Dans les articles précédent sur l'implémentation du gamepad sous Windows, j'ai déjà abordé l'utilisation en QML. Nous allons voir un peu plus en détail sur un mini exemple de clone de Shoot-them-up. Le but sera simplement de déplacer le vaisseau et de pouvoir tirer.

Utiliser le gamepad

Il existe plusieurs méthodes pour exporter des éléments C++ en QML. Il est possible d'exporter un objet pour le rendre accessible en QML, de créer un singleton, d'exporter des fonctions, etc. Tout est détaillé dans le livre Créer des applications avec Qt 5 - Les essentiels au chapitre “Qt Quick et le C++”. Dans cet exemple, on va exporter la classe Gamepad en QML avec la fonction qmlRegisterType.

```
#include <QtQml>
int main(int argc, char *argv[])
{
    qmlRegisterType<Gamepad_Properties>(
        "Gamepad", 1, 0, "Gamepad");
}
```

Cette fonction prend la classe C++ comme argument template, le nom du module à importer en QML avec le numéro de version et le nom du type QML à créer.

Dans le code QML, il faut donc importer ce module et créer un élément Gamepad.

```
import QtQuick 2.0
import Gamepad 1.0

Item {
    Gamepad { id: gamepad }
}
```

On pourra ensuite accéder directement aux propriétés de Gamepad.

```
gamepad.x
```

```
gamepad.y
```

Le code d'exemple QML

Le fichier main.qml

Le premier élément à créer un timer, qui servira à mettre à jour la position du vaisseau, les tirs et les ennemis. Il faut un délai de mise à jour suffisant rapide pour que le jeu ne saccade pas, ni trop rapide pour éviter de gaspiller les ressources graphiques. Un minimum est 30 FPS (33,3 millisecondes), j'aime les comptes ronds, j'utilise un temps de 25 ms (40 FPS). Il ne faut pas hésiter à tester des valeurs plus petites (FPS plus élevé), pour voir si l'animation pose problème.

```
Timer {
    id: timer
    interval: 25
    repeat: true
    running: true
    onTriggered: update()
}
```

Il faut également créer une fonction upate, qui sera appelée par le timer et qui appellera les fonction update des autres éléments QML.

```
function update() {
    player.update()
    // les autres update, quand il y en aura...
}
```

On peut ensuite créer le vaisseau contrôlé par le gamepad. On attribue les boutons du gamepad pour tirer et le stick analogique de gauche pour le déplacement. Il serait possible d'attacher directement la valeur du gamepad à x et y :

```
x: normalize(gamepad.x, -1, 1, 0, parent.width)
```

```
y: normalize(gamepad.y, -1, 1, 0, parent.height)
La fonction normalize est une fonction JavaScript qui
normalise une valeur.

function normalize(value, fromMin, fromMax, toMin, toMax)
{
    if ((fromMax !== fromMin))
        return ((value-fromMin)/(fromMax-fromMin)*
            (toMax-toMin)+toMin)
    else
        return 0
}
```

Cependant, avec cette approche, le vaisseau pourra se déplacer quasi instantanément d'un bord de l'écran au bord opposé, ce qui serait étrange. Pour éviter cela, le gamepad contrôlera l'accélération du vaisseau et non la position.

```
Player {
    id: player
    fire: gamepad.buttons !== 0
    ax: gamepad.x
    ay: gamepad.y
}
```

Pour l'arrière-plan, on va simplement utiliser une image fixe dans un premier temps. Dans une version plus avancée, il faudra ajouter un scrolling de l'image.

```
Image {
    anchors.fill: parent
    source: "background.png"
}
```

Pour les tirs, il est possible de créer une image pour chaque tir. Cependant, cette approche sera très coûteuse en performances et ne permettra pas de créer que quelques tirs. Pour éviter cela, nous allons utiliser le moteur de particules QML. Celui-ci nécessite la création d'au moins trois éléments :

1. un élément ParticleSystem, qui affiche les particules générées. Cet élément est unique et dimension à la taille de l'élément principal (il est donc créé dans le fichier main.qml et passé en paramètre aux éléments qui utilise ce système de particule, comme par exemple player.qml) ;
2. un élément Emitter, qui émet les particules. Chaque unité du jeu capable de tirer aura un élément Emiter attaché ;
3. un élément ImageParticle, qui permet d'utiliser une image pour générer les particules. Il faudra créer un élément ImageParticle pour chaque image utilisée.

```
ParticleSystem {
    id: particleSystem
    anchors.fill: parent
}
```

