Dans la première partie de cet article, j'ai abordé le fonctionnement du gamepad sous Windows et les classes de base pour la prise en charge du gamepad dans Qt (GamepadEvent). Nous allons voir maintenant l'implémentation de la classe manager, qui testera à intervalle régulier si un gamepad s'est connecté, et d'une classe qui permet de gérer un gamepad.

Classes Qt de gestion de gamepad (suite)

La classe GamepadManager

Le rôle de cette classe est de gérer un timer qui appelle un slot update à intervalle régulier pour tester la connexion des gamespads. Cette classe est également implémentée en utilisant une classe private à un pointeur D.

```
class GamepadManagerPrivate;
class GamepadManager : public QTimer
{
    Q_OBJECT
public:
    explicit GamepadManager(QObject *parent = 0);
    virtual ~GamepadManager();
private:
    GamepadManagerPrivate* d;
};
```

La classe GamepadManagerPrivate contient un constructeur pour initialiser la variable maxId (nombre maximum de gamepads non virtuels possibles) et une liste des gamepads connectés.

```
class GamepadManagerPrivate
{
public:
```

```
    GamepadManagerPrivate();
    const unsigned maxId;
    QMap<unsigned int, Gamepad*> gamepads;
};
```

Remarque : initialement, j'avais utilisé un pointeur intelligent dans la map, malheureusement Qt n'utilise pas la move semantic pour le moment avec QScopedPointeur (étonnant... je vérifierais), il n'est donc pas possible de les utiliser ensemble. J'aurais pu utiliser std::map et std::unique_ptr, mais je voulais rester dans du full-Qt.

Dans le constructeur de GamepadManager, le timer lancera le slot toutes les cinq secondes. Il sera possible de modifier cette valeur avec la propriété interval (avec les fonctions setInterval et interval de QTimer). Il est également possible d'arrêter et redémarrer le timer avec les fonctions start et stop de QTimer. Il faut également créer un objet GamepadManagerPrivate. La fonction update est appelée une premère fois, pour tester si un gamepad est connecté dès le démarrage. Dans le destructeur, on détruit le D pointeur.

```
GamepadManager::GamepadManager(QObject *parent) :
    QTimer(parent), d(new GamepadManagerPrivate)
{
    connect(&QTimer::timeout, this, &GamepadManager::update);
    start(5000);
    update();
}

GamepadManager::~GamepadManager()
{
    delete d;
}
```

Remarque : dans Qt, la création des D et Q pointeurs est simplifié, en utilisant les macros Q_D, Q_Q et Q_DECLARE_PRIVATE. Cependant, cela nécessite d'avoir accès aux en-têtes privés de Qt, ce qui n'est pas possible dans avoir une version “développeur” de Qt (version compilée soi-même). Dans la version finale, j'utiliserai ces macros.

Le slot update permet de mettre à jour la liste des gamepads connectés.

Il est possible d'appeler manuelle cette fonction.

```
public slots:
    void update();
```

L'implémentation de ce slot consiste à tester chaque identifiant de gamepad possible (donc de 0 à maxId) et d'appeler la fonction create si un gamepad est valide ou la fonction remove si un gamepad est déconnecté.

```
void GamepadManager::update()
{
    JOYINFOEX info;
    for (unsigned id = 0; id < d->maxId; ++id) {
        auto result = joyGetPosEx(id, &info);
        if (result == JOYERR_NOERROR)
            create(id);
        else if (result == JOYERR_UNPLUGGED)
            remove(id);
    }
}
```

Les fonctions create et remove testent si un gamepad existe dans la map et créer ou supprimer un gamepad correspondant à l'identifiant.

```
Gamepad* create(unsigned id);
void remove(unsigned id);
```

