Finalement, voici un troisième article sur l'implémentation Windows. Deux raisons (donc deux parties dans cet article) :

1. montrer l'utilisation des GamepadEvent dans une classe ;
2. montrer une implémentation alternative aux événements en utilisant les signaux-slots.

Utilisation des événements GamepadEvent

Habituellement, pour utiliser les événements (clavier ou souris par exemple), vous devez surcharger les fonctions virtuelles correspondantes dans les classes dérivées de QWidget ou de QWindow (le plus souvent keyPressEvent et keyReleaseEvent pour le clavier, mouseMoveEvent, mousePressEvent et mouseReleaseEvent pour la souris). Comment sont appelées en interne ces fonctions par le gestionnaire d'événements de Qt ?

La boucle d'événements est une simple boucle qui tourne en permanence durant toute la durée de vie de l'application. Elle gère une liste des événements à traiter, cette liste étant remplie par les différents objets Qt (comme nous l'avons fait dans le dernier article, avec la fonction postEvent). À chaque fois que la boucle traite un événement, elle appelle en fait la fonction event en passant l'événement en paramètre. La fonction virtuelle event vérifie alors le type d'événement puis appelle la fonction correspondante.

Pour prendre en compte les événements du gamepad dans une classe, il faut dériver cette classe et surcharger la fonction event. Si c'est un événement provenant d'un gamepad, on appelle des fonctions spécialisées pour les gérer (gamepadConnectEvent, gamepadDiconnectEvent et gamepadUpdateEvent). Sinon, on appelle la fonction event de la classe parent pour qu'elle puisse traiter les autres types d'événement.

Par exemple, pour gérer les événements dans une vue Qt Quick :

```
class GamepadView : public QQuickView
{
protected:
    bool event(QEvent* event) override;
    virtual bool gamepadConnectEvent(GamepadEvent* event);
    virtual bool gamepadDiconnectEvent(GamepadEvent* event);
    virtual bool gamepadUpdateEvent(GamepadEvent* event);
}
```

Dans la fonction event, on teste les différents types possibles d'événement gamepad :

```
bool GamepadView::event(QEvent* event)
{
    if (event->type() == static_cast<QEvent::Type>(
            GamepadEvent::GamepadConnection))
        return gamepadConnectEvent(
            static_cast<GamepadEvent*>(event));
    else if (event->type() == static_cast<QEvent::Type>(
            GamepadEvent::GamepadDisconnection))
        return gamepadDiconnectEvent(
            static_cast<GamepadEvent*>(event));
    else if (event->type() == static_cast<QEvent::Type>(
            GamepadEvent::GamepadUpdate))
        return gamepadUpdateEvent(
            static_cast<GamepadEvent*>(event));
   else return QQuickView::event(event);
}
```

On peut alors ajouter les fonctions de traitement des événements (remarque : il faut au moins une implémentation par défaut de ces fonctions, qui acceptent les événements et retournent vraie).

```
bool GamepadView::gamepadUpdateEvent(GamepadEvent* event)
{
    qDebug() << "gamepadUpdateEvent";
    event->accept();
    return true;
}

bool GamepadView::gamepadConnectEvent(GamepadEvent *event)
```

```
{
    qDebug() << "gamepadConnectEvent";
    event->accept();
    return true;
}

bool GamepadView::gamepadDiconnectEvent(GamepadEvent *event)
{
    qDebug() << "gamepadDiconnectEvent";
    event->accept();

    return true;
}
```

Sauf oubli, on a fini de faire le tour de la création d'événements personnalisés avec Qt. On va pouvoir passer à la suite, c'est-à-dire soit l'utilisation de ces classes dans un exemple (clone de R-Type).

Implémentation utilisant les signaux et slots

Lors de l'implémentation du code d'exemple, je me suis retrouvé confronté à la question de l'utilisation des signaux et slots. En effet, la communication entre le C++ et le QML passe par des signaux-slots, il faut

On a donc le fonctionnement suivant :

1. un timer vérifie l'état du gamepad ;
2. il émet un événement si celui-ci a changé ;
3. l'événement est géré par la classe GamepadView ;
4. elle émet un signal qui sera récupéré dans le code QML.

On peut donc se poser la question de l'utilité des événements dans ce cas et proposer une implémentation alternative connectant directement le timer au code QML.

Deux implémentations sont possibles : soit on crée une propriété pour chaque variable, soit on crée une seule propriété contenant une structure de données contenant toutes les variables. Il est possible que cela ait un impact sur les performances, il faudra donc implémenter les différentes méthodes puis faire des tests de performances.

Créer une propriété pour chaque variable

Pour la première méthode (je ne présente que pour une variable, le code pour les autres variables est similaire), il faut donc créer une propriété en lecture seule, avec un signal changed. La fonction setGamepadX est privée et n'est pas dans la propriété, elle ne sera utilisée que par le timer.

```
class Gamepad : public QTimer
{
    Q_OBJECT
    Q_PROPERTY(float x READ gamepadX NOTIFY gamepadXChanged)
public:
    float gamepadX() const;
signals:
    void gamepadXChanged();
private:
    void setGamepadX(float value);
};
```

La fonction setGamepadX teste si la nouvelle valeur est différente de la valeur actuelle et émet un signal si c'est le cas.

```
void Gamepad::setGamepadX(float value)
{
    if (m_gamepadX != value) {
        m_gamepadX = value;
        emit gamepadXChanged(m_gamepadX);
    }
}
```

Il faut également modifier la fonction update de Gamepad. Au lieu de tester l'état courant par rapport au précédent, on peut attribuer directement la nouvelle valeur à la variable.

```
void Gamepad::update()
{
    JOYINFOEX info;
    if (joyGetPosEx(d->id, &info) == JOYERR_NOERROR) {
        setGamepadX(normalize(info.dwXpos, d->capabilities.
xMin,
            d->capabilities.xMax, -1.0f, 1.0f));
    }
}
```

Cette classe est alors directement utilisable dans le code QML après l'avoir enregistrée dans le moteur QML :

```
qmlRegisterType<Gamepad>("Gamepad", 1, 0, "Gamepad");
```

On peut alors créer ce nouvel élément pour gérer le gamepad en QML.

Gamepad { id: gamepad } Image {

```
  id: player
  source: "player.png"
  x: gamepad.x
  y: gamepad.y
```

} </code>

Créer une seule propriété pour toutes les variables

Le code est similaire, mais au lieu d'exporter chaque variable, on crée une structure et on exporte celle-ci.

```
struct GamepadState {
```

```
    float x, y, z, r, u, v, pov;
    unsigned buttons;
};
Q_PROPERTY(GamepadState* state READ state NOTIFY
stateChanged)
```

Dans le code QML, les variables sont accessibles via cette propriété state.

```
x: gamepad.state.x
```

Dans le prochain article, nous verrons le code d'exemple permettant de créer un clone minimaliste de R-Type gérant le gamepad.