**GameFrameWork**

**Tutoriel**

# 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.

## 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 au moyen de Qt Creator 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.



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\* pSprite = new Sprite(GameFramework::imagesPath() + "player\_m1.png");

m\_pScene->addSpriteToScene(pSprite);

// ...

m\_pGameCanvas->startTick();

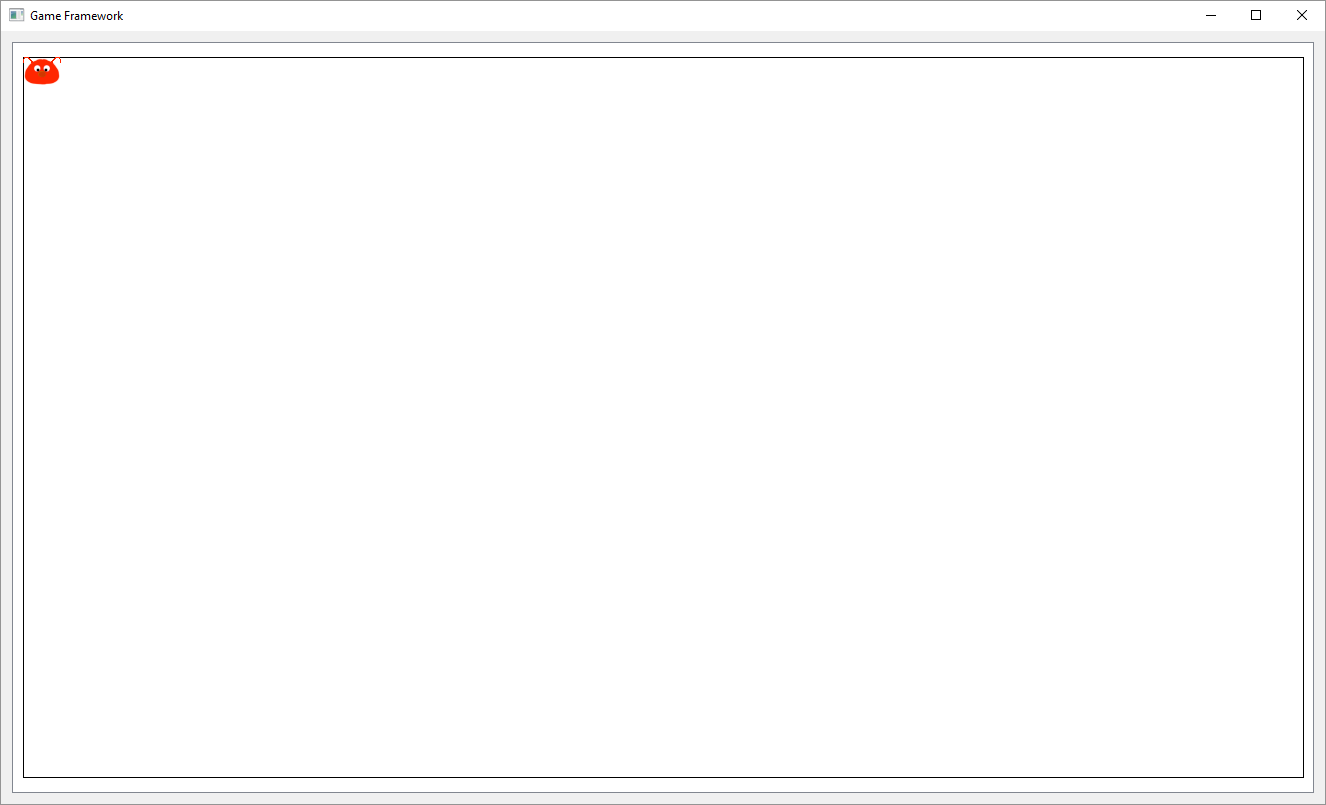
}

**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), comme expliqué au §7.

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

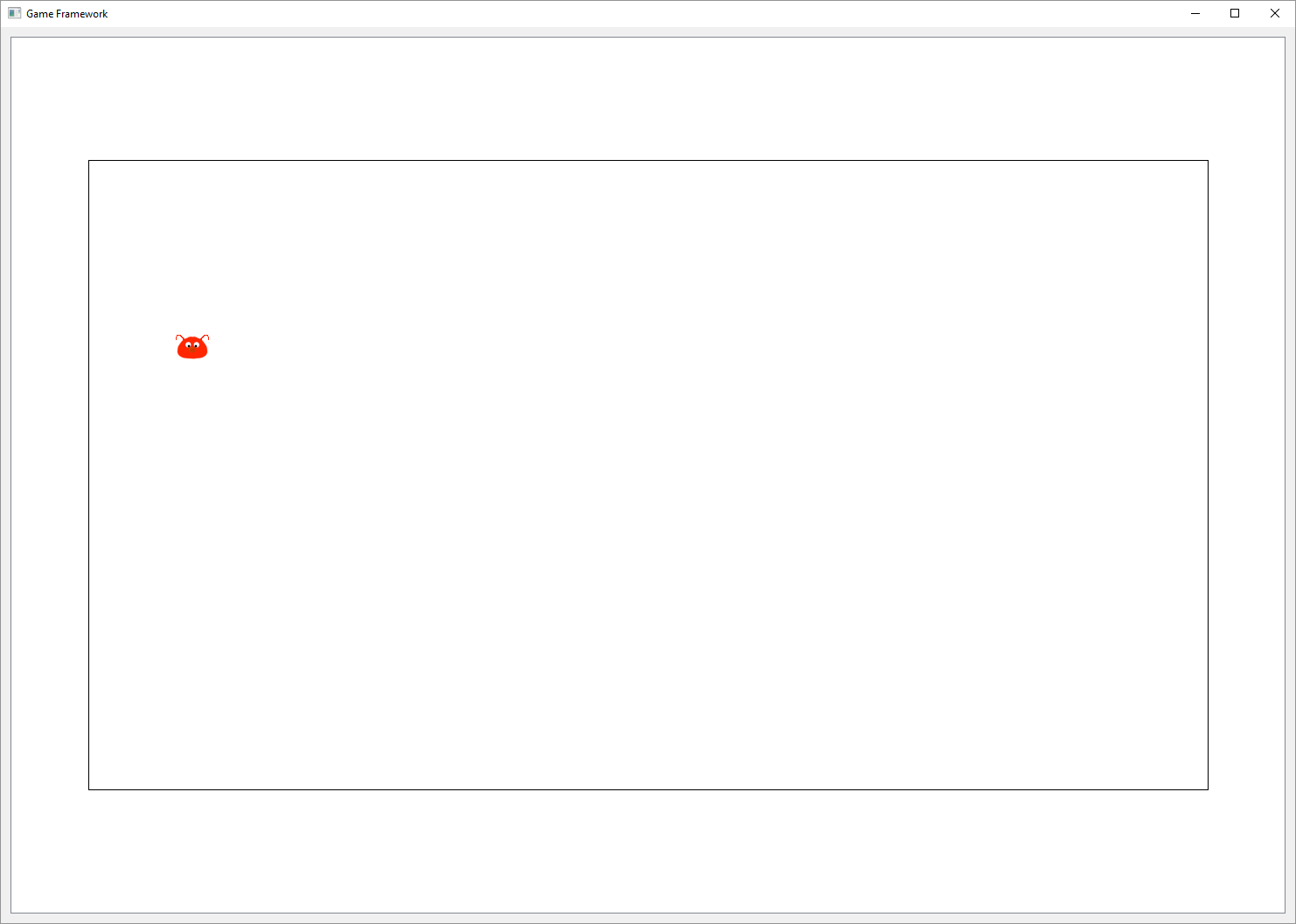
// Instancier et initialiser les sprites ici :

// ...