Pour terminer, il faut initialiser si nécessaire les différentes variables au démarrage. Pour cela, on utilise le signal handler attaché Component.onCompleted. Par exemple pour placer le vaisseau au centre de l'écran :

```
Component.onCompleted: {
    player.x = width / 2
    player.y = height / 2
}
```

Le code final du fichier principal main.qml est donc le suivant :

```
import QtQuick 2.0
import Gamepad 1.0
import QtQuick.Particles 2.0

Item {
    width: 600; height: 600

    Image {
        anchors.fill: parent
        source: "background.png"
    }
```

```
    Gamepad { id: gamepad }

    Player {
        id: player
        particleSystem: particleSystem
        fire: gamepad.buttons !== 0
        ax: gamepad.x
        ay: gamepad.y
    }

    Timer {
        id: timer
        interval: 25
        repeat: true
        running: true
        onTriggered: update()
    }

    ParticleSystem {
        id: particleSystem
        anchors.fill: parent
    }

    Component.onCompleted: {
        player.x = width / 2
        player.y = height / 2
    }

    function update() {
        player.update()
    }
}
```

Le fichier Player.qml

(Remarque : attention à la majuscule dans le nom du fichier)

La base de Player est une simple image du vaisseau. Pour faire simple, j'utilise une image fixe, mais il est possible d'utiliser des images animées

(AnimatedImage pour une image animée type GIF, AnimatedSprite pour un sprite).

```
Image {
    source: "player.png"
}
```

Pour le déplacement, il faut créer les propriétés ax et ay correspondantes à l'accélération. Les valeurs gamepad.x et gamepad.y étant compris entre -1 et 1, on multiplie par le nombre de pixels maximal que l'on peut se déplacer à chaque update.

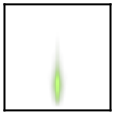
```
property real ax: 0
property real ay: 0

function update() {
    x += ax * 20
    y += ay * 20
}
```

Ainsi, dans le code précédent, à chaque appel de la fonction update, le vaisseau est déplacé (Les valeurs sont choisies arbitrairement, il suffit de tester et voir si cela convient en termes de jouabilité. Il sera possible, par la suite, d'utiliser une variable au lieu d'une valeur constante, pour par exemple proposer des bonus dans le jeu permettant d'accélérer la vitesse du vaisseau).

Pour la gestion des tirs, il faut créer un élément ImageParticule pour chaque type de tir.

Pour faire un peu plus joli, j'utilise deux images, une pour les tirs en avant et l'autre pour les tirs en arrière.



```
ImageParticle {
    system: particleSystem
    groups: ["fire"]
    source: "fire.png"
}
ImageParticle {
    system: particleSystem
    groups: ["fire_back"]
    source: "fire_back.png"
}
```

Le vaisseau se comporte que comme un émetteur de particules, il faut donc lui associer un élément Emitter qui génère les particules.

```
Emitter {
    system: particleSystem
    group: "shot"
    emitRate: 100
    lifeSpan: 2000
    enabled: gamepad.buttons !== 0
    size: 50
    velocity: PointDirection { x: -500; y: -500 }
    x: 3
    y: 36
}
```

Pour que l'élément Emitter fonctionne, il faut qu'il soit attaché à un système de particules, avec la propriété system (ou que l'élément Emitter soit enfant d'un élément ParticleSystem, ce qui n'est pas le cas

ici).

L'émetteur produit une particule (de type ImageParticle ici), avecune durée de vie (lifeSpan), un taux d'émission (emitRate), une vitesse initiale (velocity) définis.

La propriété group permet d'avoir plusieurs types de particules dans un même système, chaque type étant identifié par un groupe. Dans le code d'exemple, il y a un groupe “shot” pour les tirs en avant et un group “shot_back” pour les tirs en arrière.

La propriété size permet de définir la taille des particules générées. Les images utilisées pour les particules doivent être carrée, sinon elles seront déformées (je crois que c'est une nécessité imposée par OpenGL, non une limitation de Qt). Si ce n'est pas le cas, vous pouvez redimensionner vos images (avec Gimp par exemple) en ajoutant des zones transparentes.

