarré depuis Qt Creator ou depuis une installation, l’emplacement des ressources change.

1. Il peut être changé et déplacé. [↑](#footnote-ref-1)
2. On se limite ici au cas simple où la balle ne touche que des surfaces orthonormées, donc verticales ou horizontales, mais pas inclinées. [↑](#footnote-ref-2)
3. Il peut être obtenu avec QCoreApplication::applicationDirPath(). [↑](#footnote-ref-3)