Sprite\* pSprite = new Sprite(GameFramework::imagesPath() + "player\_m1.png");

m\_pScene->addSpriteToScene(pSprite);

pSprite->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()) :

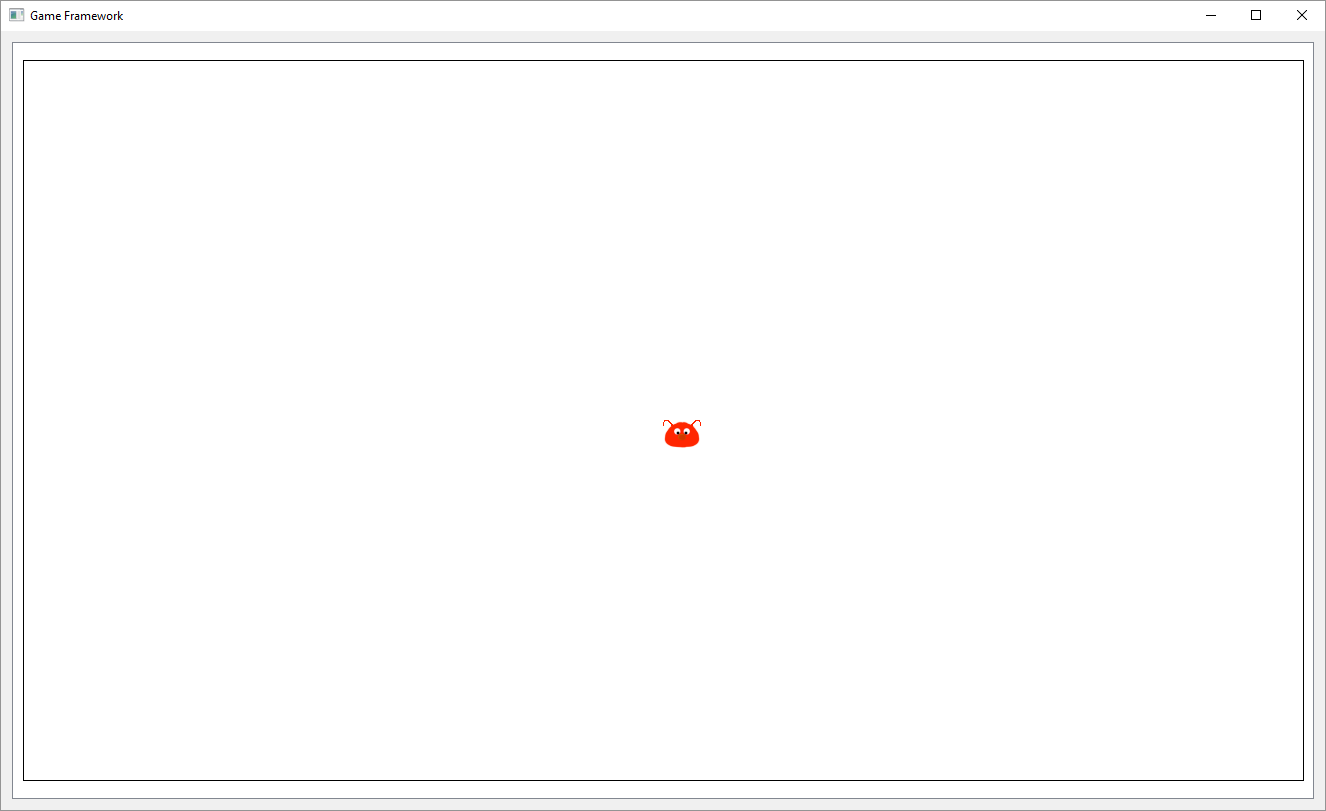
// Instancier et initialiser les sprites ici :

// ...

Sprite\* pSprite = new Sprite(GameFramework::imagesPath() + "player\_m1.png");

m\_pScene->addSpriteToScene(pSprite);

pSprite->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 sprite ici :

Sprite\* pSprite = new Sprite(GameFramework::imagesPath() + "tutorial/player\_m1.png");

m\_pScene->addSpriteToScene(pSprite);

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

pSprite->addAnimationFrame(GameFramework::imagesPath() + "player\_m2.png");

pSprite->startAnimation(500);

Dans l’exemple ci-dessus, le sprite affichera tour à tour la première image (player\_m1.png) puis la deuxième (player\_m2.png), toutes les 500 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\_pPlayer** = 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 sprite ici :

Sprite\* **pSprite** = new Sprite(GameFramework::imagesPath() + "tutorial/player\_m1.png");

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

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

pSprite->addAnimationFrame(GameFramework::imagesPath() + "tutorial/player\_m2.png");

pSprite->startAnimation(500);

m\_pPlayer = pSprite;

## Déplacement simple

Pour déplacer un sprite, il suffit de **changer** sa position X ou Y au moyen des méthodes Sprite::setPos(), Sprite::setX() et Sprite::setY().

Le déplacement peut, par exemple, être lié à l'appui d'une touche.

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

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

//! 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\_pPlayer->setX(m\_pPlayer->x() - 20); break;

case Qt::Key\_Right:

m\_pPlayer->setX(m\_pPlayer->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 de la **fréquence de répétition** des touches.

Une meilleure solution consiste à utiliser le mécanisme de **cadence du jeu** (tick).

## Déplacement en cadence (tick)

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. 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, en se basant sur la vitesse de déplacement du sprite (, en pixels par seconde) puis calculer par la formule ci-dessous la distance parcourue (, en pixels) pendant la durée du tick (, en secondes) :

const int PLAYER\_SPEED = 150 ; // vitesse de déplacement du joueur 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 ne tient donc **plus compte** des touches pressées au clavier, mais se contente de déplacer le sprite sur la droite, à une vitesse constante. Il convient donc de supprimer le code précédemment écrit dans la méthode keyPressed() :

//! 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\_pPlayer->setX(m\_pPlayer->x() - 20); break;~~

~~case Qt::Key\_Right:~~

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

~~}~~

}

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

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

**gamecore.h :**

private:

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

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

Sprite\* **m\_pPlayer** = nullptr;

int **m\_playerDirection** = 1;

**gamecore.cpp :**

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

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

double distance = PLAYER\_SPEED \*

elapsedTimeInMilliseconds / 1000.0 \*

m\_playerDirection;

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

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

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

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

m\_playerDirection \*= -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 par un *tick handler*

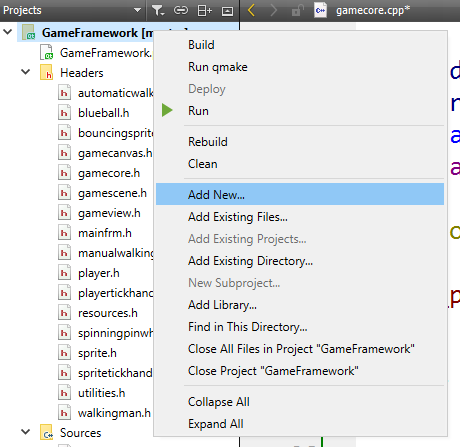
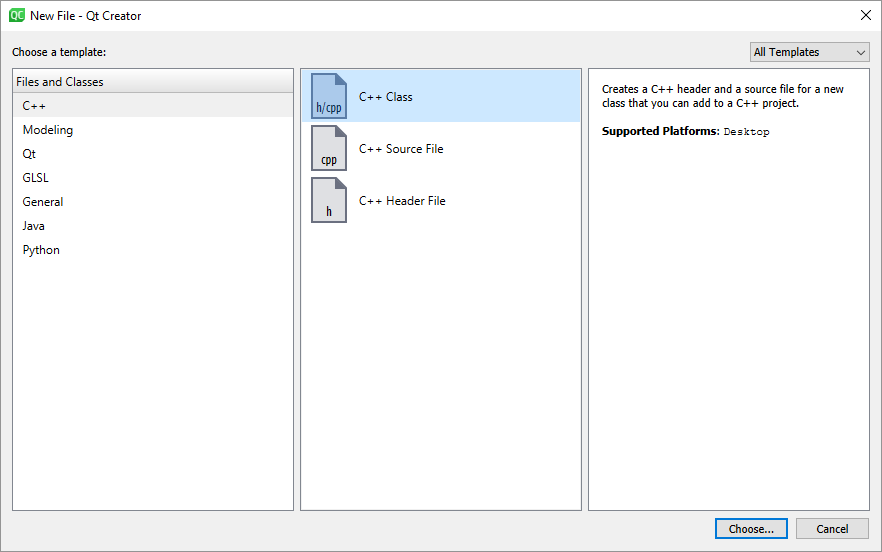
Une autre façon de gérer le déplacement d’un sprite, plus proche d’une philosophie orientée-objet, est de placer cette gestion dans une classe dédiée, appelée ici un *tick handler*, ou *gestionnaire de tick*.