Pour la fonction create, on teste si un enregistrement correspond à l'identifiant et on crée un nouvel objet Gamepad (la classe Gamepad permet de gérer un gamepad, elle sera décrite par la suite). Pour la fonction remove, on récupère l'objet correspondant à l'identifiant (avec la fonction take, qui supprime l'élément de la map et retourne le pointeur vers l'objet) et on supprime l'objet.

```
Gamepad* GamepadManager::create(unsigned id)
{
    if (!d->gamepads.contains(id)) {
        d->gamepads.insert(id, new Gamepad(id, this));
    }
```

```
    return d->gamepads.value(id);
}

void GamepadManager::remove(unsigned id)
{
    Gamepad* gamepad = d->gamepads.take(id);
    if (gamepad)
        delete gamepad;
}
```

Dans le constructeur de GamepadManagerPrivate, la variable maxId est initialisée avec la fonction joyGetNumDevs de l'API Windows (il est possible d'appeler cette fonction dans la classe GamepadManager, ce qui évite de créer un constructeur dans GamepadManagerPrivate. Là, on peut mettre la variable maxId en constante... je n'ai pas de préférence).

```
GamepadManagerPrivate::GamepadManagerPrivate() :
    maxId(joyGetNumDevs())
{
}
```

La classe Gamepad

Comme pour la classe GamepadManager, la classe Gamepad est basée sur un QTimer, mais avec un intervalle plus court (par défaut, 25 millisecondes, soit 40 mise à jour par seconde). Elle est également implémentée en utilisant un pointeur D et une classe GamepadPrivate. La fonction update est appelée par le timer pour mettre à jour les données.

```
class Gamepad : public QTimer
{
    Q_OBJECT
public:
    explicit Gamepad(QObject *parent = 0);
    virtual ~Gamepad();
public slots:
    void update();
private:
    GamepadPrivate* d;
```

```
};
```

La classe GamepadPrivate contient les variables nécessaires pour conserver les capacités du gamepad. Comme il faut envoyer un événement lorsque l'état du gamepad évolue, il faut conserver l'état du courant du gamepad et l'état précédent. Pour éviter le coût d'une copie lorsque l'on change l'état courant en état précédent, l'état est géré sous forme de pointeur (plus précisément, sous forme de pointeur intelligent).

```
class GamepadPrivate
{
public:
    GamepadPrivate(unsigned id);
    ~GamepadPrivate();

    const unsigned id;
    const GamepadCapabilities capabilities;
    QScopedPointer<GamepadState> currentState;
    QScopedPointer<GamepadState> lastState;
};
```

Les classes GamepadState et GamepadCapacities sont des classes internes (nested class), qui servent à conserver l'état et les capacités du gamepad. La première propose l'opérateur != pour tester si les deux états sont différents, la seconde lit les capacités lors de la création de l'objet.

```
struct GamepadCapabilities
{
    explicit GamepadCapabilities(unsigned id);
    float xMin = 0;
    float xMax = 0;
    float yMin = 0;
    float yMax = 0;
    float zMin = 0;
    float zMax = 0;
    float rMin = 0;
    float rMax = 0;
    float uMin = 0;
    float uMax = 0;
    float vMin = 0;
```

```
    float vMax = 0;
    float povMin = 0;
    float povMax = 0;
};

struct GamepadState {
    float gamepadX = 0;
    float gamepadY = 0;
    float gamepadZ = 0;
    float gamepadR = 0;
    float gamepadU = 0;
    float gamepadV = 0;
    float gamepadPov = 0;
    unsigned pressedButtons = 0;
    bool operator!= (GamepadState const& state);
};
```