Pour terminer, la propriété enable permet d'activer ou non l'émetteur. Cette propriété est liée à une expression testant si au moins un bouton du gamepad est activé. (Remarque : dans une version finale du jeu, il faudra avoir qu'un seul bouton pour tirer. De plus, il ne faudra pas appeler directement l'élément gamepad dans Player.qml, mais créer une propriété shot qui sera liée au gamepad dans main.qml)

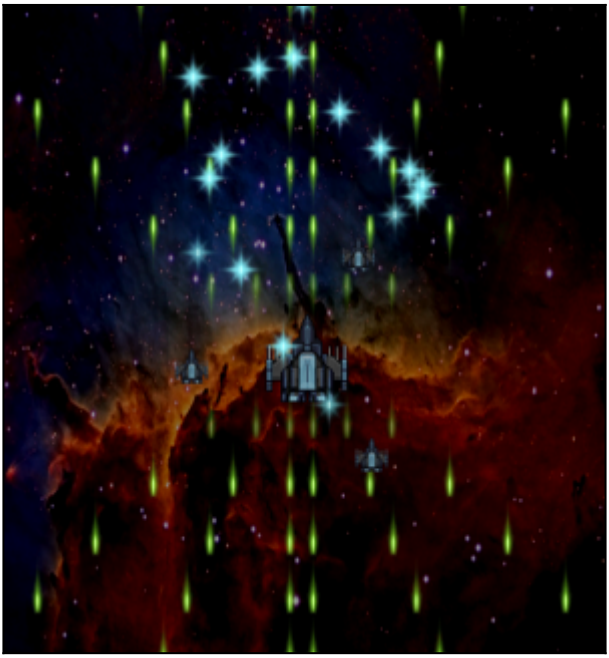
S'il y a plusieurs vaisseaux qui peuvent tirer (jeu à plusieurs joueurs, plusieurs adversaires), il faudra créer autant d'émetteur que nécessaire. Par exemple, si le vaisseau peut tirer plusieurs tirs en même temps, il est possible de mettre les éléments Emitter dans un élément Repeater pour créer plusieurs émetteurs.

```
Repeater {
    model: ListModel {
        ListElement { xx:  3; yy: 36; vx: -500; vy: -500 }
        ListElement { xx: 13; yy: 36; vx: -200; vy: -500 }
        ListElement { xx: 23; yy: 36; vx:    0; vy: -500 }
        ListElement { xx: 43; yy: 36; vx:    0; vy: -500 }
        ListElement { xx: 53; yy: 36; vx:  200; vy: -500 }
        ListElement { xx: 63; yy: 36; vx:  500; vy: -500 }
    }
```

```
    Emitter {
        velocity: PointDirection { x: vx; y: vy; }
        x: xx
        y: yy
    }
}
```

Dans l'image suivante, j'ai par exemple créer plusieurs émetteurs :

- Deux tirs parallèles en avant, deux tirs en diagonal à droite et idem à gauche, soit 6 tirs en avant.
- La même chose en arrière.
- Le vaisseau principal est accompagné de 3 mini-vaisseaux qui tirent en ligne droite devant eux (les tirs en formes d'étoiles bleues).



On peut imaginer que dans la version finale, le vaisseau ne puisse faire qu'un tir à la fois au début, puis acquière progressivement des tirs supplémentaire en activant des bonus.

Une autre implémentation

Pour terminer, une autre implémentation de la classe Gamepad. Comme je l'ai expliqué sur le forum de QtFr.org, l'utilisation d'un timer est discutable, en particulier parce que l'on a besoin d'un autre timer dans le jeu. Il n'est peut-être pas nécessaire de créer deux timers, qui ont finalement une utilisation similaire.

Une autre implémentation consisterait donc à remplacer l'héritage de QTimer par QObject et mettre la fonction update comme slot public.

```
class Gamepad_Property : public QObject
{
    Q_OBJECT
public slots:
    void update();
};
```

Dans le code QML, il faut donc maintenant appeler la fonction update manuellement.

```
function update() {
    gamepad.update()
    player.update()
}
```

Il faudra voir à l'utilisation ce qui semble le plus simple, le plus naturel à utiliser et le plus performant.

Conclusion

Dans le prochaine article, j'aborderais l'implémentation sous Linux (dès que j'aurais récupérer mon nouvel ordinateur).