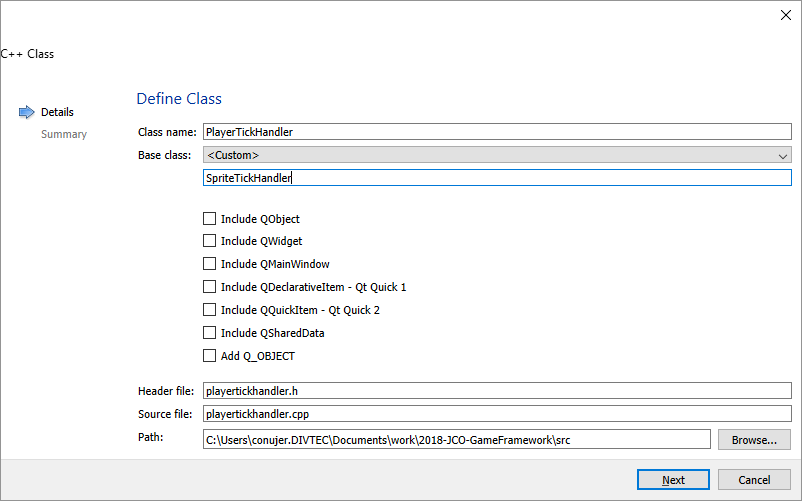
Le GameFramework propose une classe de base, SpriteTickHandler, qui peut être spécialisée pour gérer un déplacement selon une **stratégie** particulière, puis **affectée** à un sprite pour que celui-ci se déplace selon cette stratégie.

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.

Il s’agit donc de créer une nouvelle classe, par exemple PlayerTickHandler, qui hérite de SpriteTickHandler et d’y placer la code qui prendra en charge le déplacement d’un sprite.



**playertickhandler.h :**

#ifndef PLAYERTICKHANDLER\_H

#define PLAYERTICKHANDLER\_H

#include "spritetickhandler.h"

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

//!

//! Cette classe gère le déplacement d'un sprite de façon automatique, à une vitesse

//! prédéterminée, afin de faire des aller retour jusqu'aux bordures de la scène.

//!

class **PlayerTickHandler** : public SpriteTickHandler

{

public:

**PlayerTickHandler**(Sprite\* pParentSprite = nullptr);

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

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

private:

int m\_playerDirection = 1;

};

#endif // PLAYERTICKHANDLER\_H

**playertickhandler.cpp :**

#include "playertickhandler.h"

#include "sprite.h"

#include "gamescene.h"

const int **PLAYER\_SPEED** = 200;

//! Construit ce tick handler.

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

}

//! Initialise ce tick hanlder.

void PlayerTickHandler::***init***() {

}

//! Déplace le sprite parent.

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

double **distance** = PLAYER\_SPEED \*

elapsedTimeInMilliseconds / 1000.0 \*

m\_playerDirection;

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

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

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

m\_pParentSprite->left() < 0.0) {

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

}

}

Une fois le gestionnaire codé, il suffit de l’instancier (pour ce faire, ne pas oublier l’inclure l’entête playertickhandler.h dans le fichier gamecore.cpp) et d’affecter cette instance au sprite qui devra être automatiquement déplacé, par exemple dans le constructeur de GameCore :

#include "playertickhandler.h"

…

// Instancier et initialiser les sprite ici :

Sprite\* **pSprite** = new Sprite(GameFramework::imagesPath() + "tutorial/player\_m1.png");

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

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

pSprite->addAnimationFrame(GameFramework::imagesPath() + "tutorial/player\_m2.png");

pSprite->startAnimation(500);

pSprite->setTickHandler(new PlayerTickHandler);

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().

Maintenant que le déplacement du sprite est pris en charge par un tickhandler, il n’est plus nécessaire que GameCore prenne également cela en charge.

//! Cadence.

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

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

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

~~double~~ **~~distance~~** ~~= PLAYER\_SPEED \*~~

~~elapsedTimeInMilliseconds / 1000.0 \*~~

~~m\_playerDirection;~~

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

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

~~if (m\_pPlayer->right() > m\_pScene->width() ||~~

~~m\_pPlayer->left() < 0) {~~

~~m\_playerDirection \*= -1; // Change de direction~~

~~}~~

}

Il faut également supprimer de GameCore la constante PLAYER\_SPEED et les variables membre m\_playerDirection et m\_pPlayer. Du coup, la déclaration de la classe Sprite dans gamecore.h peut également être retirée.

class **GameCanvas**;

class **GameScene**;

~~class~~ **~~Sprite~~**~~;~~

…

class **GameCore** : public QObject

{

…

private:

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

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

~~Sprite\*~~ **~~m\_pPlayer~~** ~~= nullptr;~~

~~int~~ **~~m\_playerDirection~~** ~~= 1;~~

…

};

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.

### Intégrer les touches du clavier à un tickhandler

Une image contenant intérieur, sombre

Description générée automatiquement

Pour qu’un gestionnaire de tick puisse gérer le déplacement d’un sprite en tenant compte des touches du clavier, il faut qu’il puisse se connecter aux signaux émis par la classe GameCore (les signaux 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.

La première étape, pour que notre gestionnaire de tick puisse se connecter à un signal et de mettre en place ce mécanisme en faisant hériter notre gestionnaire de la classe QObject (ce qui nécessite d’inclure QObject) et en activant les signaux au moyen de la macro Q\_OBJECT. 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. Comme ces signaux sont émis par la classe GameCore, il faut que PlayerTickHandler ait connaissance de cette classe, afin de pouvoir s’y connecter. L’instance de GameCore sera donc passée en paramètre au constructeur. Cela augmente malheureusement le couplage de PlayerTickHandler, ce qu’il faudrait éviter au maximum, mais dans cette situation, nous n’avons pas d’autre choix.

À noter qu’il n’est pas nécessaire, et même pas recommandé, d’inclure toute la classe GameCore dans le fichier d’entête de PlayerTickHandler. Une simple déclaration de l’existence de la classe GameCore suffit.

**playertickhandler.h :**

#ifndef PLAYERTICKHANDLER\_H

#define PLAYERTICKHANDLER\_H

#include <QObject>

#include <QPointF>

#include "spritetickhandler.h"

class **GameCore**;

class **PlayerTickHandler** : public QObject, public SpriteTickHandler

{

Q\_OBJECT

public:

**PlayerTickHandler**(GameCore\* pGameCore, Sprite\* pParentSprite = nullptr);

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

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

private:

int m\_playerDirection = 0; // pas de mouvement tant qu'une touche n'est pas appuyée

private slots:

void **onKeyPressed**(int key);

void **onKeyReleased**(int key);

};

#endif // PLAYERTICKHANDLER\_H

**Attention** : la macro Q\_OBJECT nécessite que le compilateur génère un fichier temporaire supplémentaire, appelé fichier moc. Pour que le projet en tienne compte, il faut exécuter manuellement la commande qmake, 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'

Dans le constructeur de la classe PlayerTickHandler, il faut ensuite connecter les signaux émis par GameCore aux slots :

//! Constructeur

//! \param pGameCore Pointeur sur le GameCore qui signal l'appui sur les touches du clavier.

