**GameFrameWork**

**Tutoriel**

# But

Le projet GameFramework 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.

Les fichiers gamecore\_blank.h et gamecore\_blank.cpp contiennent une version de la classe GameCore sans aucun exemple.

C’est à partir de ces deux fichiers (qu’il faudrait renommer en gamecore.h et gamecore.cpp) qu’il est possible de développer un jeu.

Le but de ce document est d’introduire les principaux concepts d’utilisation de ce framework.

# Tutoriel

## Ajouter un sprite à la scène

Pour ajouter un sprite à la scène, 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 :

#include "sprite.h"

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

// ...

// Instancier et initialiser les sprite ici :

// ...

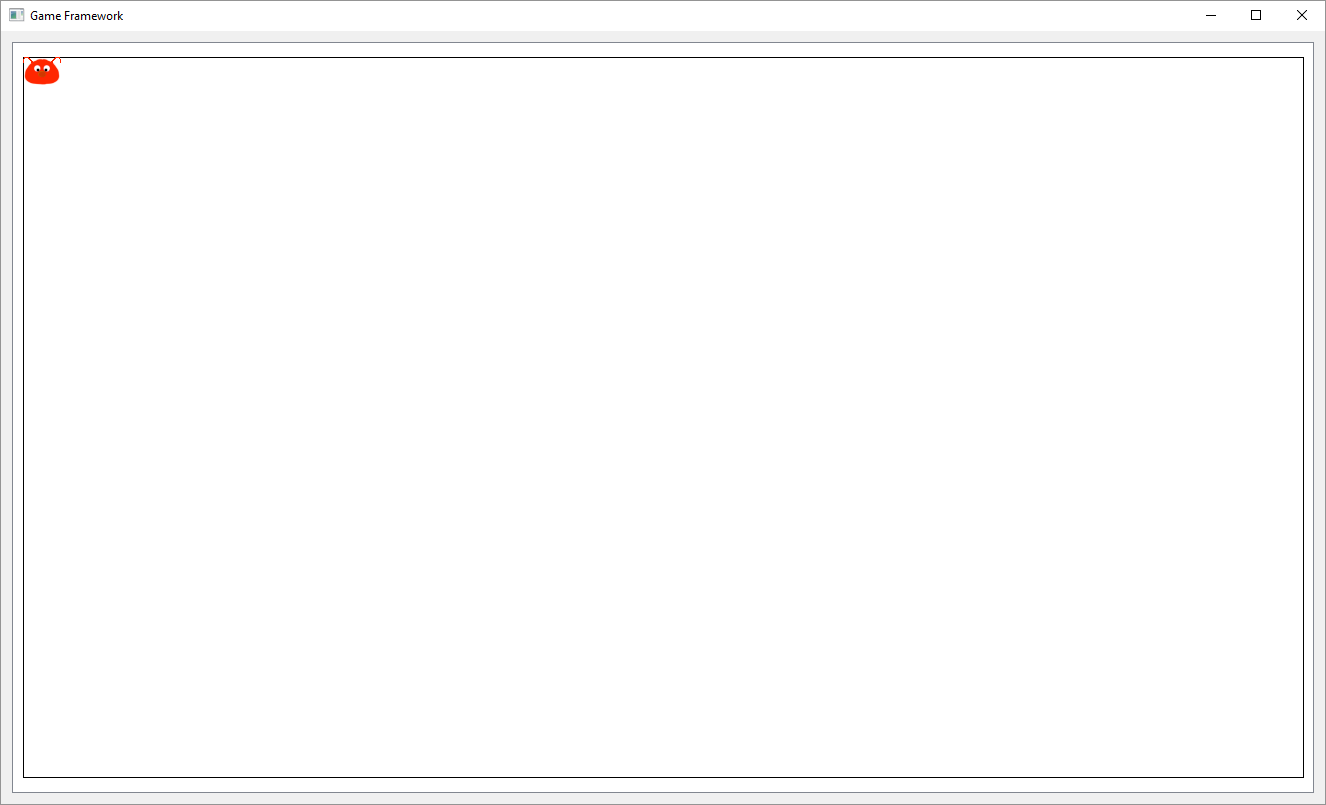
Sprite\* pSprite = new Sprite(GameFramework::imagesPath() + "player\_m1.png");

m\_pScene->addSpriteToScene(pSprite);

// ...

m\_pGameCanvas->startTick();