Le constructeur de GamepadCapacities utilise la fonction joyGetDevCaps pour obtenir les valeurs minimale et maximale que peuvent prendre les différents sticks du gamepad (sauf le stick POV, qui retourne une valeur entre 0 et 36000). Ces valeurs seront utilisées par la suite pour normaliser la position des sticks analogiques entre -1 et 1.

```
GamepadPrivate::GamepadCapabilities::GamepadCapabilities(
    unsigned id)
{
    JOYCAPS capabilities;
    if (joyGetDevCaps(id, &capabilities, sizeof(JOYCAPS))
            == JOYERR_NOERROR) {
        xMin = static_cast<float>(capabilities.wXmin);
        xMax = static_cast<float>(capabilities.wXmax);
        yMin = static_cast<float>(capabilities.wYmin);
        yMax = static_cast<float>(capabilities.wYmax);
        zMin = static_cast<float>(capabilities.wZmin);
        zMax = static_cast<float>(capabilities.wZmax);
        rMin = static_cast<float>(capabilities.wRmin);
        rMax = static_cast<float>(capabilities.wRmax);
        uMin = static_cast<float>(capabilities.wUmin);
        uMax = static_cast<float>(capabilities.wUmax);
        vMin = static_cast<float>(capabilities.wVmin);
        vMax = static_cast<float>(capabilities.wVmax);
```

```
        povMin = 0.0f;
        povMax = 36000.0f;
    }
}
```

L'opérateur != compare les différentes variables de deux états et retourne vrai si au moins une des variables est différente.

```
bool GamepadPrivate::GamepadState::operator!= (
    GamepadState const& state)
{
    return
            gamepadX != state.gamepadX ||
            gamepadY != state.gamepadY ||
            gamepadZ != state.gamepadZ ||
            gamepadR != state.gamepadR ||
            gamepadU != state.gamepadU ||
            gamepadV != state.gamepadV ||
            gamepadPov != state.gamepadPov ||
            pressedButtons != state.pressedButtons;
}
```

Pour revenir à la classe Gamepad, il faut initialiser le D pointeur dans le constructeur et lancer le timer. Dans le destructeur, on détruit le pointeur.

```
Gamepad::Gamepad(unsigned id, QObject *parent) :
    QTimer(parent), d(new GamepadPrivate(id))
{
    connect(this, &Gamepad::timeout, this, &Gamepad::update);
    start(25);
}

Gamepad::~Gamepad()
{
    delete d;
}
```

Pour la fonction update, on utilise encore la fonction joyGetPosEx de l'API Windows pour obtenir les valeurs du gamepad. Si la lecture s'est correctement passée, on passe l'état actuel en état précédent (avec swap), puis on normalise chaque valeur lue entre -1 et 1 (sauf pour POV

qui est normalisé entre 0 et 360). Pour terminer, on compare l'état courant avec l'état précédent, puis on envoie un événement si l'état est changé.

```
void Gamepad::update()
{
    JOYINFOEX info;
    if (joyGetPosEx(d->id, &info) == JOYERR_NOERROR)
    {
        std::swap(d->currentState, d->lastState);
        d->currentState->gamepadX = normalize(info.dwXpos,
            d->capabilities.xMin, d->capabilities.xMax,
            -1.0f, 1.0f);
        d->currentState->gamepadY = normalize(info.dwYpos,
            d->capabilities.yMin, d->capabilities.yMax,
            -1.0f, 1.0f);
        d->currentState->gamepadZ = normalize(info.dwZpos,
            d->capabilities.zMin, d->capabilities.zMax,
            -1.0f, 1.0f);
        d->currentState->gamepadR = normalize(info.dwRpos,
            d->capabilities.rMin, d->capabilities.rMax,
            -1.0f, 1.0f);
        d->currentState->gamepadU = normalize(info.dwUpos,
            d->capabilities.uMin, d->capabilities.uMax,
            -1.0f, 1.0f);
        d->currentState->gamepadV = normalize(info.dwVpos,
            d->capabilities.vMin, d->capabilities.vMax,
            -1.0f, 1.0f);
        d->currentState->gamepadPov = normalize(info.dwPOV,
            d->capabilities.povMin, d->capabilities.povMax,
            0.0f, 360.0f);
        d->currentState->pressedButtons = static_cast<
unsigned>(
            info.dwButtons);
        if (*(d->currentState) != *(d->lastState)) {
            postEvent(d->id, this, GamepadEvent::
GamepadUpdate);
        }
    }
}
```