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

PlayerTickHandler::**PlayerTickHandler**(GameCore\* pGameCore, Sprite\* pParentSprite) : SpriteTickHandler(*pParentSprite*) {

connect(pGameCore, &GameCore::notifyKeyPressed,

this, &PlayerTickHandler::onKeyPressed);

connect(pGameCore, &GameCore::notifyKeyReleased,

this, &PlayerTickHandler::onKeyReleased);

}

Pour que ce code fonctionne, il faut inclure le fichier d’entête de la classe GameCore (gamecore.h) dans le fichier .cpp de la classe PlayerTickHander (playertickhandler.cpp) :

#include "gamecore.h"

Une implémentation très simple des slots qui prennent en charge les touches pourrait être la suivante :

//! 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 = -1; break;

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

}

}

//! Une touche a été relâchée.

//! \param key Code de la touche relâchée.

void PlayerTickHandler::**onKeyReleased**(int key) {

switch (key) {

case Qt::Key\_Left: m\_playerDirection = 0; break;

case Qt::Key\_Right: m\_playerDirection = 0; break;

}

}

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, pour que le code fonctionne, il faut encore adapter l’instanciation de cette classe dans le constructeur de GameCore, car la signature du constructeur de la classe PlayerTickHandler a changé :

// Instancier et initialiser les sprites ici :

Sprite\* pSprite = new Sprite(GameFramework::imagesPath() + "player\_m1.png");

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

pSprite->setTickHandler(new PlayerTickHandler*(*this*)*);

pSprite->registerForTick();

### Utiliser un vecteur de déplacement

Jusqu’ici, le déplacement du sprite était limité à l’axe horizontal.

Lorsqu’un sprite est amené à se déplacer dans 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.

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

QPointF m\_playerDirection = QPointF(0,0); // Le sprite ne bouge pas tant qu'une

// touche n'est pas appuyée

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.

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

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

QPointF spriteMovement = m\_playerDirection \* PLAYER\_SPEED \* elapsedTimeInMilliseconds / 1000.0;

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

}

Finalement, il faut encore modifier les deux slots qui réagissent aux touches du clavier afin de permettre un déplacement dans toutes les directions.

//! 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); break;

case Qt::Key\_Right: m\_playerDirection = QPointF(1,0); break;

case Qt::Key\_Up: m\_playerDirection = QPointF(0,-1); break;

case Qt::Key\_Down: m\_playerDirection = QPointF(0,1); break;

}

}

//! Une touche a été relâchée.

//! \param key Code de la touche relâchée.

void PlayerTickHandler::**onKeyReleased**(int key) {

switch (key) {

case Qt::Key\_Left:

case Qt::Key\_Right:

case Qt::Key\_Up:

case Qt::Key\_Down: m\_playerDirection = QPointF(0,0); break;

}

}

# Détecter les collisions

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

Il y a deux façons de gérer les collisions :

* **Déplacer** le sprite, **puis** **vérifier** s’il collisionne avec un autre sprite ;
* Déterminer son prochain emplacement, et **vérifier**, **avant de la déplacer**, si ce prochain emplacement entre en collision avec un autre sprite et alors, par exemple, renoncer à déplacer le sprite.

Pour illustrer cela, ajoutons deux éléments dans la scène : un mur et une sphère bleue.

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

// Ajout de trois obstacles

Sprite\* pWall = new Sprite(GameFramework::imagesPath() + "brique.jpg");

pWall->setScale(0.3);

m\_pScene->addSpriteToScene(*pWall*, 70, 50);

Sprite\* pGround = new Sprite(GameFramework::imagesPath() + "grass.png");

pGround->setScale(0.3);

pGround->setZValue(-1); // passe derrière les autres sprites

m\_pScene->addSpriteToScene(*pGround*, 0, 80);

Sprite\* pSphere = new Sprite(GameFramework::imagesPath() + "ball.png");

m\_pScene->addSpriteToScene(*pSphere*, 250, 50);

## Déplacer puis vérifier

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.

Depuis un SpriteTickHandler, la scène est accessible via le sprite : m\_pParentSprite->parentScene().

La méthode tick() de la classe PlayerTickHandler pourrait être modifiée comme suit :

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

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

QPointF spriteMovement = m\_playerDirection \* PLAYER\_SPEED \* elapsedTimeInMilliseconds / 1000.0;

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

// Détecte si le sprite est en collision

auto listeCollision = m\_pParentSprite->parentScene()->collidingSprites(m\_pParentSprite);

// S'il y a collision : le sprite est remis reculé à sa place de départ

if (!listeCollision.isEmpty())

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

}

À noter qu’avec ce code, on ne détecte plus si le sprite sort des limites de la scène. Cette fonctionnalité sera rétablie un peu plus loin.

## Vérifier avant de déplacer

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.

Depuis un SpriteTickHandler, la scène est accessible via le sprite : m\_pParentSprite->parentScene().

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.

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);

// Récupère tous les sprites de la scène que toucherait ce sprite à sa prochaine position

auto collidingSprites = m\_pParentSprite->parentScene()->collidingSprites(nextSpriteRect);

// Supprimer le sprite lui-même, qui collisionne toujours avec son bounding rect

collidingSprites.removeAll(m\_pParentSprite);

// S'il n'y a pas de collision et que le sprite ne sort pas de la scène, on le déplace

// (en lui appliquant le vecteur de déplacement)

if (m\_pParentSprite->parentScene()->isInsideScene(nextSpriteRect) &&

collidingSprites.isEmpty()) {

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

}

}

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().

## Distinguer différents types de collision

Lorsqu’il y a une collision, il sera utile de pouvoir dire si le sprite en entré en collision avec un obstacle qui se contente de le bloquer ou si cet obstacle doit provoquer la destruction du sprite.

Pour faire cette distinction, il y a deux méthodes : ajouter une donnée distinctive à l’obstacle, permettant de l’identifier, ou se baser sur le type exact de l’obstacle au sein d’une hiérarchie de sprites.

### 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 pWall->setData(0, "mur") mémorise pour la clé 0 la chaîne de caractères "mur".

Il est ensuite possible de lire cette données grâce à la méthode data() : pWall->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 {

WALL,

BOMB,

BACKGROUND

};

Ainsi, le code ci-dessus deviendrait :

pWall->setData(SPRITE\_TYPE\_KEY, WALL);

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

Toutefois, ces types énumérés doivent être accessibles aussi bien dans GameCore (c’est là que les sprites sont créés) et dans notre PlayerTickHandler et potentiellement dans d’autres classes également.

Nous avons donc les possiblités suivantes : déclarer ces types énumérés dans GameCore, et inclure GameCore dans PlayerTickHandler et les autres classes qui en auraient besoin. Cela provoque un fort couplage de GameCore mais qui pourrait être acceptable étant donné que GameCore est un élément central du jeu.

Une autre possibilité est de déclarer ces types énumérés dans notre classe PlayerTickHandler, mais ce ne serait pas judicieux puisque ces types énumérés concernent la globalité du jeu plutôt qu’un seul gestionnaire de tick, qui devrait alors être inclus partout où ces énumérations sont nécessaires.