}



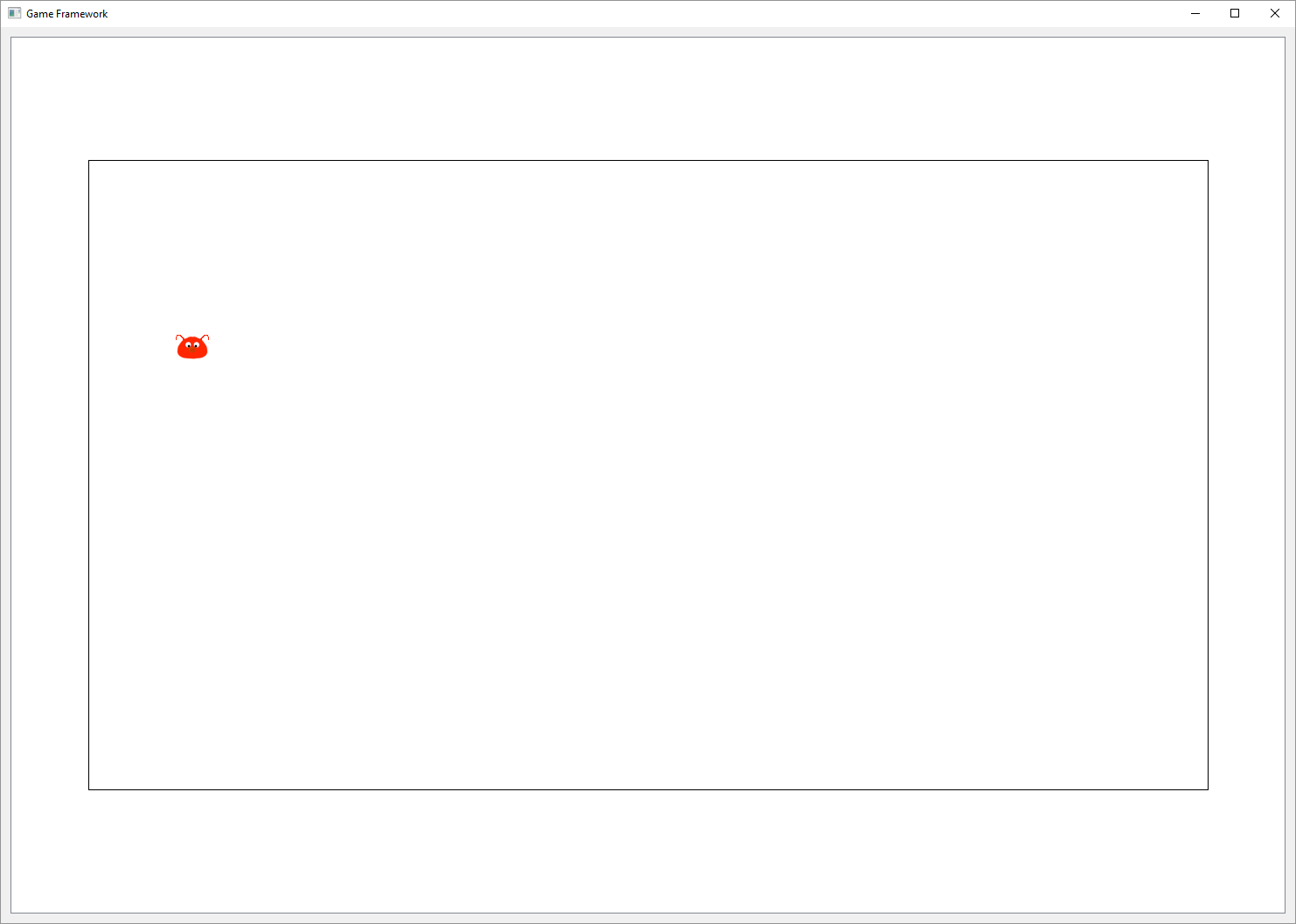
En C++, il est de la responsabilité du développeur qui instancie des objets, de s'assurer qu'ils soient détruits lorsqu'ils ne sont plus utilisés, afin de libérer la mémoire qui était occupée.

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).