La fonction normalize prend une valeur, ses valeurs minimale et maximale actuelles et les valeurs minimale et maximale qu'il faut utiliser pour normaliser.

```
template<class T>
float normalize(T value, float fromMin, float fromMax, float
toMin, float toMax)
{
    if ((fromMax != fromMin))
        return (static_cast<float>(value)-fromMin) /
           (fromMax-fromMin) * (toMax-toMin) + toMin;
    else
        return 0.0f;
}
```

La fonction postEvent est également une fonction utilitaire, qui vérifie s'il y a une fenêtre avec le focus et lui envoie un objet GamepadEvent si c'est le cas.

```
void postEvent(unsigned id, Gamepad* gamepad, QEvent::Type
type)
{
    if (QGuiApplication::focusObject())
    {
        GamepadEvent* event = (gamepad ?
            new GamepadEvent(gamepad, type) :
            new GamepadEvent(id, type));
        QGuiApplication::postEvent(QGuiApplication::
focusObject(),
            static_cast<QEvent*>(event));
    }
}
```

Cette fonction prend en paramètre l'identifiant du gamepad et un pointeur facultatif vers l'objet Gamepad. L'événement peut être créé à partir de l'objet s'il existe (cas d'une mise à jour d'un gamepad) ou de l'identifiant si l'objet n'existe pas (lors de la création ou la destruction de l'objet Gamepad).

Dans le constructeur et le destructeur de GamepadPrivate, on initialise

les variables et on lance un événement (GamepadConnection ou GamepadDisconnection).

```
GamepadPrivate::GamepadPrivate(unsigned id) :
    id(id),
    capabilities(id),
    currentState(new GamepadState),
    lastState(new GamepadState)
{
    postEvent(id, 0, GamepadEvent::GamepadConnection);
}

GamepadPrivate::~GamepadPrivate()
{
    postEvent(id, 0, GamepadEvent::GamepadDisconnection);
}
```

Pour terminer, il faudra accéder aux valeurs du gamepad depuis l'extérieur (par exemple dans la classe GamepadEvent). On ajoute donc des accesseurs pour les variables de GamepadPrivate dans Gamepad.

```
public:
    unsigned id() const;
    float gamepadX() const;
    float gamepadY() const;
    float gamepadZ() const;
    float gamepadR() const;
    float gamepadU() const;
    float gamepadV() const;
    float gamepadPov() const;
    unsigned pressedButtons() const;
```

Ces fonctions renvoient simplement les valeurs, je ne donne pas le détail de toutes les fonctions, elles sont identiques.

```
float Gamepad::gamepadX() const
{
    return d->currentState->gamepadX;
}
```

Conclusion

La gestion du gamepad sous Windows est donc terminée et fonctionnelle (tout au moins chez moi...) Il reste plus qu'à créer un exemple de code utilisant ces classes. J'ai commencé un mini-clone de R-Type en QML pour cela (voir en dessus). Il est possible que l'API évolue encore (elle a déjà pas mal évoluée pendant la rédaction de cet article), en fonction des cas d'utilisation que je verrais en créer l'exemple ou en fonction des performances.

Je n'ai pas utilisé, volontairement, toutes les fonctionnalités proposées par l'API Widnows, je verrais à l'utilisation si c'est nécessaire ou non (voir en particulier les structures JOYCAPS et JOYINFOEX).

Il faudra également ajouter des fonctionnalités pour gérer les gamepads en QML et les gamepads virtuels (gamepad directement sur un écran tactile – et donc créer également un élément QML pour créer des gamepads sur écran tactile).

N'hésitez pas si vous avez des remarques, en particulier sur l'API ou les fonctionnalités, n'hésitez pas à en parler sur le forum de QtFr.org.