Une troisième possibilité, et c’est celle que nous allons implémenter, sera de créer un fichier d’entête spécifique, contenant les déclarations qui concernent la globalité du jeu et il suffira aux classes qui ont besoin de ces déclarations, de les inclure.

Attention toutefois à ne pas abuser de cette solution et d’en faire un espace de déclaration de variables globales, il faut absolument se limiter au strict nécessaire.

Il faut donc créer un fichier d’entête nommé par exemple **gametypes.h** et d’y écrire le code suivant :

**gametypes.h :**

#ifndef GAMETYPES\_H

#define GAMETYPES\_H

namespace **GameFramework** {

// Enuméré pour l'identification des clés de données d'un sprite

enum **SpriteDataKey** {

SPRITE\_TYPE\_KEY = 0

};

// Enumération des différents types de sprites

enum **SpriteType** {

WALL,

BOMB

};

}

#endif // GAMETYPES\_H

Il faut ensuite inclure ce fichier dans gamecore.cpp et modifier le code de création des sprites ainsi :

#include "gametypes.h"

…

// Ajout de trois obstacles

Sprite\* pWall = new Sprite(GameFramework::imagesPath() + "brique.jpg");

pWall->setData(GameFramework::SPRITE\_TYPE\_KEY, GameFramework::WALL);

pWall->setScale(0.3);

m\_pScene->addSpriteToScene(*pWall*, 70, 50);

Sprite\* pGround = new Sprite(GameFramework::imagesPath() + "grass.png");

pGround->setData(GameFramework::SPRITE\_TYPE\_KEY, GameFramework::BACKGROUND);

pGround->setScale(0.3);

pGround->setZValue(-1); // passe derrière les autres sprites

m\_pScene->addSpriteToScene(*pGround*, 0, 80);

Sprite\* pSphere = new Sprite(GameFramework::imagesPath() + "ball.png");

pSphere->setData(GameFramework::SPRITE\_TYPE\_KEY, GameFramework::BOMB);

m\_pScene->addSpriteToScene(*pSphere*, 250, 50);

Ensuite, dans PlayerTickHandler, on peut, lors d’une collision, vérifier le type du sprite collisionné et agir en conséquence.

Pour cela, il faut commencer par inclure le fichier gametypes.h dans playertickhandler.cpp.

#include "gametypes.h"

Puis la méthode PlayerTickHandler::tick() peut être modifiée comme suit :

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);

// Récupère tous les sprites de la scène que toucherait ce sprite à sa prochaine position

auto collidingSprites = m\_pParentSprite->parentScene()->collidingSprites(nextSpriteRect);

// Supprimer le sprite lui-même, qui collisionne toujours avec son bounding rect

collidingSprites.removeAll(m\_pParentSprite);

bool moveAllowed = true;

// S'il y a des collisions, on vérifie de quelle nature pour décider

// si le déplacement peut avoir lieu ou pas

if (!collidingSprites.isEmpty()) {

// Même si le sprite collisionne avec plusieurs sprites, on ne gère

// que le premier

Sprite\* pFirstCollidingSprite = collidingSprites[0];

int spriteData = pFirstCollidingSprite->data(GameFramework::SPRITE\_TYPE\_KEY).toInt();

GameFramework::SpriteType spriteType = static\_cast<GameFramework::SpriteType>

(spriteData);

switch (spriteType) {

case GameFramework::WALL:

// déplacement interdit

case GameFramework::BOMB:

// le sprite devrait être détruit, pour l'instant, déplacement interdit

moveAllowed = false;

break;

case GameFramework::BACKGROUND:

moveAllowed = true;

break;

}

}

if (moveAllowed && m\_pParentSprite->parentScene()->isInsideScene(nextSpriteRect)) {

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

}

}

Bien sûr, le code pourrait être un peu raccourci car les variables pFirstCollidingSprite, spriteData et spriteType ne servent à rien, mais cela implique un critère de switch un peu long :

// Même si le sprite collisionne avec plusieurs sprites, on ne gère

// que le premier

switch (static\_cast<GameFramework::SpriteType>(collidingSprites[0]

->data(GameFramework::SPRITE\_TYPE\_KEY)

.toInt())) {

case GameFramework::WALL:

// déplacement interdit

case GameFramework::BOMB:

// le sprite devrait être détruit, pour l'instant, déplacement interdit

moveAllowed = false;

break;

case GameFramework::BACKGROUND:

moveAllowed = true;

break;

}

Cette solution a l’avantage de permettre de distinguer plusieurs sprites sans avoir à spécialiser la classe Sprite, qui fait l’objet de l’autre solution, décrite au paragraphe suivant.

### Identifier un sprite par son type exact

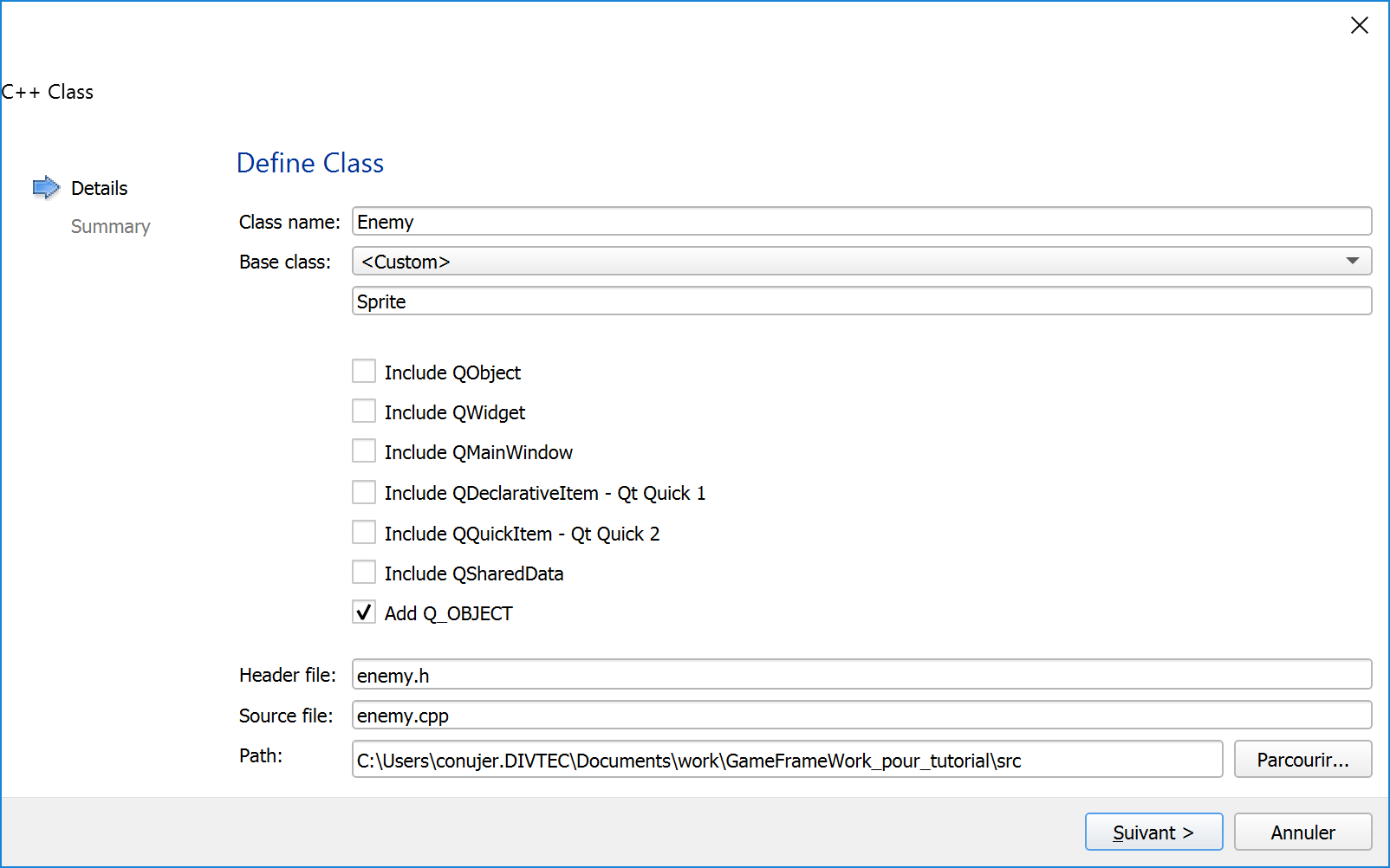
Cette solution repose sur la spécialisation de la classe Sprite. Elle est donc particulièrement utile pour les projets qui tirent parti de l’héritage d’un Sprite.