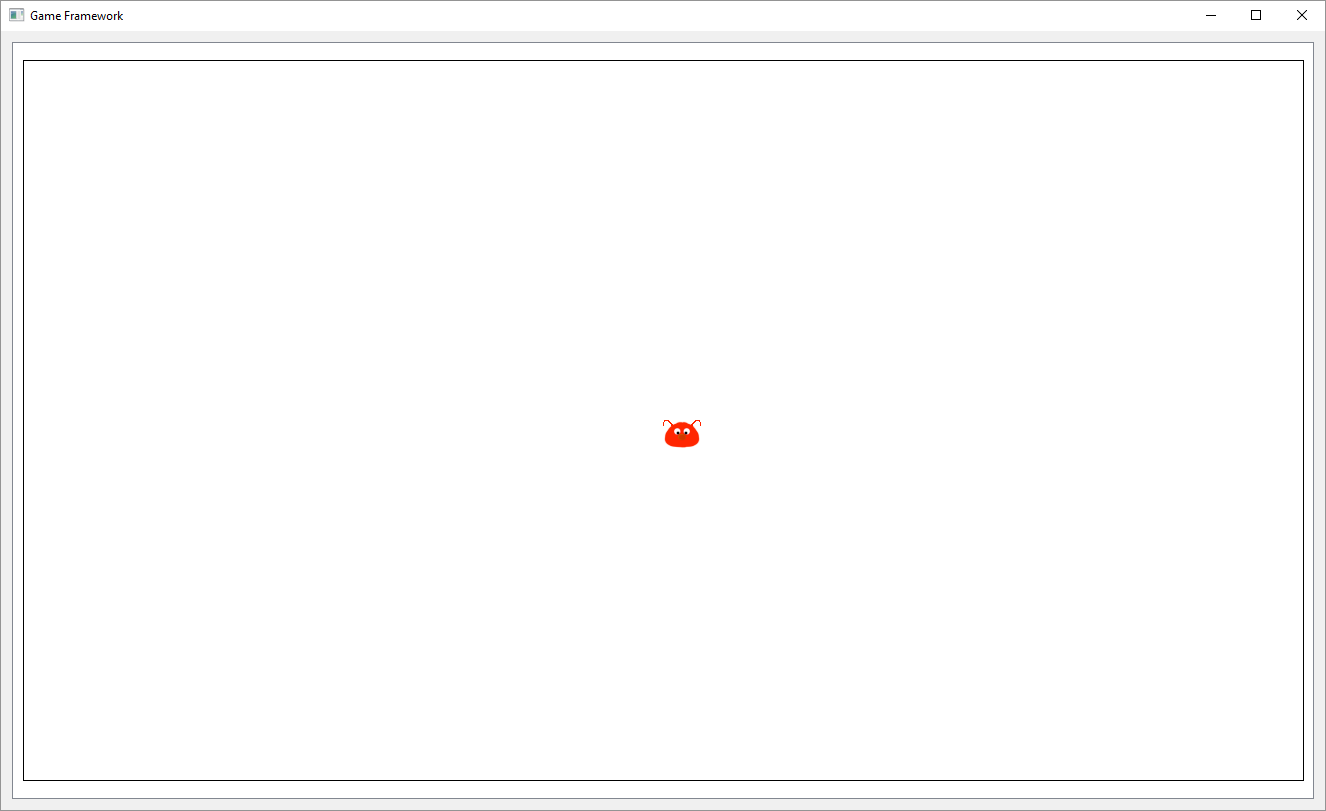
Pour le positionner ailleurs, utiliser la méthode Sprite::setPos() :

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()) :

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() :

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.

## 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 privée. Il faut donc déclarer cette variable dans le fichier d’entête gamecore.h :

private:

Sprite\* m\_pPlayer;

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 :

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

m\_pScene->addSpriteToScene(pSprite);

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.

Pour obtenir un mouvement le plus fluide possible, il faut donc déterminer la vitesse de déplacement du sprite (en pixels par seconde par exemple) puis calculer par un calcul de proportion de combien de pixels il s’est déplacé depuis le dernier tick :

const int PLAYER\_SPEED = 150 ; // vitesse de déplacement du joueur en pixels/s

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

float distance = PLAYER\_SPEED \* elapsedTimeInMilliseconds / 1000.0F;

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.