Imaginons que nous voulions faire la hiérarchie de classes suivante, afin de distinguer un ennemi (qui détruit le joueur), d’un mur (qui bloque le joueur) et d’un élément du décor (qui ne bloque pas le joueur).



Il faut donc ajouter à notre projet les nouvelles classes Enemy, Wall et BackgroundItem, qui héritent de Sprite. Pour simplifier les constructeurs, ces classes n’en proposeront qu’un, qui prend en paramètre un QPixmap et un QGraphicsItem parent, à nullptr par défaut.

Il est important de cocher Add Q\_OBJECT au moment de la création de la classe afin de pouvoir ultérieurement vérifier le type exact d’un sprite.



**enemy.h :**

#ifndef ENEMY\_H

#define ENEMY\_H

#include "sprite.h"

class **Enemy** : public Sprite

{

Q\_OBJECT

public:

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

};

#endif // ENEMY\_H

**enemy.cpp :**

#include "enemy.h"

Enemy::**Enemy**(const QPixmap& rPixmap,

QGraphicsItem\* pParent) : Sprite(rPixmap, *pParent*)

{

}

Le code des classes Wall et BackgroundItem est exactement le même, seul le nom de la classe et du constructeur changent.

Il faut maintenant modifier le constructeur de GameCore, pour utiliser ces nouvelles classes lors de la création des obstacles. Cela implique également d’inclure dans gamecore.cpp les trois fichiers d’entête concernés.

#include "enemy.h"

#include "backgrounditem.h"

#include "wall.h"

…

// Ajout de trois obstacles

Sprite\* pWall = new Wall(GameFramework::imagesPath() + "brique.jpg");

pWall->setScale(0.3);

m\_pScene->addSpriteToScene(*pWall*, 70, 50);

Sprite\* pGround = new BackgroundItem(GameFramework::imagesPath() + "grass.png");

pGround->setScale(0.3);

pGround->setZValue(-1); // passe derrière les autres sprites

m\_pScene->addSpriteToScene(*pGround*, 0, 80);

Sprite\* pSphere = new Enemy(GameFramework::imagesPath() + "ball.png");

m\_pScene->addSpriteToScene(*pSphere*, 250, 50);

Ensuite, dans PlayerTickHandler, on peut, lors d’une collision, vérifier le type du sprite collisionné et agir en conséquence.

Le principe de fonctionnement est le suivant : lorsqu’une classe hérite de QObject (c’est le cas de nos classes Enemy, Wall et BackgroundItem, qui héritent de QObject au travers de la classe Sprite) et intègre la macro Q\_OBJECT, son type exact peut être vérifié avec la commande qobject\_cast<>().

Par exemple, si pSpriteEnCollision est un pointeur sur Sprite\*, on peut essayer de le convertir en un pointeur sur Wall\* ainsi :

Wall\* pWall = qobject\_cast<Wall\*>(pSpriteEnCollision).

Si pSpriteEnCollision est effectivement une instance de Wall, alors le pointeur pWall sera valide et pourra être utilisé. À l’inverse, si le type exact du sprite pSpriteEnCollision n’est pas Wall, alors qobject\_cast retourne nullptr.[[2]](#footnote-2)

Nous pouvons donc utiliser ce mécanisme dans PlayerTickHandler pour savoir, lors d’une collision, avec quel type de sprite le joueur est entré en collision.

Pour cela, il faut commencer par inclure les classes Enemy, Wall et BackgroundItem dans playertickhandler.cpp.

#include "enemy.h"

#include "wall.h"

#include "backgrounditem.h"

Ceci fait, la méthode PlayerTickHandler::tick() peut être modifiée comme suit :

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);

// Récupère tous les sprites de la scène que toucherait ce sprite à sa prochaine position

auto collidingSprites = m\_pParentSprite->parentScene()->collidingSprites(nextSpriteRect);

// Supprimer le sprite lui-même, qui collisionne toujours avec son bounding rect

collidingSprites.removeAll(m\_pParentSprite);

bool moveAllowed = true;

// S'il y a des collisions, on vérifie de quelle nature pour décider

// si le déplacement peut avoir lieu ou pas

if (!collidingSprites.isEmpty()) {

// Même si le sprite collisionne avec plusieurs sprites, on ne gère

// que le premier

Sprite\* pFirstCollision = collidingSprites[0];

if (qobject\_cast<Enemy\*>(*pFirstCollision*) || qobject\_cast<Wall\*>(*pFirstCollision*)) {

// C'est un ennemi ou un mur, pour l'instant, déplacement interdit

moveAllowed = false;

} else if (qobject\_cast<BackgroundItem\*>(*pFirstCollision*)) {

// C'est un élément du décor, on pourrait mettre du code spécifique ici

}

}

if (moveAllowed && m\_pParentSprite->parentScene()->isInsideScene(nextSpriteRect)) {

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

}

}

## 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 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**

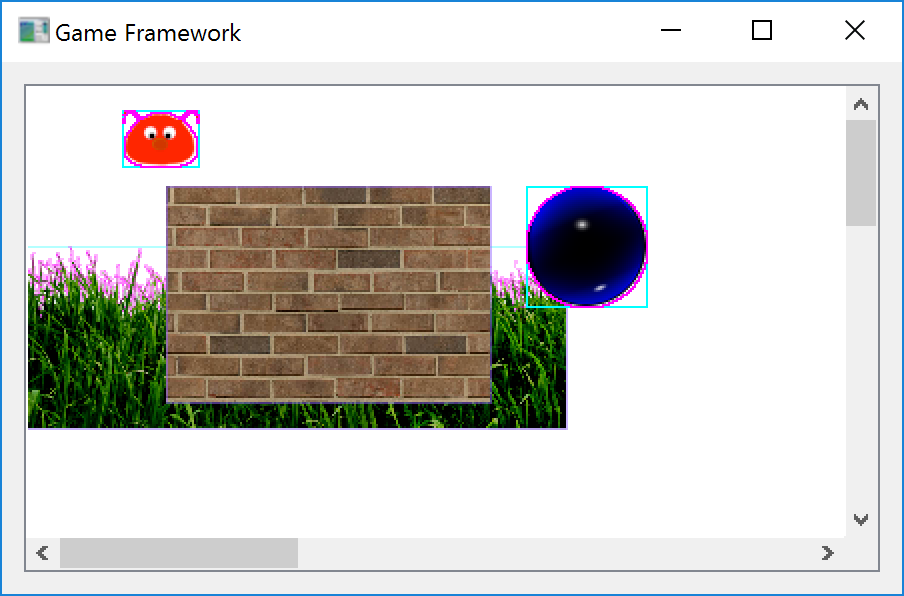
Ceci fait, chaque sprite dessiné est représenté avec son *bounding rect* en cyan et sa *shape* en magenta.

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

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.

Ce n’est pas toujours possible : parfois, c’est au sein d’un *SpriteTickHandler* que l’on détecte une collision qui nécessite de supprimer le sprite. Or, si le *SpriteTickHandler* supprime le sprite qu’il gère, cela provoque la destruction du *SpriteTickHandler* lui-même, qui est justement en train de s’exécuter, ce qui provoque un crash.

Dans ce cas, il faut utiliser une astuce proposée par *Qt* : indiquer qu’un objet doit être effacé plus tard, lors du prochain événement système. La méthode Sprite::deleteLater() (ou plus généralement QObject::deleteLater()) permet de faire cela.

Reprenons notre PlayerTickHandler afin que le sprite se détruise s’il touche un ennemi. La méthode PlayerTickHandler::tick() peut être modifiée comme suit :

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);

// Récupère tous les sprites de la scène que toucherait ce sprite à sa prochaine position

auto collidingSprites = m\_pParentSprite->parentScene()->collidingSprites(nextSpriteRect);

// Supprimer le sprite lui-même, qui collisionne toujours avec son bounding rect

collidingSprites.removeAll(m\_pParentSprite);

bool moveAllowed = true;

// S'il y a des collisions, on vérifie de quelle nature pour décider

// si le déplacement peut avoir lieu ou pas

if (!collidingSprites.isEmpty()) {

// Même si le sprite collisionne avec plusieurs sprites, on ne gère

// que le premier

Sprite\* pFirstCollision = collidingSprites[0];

if (qobject\_cast<Enemy\*>(*pFirstCollision*)) {

// C'est un ennemi, destruction

m\_pParentSprite->deleteLater();

} else if (qobject\_cast<Wall\*>(*pFirstCollision*)) {

// C'est un mur, déplacement interdit

moveAllowed = false;

} else if (qobject\_cast<BackgroundItem\*>(*pFirstCollision*)) {

// C'est un élément du décor, on pourrait mettre du code spécifique ici

}

}

if (moveAllowed && m\_pParentSprite->parentScene()->isInsideScene(nextSpriteRect)) {

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

}

}

Ainsi, dès que le joueur touche l’ennemi (la sphère bleue dans notre exemple), il est détruit et disparaît instantanément de la scène.

Il est souvent désiré que lorsqu’un sprite est détruit, il ne disparaisse pas instantanément mais explose ou s’évapore, ou laisse place à un cadavre. Pour pouvoir faire cela, il faut que GameCore sache qu’un sprite a été détruit.

## Savoir qu’un sprite va être détruit

Un sprite qui est sur le point d’être détruit émet le signal spriteDestroyed.

Il est possible de connecter un slot à ce signal, afin d’en être informé :

connect(pSprite, &Sprite::spriteDestroyed, this, &GameCore::onSpriteDestroyed);

Dans l’exemple ci-dessus, pSprite est le sprite pour lequel on souhaite être informé de sa destruction et onSpriteDestroyed est le nom du slot qui devra être appelé lorsque le signal spriteDestroyed sera émis.

La signature du slot (dans l’exemple ci-dessus : onSpriteDestroyed) doit être la suivante :

void **onSpriteDestroyed**(Sprite\* pSprite)

Le slot reçoit en paramètre le sprite qui est sur le point d’être détruit.

**Attention** : Il faut être extrêmement prudent lorsqu’on se connecte au signal spriteDestroyed. En effet, il ne faut pas oublier que le sprite peut être détruit pour au moins deux raisons :

* Destruction **prévue** et volontaire, à la suite d’un événement du jeu
* Destruction **automatique**, par exemple parce que l’application est fermée.

Lorsque l’application est fermée[[3]](#footnote-3), la scène de jeu (GameScene) est détruite. Cela provoque la destruction automatique de tous les sprites appartenant à cette scène. Lorsque ces sprites sont détruits, ils émettent bien sûr le signal spriteDestroyed mais il ne faut à ce moment-là, **en aucun cas**, ajouter à la scène de nouveaux sprites car la scène n’existe déjà plus et cela pourrait provoquer un crash et l’affichage dans la console du message d’erreur suivant :

ASSERT failure in *une\_classe*: "Called object is not of the correct type (class destructor may have already run)"

La solution la plus simple à ce problème est de déconnecter dans le destructeur de la classe concernée (par exemple GameCore) tous les sprites dont le signal spriteDestroyed est connecté à un slot de cette même classe (par exemple GameCore).

Il est même possible de simplifier davantage en déconnectant tous les sprites de ce slot, même s’il n’y sont pas connectés, cela évite de faire une vérification inutile.

GameCore::~**GameCore**() {

// Destruction de la scène principale du jeu.

// On se déconnecte tous les sprites présents sur la scène qui va être

// détruite (sans se soucier de leur nature), afin que la

// destruction d'un sprite ne provoque pas la création d'autres sprites sur

// une scène qui est en cours de destruction.

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

disconnect(pSprite, &Sprite::spriteDestroyed, this, &GameCore::onSpriteDestroyed);

}

delete m\_pScene;

m\_pScene = nullptr;

}

Modifions maintenant notre jeu de sorte que lorsque le joueur est détruit, GameCore signale la fin du jeu.

Il faut tout d’abord ajouter un slot privé à GameCore, nommé par exemple onSpriteDestroyed.

**gamecore.h :**

private slots:

void **onSpriteDestroyed**(Sprite\* pSprite);

Définir ensuite cette méthode dans le fichier .cpp de GameCore :

//! Un sprite a été détruit.

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

Sprite\* pDeadSprite = new Sprite(GameFramework::imagesPath() + "player\_dead.png");

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

pDeadSprite->setPos(pSprite->pos());

m\_pScene->createText(QPointF(50, 100), "FIN DU JEU !", 40, Qt::red);

}

Finalement, il faut connecter le signal spriteDestroyed de notre sprite au slot que nous venons de créer dans GameCore. Cette connexion peut se faire au moment de la création de notre sprite, dans le constructeur de GameCore.

// Instancier et initialiser les sprites ici :

Sprite\* pSprite = new Sprite(GameFramework::imagesPath() + "player\_m1.png");

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

pSprite->setTickHandler(new PlayerTickHandler*(*this*)*);

pSprite->registerForTick();

connect(pSprite, &Sprite::spriteDestroyed, this, &GameCore::onSpriteDestroyed);

# Animation avancée d’un sprite

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 modifiée ainsi :

// Instancier et initialiser les sprites ici :

Sprite\* pSprite = new Sprite(GameFramework::imagesPath() + "pacman\_1.png");

pSprite->addAnimationFrame(GameFramework::imagesPath() + "pacman\_2.png");

pSprite->addAnimationFrame(GameFramework::imagesPath() + "pacman\_1.png");

pSprite->addAnimationFrame(GameFramework::imagesPath() + "pacman\_3.png");

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

pSprite->setTickHandler(new PlayerTickHandler*(*this*)*);

pSprite->registerForTick();

pSprite->startAnimation(200); // Démarre l'animation et à l'image suivante toutes

// les 200 millisecondes

## Adapter l’animation du déplacement

### 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.

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.

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.

// Déplacement du point d'application des transformations au centre du sprite

m\_pParentSprite->setTransformOriginPoint(m\_pParentSprite->width()/2,

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

Finalement, la rotation en elle-même peut être appliquée dans le slot qui réagit à la pression sur les touches du clavier, dans notre classe PlayerTickHandler.

//! 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->setRotation(180);

break;

case Qt::Key\_Right: m\_playerDirection = QPointF(1,0);

m\_pParentSprite->setRotation(0);

break;

case Qt::Key\_Up: m\_playerDirection = QPointF(0,-1);

m\_pParentSprite->setRotation(-90);

break;

case Qt::Key\_Down: m\_playerDirection = QPointF(0,1);

m\_pParentSprite->setRotation(90);

break;

}

}

### 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 |  | 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;

for (int imageIndex = 1; imageIndex <= 8; imageIndex++) {

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

QString("marche%1.png").arg(imageIndex));

}

pSprite->setPos(20,20);

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

pSprite->setTickHandler(new PlayerTickHandler*(*this*)*);

pSprite->registerForTick();

pSprite->startAnimation(50);

Désormais, seuls les déplacements à gauche et à droite nous intéressent. On peut donc simplifier le code des méthodes de gestion des touches du clavier, dans notre classe PlayerTickHandler.