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

**gamecore.h :**

int m\_playerDirection = 1;

**gamecore.cpp :**

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

float distance = PLAYER\_SPEED \* elapsedTimeInMilliseconds / 1000.0F \*

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;

}

### Déplacement par un *tick handler*

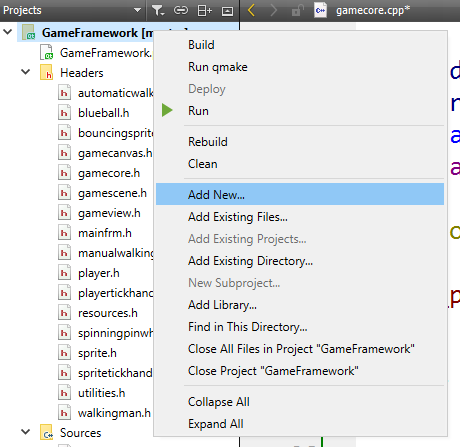
Une autre façon de gérer le déplacement d’un sprite, plus proche d’une philosophie orientée-objet, est de placer cette gestion dans une classe dédiée.

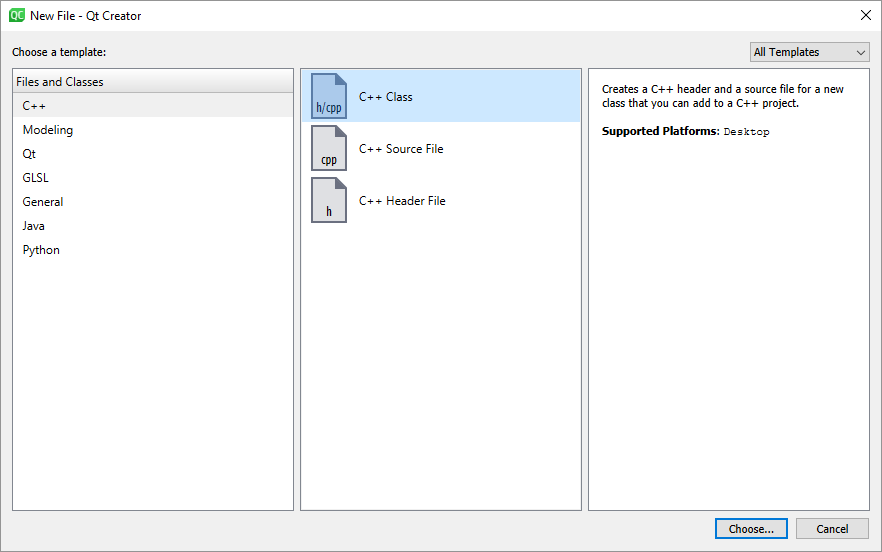
Le GameFramework propose une classe de base, SpriteTickHandler, qui peut être spécialisée pour gérer un déplacement selon une stratégie particulière, puis affectée à un sprite pour que celui-ci se déplace selon cette stratégie.

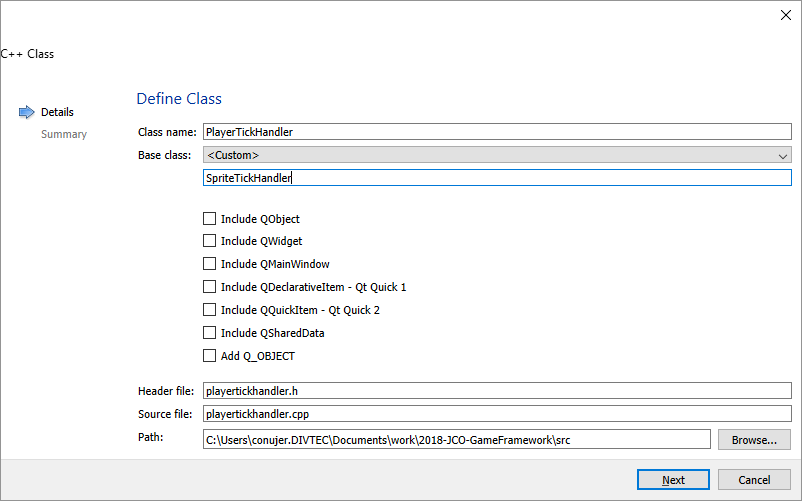
Lorsqu'un gestionnaire est affecté à un sprite (Sprite::setTickHandler()), la fonction init() du gestionnaire est appelée une unique fois, puis, lorsque la cadence démarre, la fonction tick() du gestionnaire est automatiquement appelée.

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.







**playertickhandler.h :**

#ifndef PLAYERTICKHANDLER\_H

#define PLAYERTICKHANDLER\_H

#include <QPointF>

#include "spritetickhandler.h"

class **PlayerTickHandler** : public SpriteTickHandler

{

public:

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

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

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

private:

int m\_playerDirection = 1;

QPointF m\_initialPos;

};

#endif // PLAYERTICKHANDLER\_H

**playertickhandler.cpp :**

#include "playertickhandler.h"

#include "sprite.h"

#include "gamescene.h"

const int PLAYER\_SPEED = 200;

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

{

}

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

m\_initialPos = m\_pParentSprite->pos();

}

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

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

m\_pParentSprite->setX(m\_pParentSprite->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;

}

Une fois le gestionnaire codé, il suffit de l’instancier et d’affecter au sprite cette instance, par exemple dans le constructeur de GameCore :

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

m\_pScene->addSpriteToScene(pSprite);

m\_pPlayer = pSprite;

m\_pPlayer->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(m\_pPlayer);

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().

À noter que le GameFramework propose différents gestionnaires de tick pour illustrer différentes façons de gérer le déplacement d’un sprite : BouncingSpriteHandler, KeyTickHandler, RandomMoveTickHandler.

## 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.

### 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) {

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

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

// Détecte si le sprite est en collision

bool collision = false;

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

bool collision = !listeCollision.isEmpty();

// Si les bords sont atteints ou s’il y a collision : le sprite change de direction

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

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

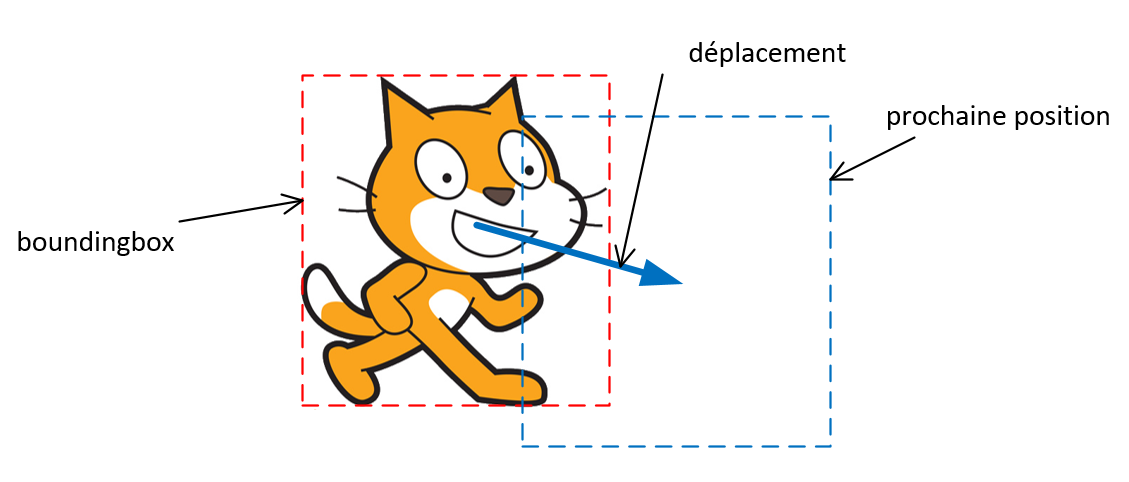
collision)

m\_playerDirection \*= -1;

}

### Vérifier avant de déplacer

L’idée de cette méthode de détection des collisions est de récupérer le rectangle qui englobe le sprite (boîte d’encombrement ou *bounding box*) avec la méthode Sprite::globalBoundingBox() 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 une *bounding box* donnée 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().

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

QPointF spriteMovement = m\_spriteVelocity \* elapsedTimeInMilliseconds / 1000.;

// Détermine la prochaine position du sprite

QRectF nextSpriteRect = m\_pParentSprite->globalBoundingBox().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 sa boundingbox

collidingSprites.removeAll(m\_pParentSprite);

bool collision = !collidingSprites.isEmpty();

// Si les bords sont atteints ou s’il y a collision : le sprite change de direction

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

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

collision)

m\_playerDirection \*= -1;

}

La solution d’utiliser la *bounding box* 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 la *bounding box*, mais la forme exacte du sprite, avec la méthode Sprite::globalShape() à la place de la méthode Sprite::globalBoundingBox().

## 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.

### Savoir qu’un sprite va être détruit

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

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

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

Dans l’exemple si 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 destroyed sera émis.

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

void **onSpriteDestroyed**(QObject\* pSprite)

Le slot reçoit en paramètre l’objet qui est sur le point d’être détruit. Dans notre cas, l’objet en question est un sprite, il faut donc convertir le pointeur pSprite :

Sprite\* pSpriteDestroyed = static\_cast<Sprite\*>(pSprite);