//! 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->setRotation(180);

break;

case Qt::Key\_Right: m\_playerDirection = QPointF(1,0);

m\_pParentSprite->setRotation(0);

break;

}

}

//! Une touche a été relâchée.

//! \param key Code de la touche relâchée.

void PlayerTickHandler::**onKeyReleased**(int key) {

switch (key) {

case Qt::Key\_Left:

case Qt::Key\_Right:

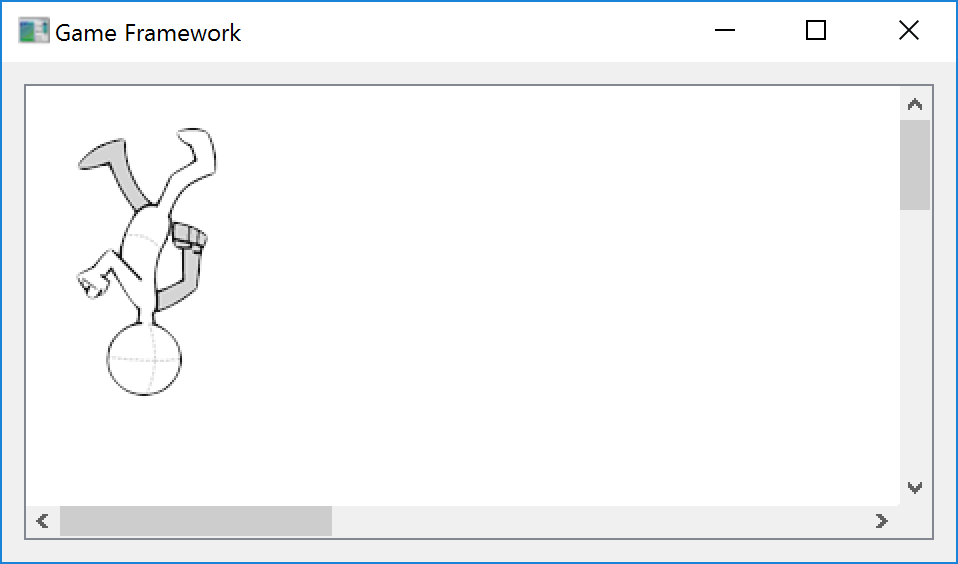
m\_playerDirection = QPointF(0,0);

break;

}

}

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 de 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 PlayerTickHandler :

**playertickhandler.h :**

#ifndef PLAYERTICKHANDLER\_H

#define PLAYERTICKHANDLER\_H

…

class **QGraphicsTransform**;

…

class **PlayerTickHandler** : public SpriteTickHandler

{

…

private:

QPointF m\_playerDirection = QPointF(0,0); // Le sprite ne bouge pas tant qu'une touche

// n'est pas appuyée

QList<QGraphicsTransform\*> m\_LeftDirectionTransform;

private slots:

void **onKeyPressed**(int key);

void **onKeyReleased**(int key);

};

#endif // PLAYERTICKHANDLER\_H

La méthode init() de PlayerTickHandler est un bon emplacement pour créer la matrice de transformation :

void PlayerTickHandler::***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 listes 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.

//! 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);

// On applique au sprite une transformation pour la gauche

m\_pParentSprite->setTransformations(m\_LeftDirectionTransform);

break;

case Qt::Key\_Right: m\_playerDirection = QPointF(1,0);

// On applique au sprite une liste de transformations vide

m\_pParentSprite->setTransformations(QList<QGraphicsTransform\*>());

break;

}

}

### 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 8.2 (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 :

// Instancier et initialiser les sprites ici :

Sprite\* pSprite = new Sprite(GameFramework::imagesPath() + "pacman\_1.png");

pSprite->addAnimationFrame(GameFramework::imagesPath() + "pacman\_2.png");

pSprite->addAnimationFrame(GameFramework::imagesPath() + "pacman\_1.png");

pSprite->addAnimationFrame(GameFramework::imagesPath() + "pacman\_3.png");

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

pSprite->setTickHandler(new PlayerTickHandler*(*this*)*);

pSprite->registerForTick();

pSprite->setAnimationSpeed(200); // Démarre l'animation et à l'image suivante toutes

// les 200 millisecondes

pSprite->setCurrentAnimationFrame(0); // Affiche la première image du sprite (pacman\_1)

connect(pSprite, &Sprite::spriteDestroyed, this, &GameCore::onSpriteDestroyed);

Le gestionnaire de tick peut ensuite être modifié ainsi au niveau de la gestion des touches du clavier :

//! 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->setRotation(180);

break;

case Qt::Key\_Right: m\_playerDirection = QPointF(1,0);

m\_pParentSprite->setRotation(0);

break;

case Qt::Key\_Up: m\_playerDirection = QPointF(0,-1);

m\_pParentSprite->setRotation(-90);

break;

case Qt::Key\_Down: m\_playerDirection = QPointF(0,1);

m\_pParentSprite->setRotation(90);

break;

}

if (!m\_pParentSprite->isAnimationRunning()) {

m\_pParentSprite->startAnimation();

}

}

//! Une touche a été relâchée.

//! \param key Code de la touche relâchée.

void PlayerTickHandler::**onKeyReleased**(int key) {

switch (key) {

case Qt::Key\_Left:

case Qt::Key\_Right:

case Qt::Key\_Up:

case Qt::Key\_Down:

m\_playerDirection = QPointF(0,0);

m\_pParentSprite->stopAnimation(Sprite::END\_OF\_CYCLE\_STOP);

break;

}

}

Cette solution simple n’est pas parfaite (l’animation démarre même si une autre touche que celles de déplacement est appuyée) mais permet néanmoins de démontrer le principe de base.

# Configurer l’apparence d’un sprite

## Point de positionnement

Par défaut, le point de positionnement du sprite (appelée également *origine* ou *hotspot*) est son coin supérieur gauche.

Il peut être parfois pratique de déplacer ce point, par exemple au centre du sprite ou au bas du sprite.

Par exemple, dans le cas où un sprite représente un personnage, il pourrait être préférable que ce point de positionnement soit situé à ses pieds.

## Adapter la taille

Si l’image du sprite est trop grande, il est possible de la redimensionner.

# Concepts avancés

## 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();

## Changement de sens animé

## Listes d’animations

Nous avons vu au chapitre §6.1 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();

}

}

## Gérer correctement l’appui sur plusieurs touches

## Gravité

## 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 = 2660;

Nous allons ensuite mettre en place (dans le constructeur de GameCore) un sprite, qui sera déplacé avec le gestionnaire de tick PlayerTickHandler qui a été développé au chapitre §3.3 et raffiné par la suite.

Nous en profitons pour ajouter un alignement de cinq murs de briques 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);

}

}

## « 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

## Quitter l’application

### Fermeture de la fenêtre par l’utilisateur

### Terminer l’application par le programme

# 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

1. Il peut être changé et déplacé (voir §7.1). [↑](#footnote-ref-1)
2. qobject\_cast est une version spécifique à *Qt* de la commande dynamic\_cast du *C++*. Pour en savoir plus : <https://stackoverflow.com/a/43995082> [↑](#footnote-ref-2)
3. Voir le diagramme de séquence qui décrit la fermeture de l’application au §9.2.2 [↑](#footnote-ref-3)