

Nous avons bien avancé sur les différents concepts associés à la mise en œuvre de projets à base de **QML**, appelée projets **Quick**. Nous allons approfondir nos connaissances afin de réaliser des tracés de courbes, comment gérer les dates et les heures, et surtout comment prendre en compte les capteurs intégrés aux smartphones.

## Fabrication d'une horloge

Nous allons reprendre un des projets exécuté avec des **QWidgets**, mais cette fois-ci développé en **QML**. Nous aurons d'ailleurs pas besoin de mettre en œuvre un **contrôleur** puisque tout est visuel.

*Ce projet consiste à fabriquer une horloge en temps réel. Pour cet aspect temps réel, nous utilisons le composant « **Timer** » qui doit fonctionner constamment avec des tops d'horloge soumis tous les 1/2 secondes.*

```
main.qml
import QtQuick 2.9
import QtQuick.Window 2.2

Window {
    visible: true
    width: 400
    height: 200
    title: qsTr("Horloge")
    color: "lightblue"

    Timer {
        interval: 500
        running: true
        repeat: true
        triggeredOnStart: true
        onTriggered: horloge.text = Qt.formatTime(new Date(), "hh : mm : ss")
    }

    Text {
        anchors {
            top: parent.top
            topMargin: 20
            horizontalCenter: parent.horizontalCenter
        }
        font {
            bold: true
            pixelSize: 18
        }
        color: "darkblue"
        text: Qt.formatDate(new Date(), "dddd dd MMMM yyyy")
    }

    Text {
        id: horloge
        anchors.centerIn: parent
        wrapMode: Text.WrapAnywhere
        color: "darkblue"
        font {
            bold: true
            italic: true
            pixelSize: 50
        }
    }
}
}
```

*Vous avez ci-dessous, les marqueurs spécifiques pour formater votre date et votre heure*

Motif pour la date	Résultat
d	Le jour du mois sans le préfixe 0 éventuel (1 à 31).
dd	Le jour du mois avec le préfixe 0 éventuel (01 à 31).
ddd	Jour de la semaine abrégé ('Lun' au 'Dim').
dddd	Jour de la semaine ('Lundi' au 'Dimanche').
M	Numéro du mois sans le préfixe 0 éventuel (1 à 12)
MM	Numéro du mois avec le préfixe 0 éventuel (01 à 12)
MMM	Nom du mois abrégé ('Jan' à 'Dec').
MMMM	Nom du mois complet ('Janvier' à 'Décembre').
yy	Année sur deux chiffres (00 à 99)
yyyy	Année sur quatre chiffres avec les années négatives (2000 ou 1999)
Motif pour l'heure	Résultat
h	L'heure actuelle sans le préfixe 0 éventuel (0 à 23)
hh	L'heure actuelle avec le préfixe 0 éventuel (00 à 23)
H	L'heure actuelle sans le préfixe 0 éventuel (0 à 23)

Motif pour l'heure	Résultat
HH	L'heure actuelle avec le préfixe 0 éventuel (00 à 23)
m	Les minutes sans le préfixe 0 éventuel (0 à 59)
mm	Les minutes avec le préfixe 0 éventuel (00 à 59)
s	Les secondes sans le préfixe 0 éventuel (0 à 59)
ss	Les secondes avec le préfixe 0 éventuel (00 à 59)
z	Les millisecondes sans préfixes 0 éventuel (0 à 999)
zzz	Les millisecondes sur trois chiffres (000 à 999)

### Fabrication d'un Timer

à aussi, nous allons reprendre un des projets exécuté avec des **QWidgets**, mais cette fois-ci développé en **QML**. Encore une fois, nous aurons pas besoin de mettre en œuvre un **contrôleur** puisque tout est visuel.

*Ce projet consiste à fabriquer un chronomètre au centième de seconde près. en temps réel. Cette application possède un bouton pour démarrer ou arrêter le chrono suivi de trois zones d'affichage du temps, respectivement pour les minutes, pour les secondes et pour les centièmes de seconde.*

*Nous profitons de l'occasion pour générer trois fichiers QML séparés qui vont s'occuper de la vue globale de l'application, un fichier spécifique pour le bouton et son apparence, un autre pour une des zones d'affichage temporel et enfin, le fichier principal.*

```

Bouton.qml
import QtQuick 2.0
import QtQuick.Controls 2.2

ToolButton {
    text: "Démarrer"
    anchors {
        top: parent.top
        topMargin: 50
        horizontalCenter: parent.horizontalCenter
    }
    background: Rectangle {
        color: "#10800000"
        radius: 7
    }
    width: 200
    font.pixelSize: 28
    checkable: true
}
    
```

Le fait de prévoir plusieurs fichiers permet de réduire la complexité et de réutiliser des composants, ce qui est le cas pour les zones temporelles. Si l'aspect du bouton ne vous plaît pas, vous allez directement dans ce fichier pour faire d'autres réglages.

```

LCD.qml
import QtQuick 2.9
import QtQuick.Controls 2.2

TextField {
    id: digit
    property int baseDeTemps: 60
    property int valeur: 1
    property string identité: "$"

    signal topHorloge()
    signal tempsEcoule()

    onTopHorloge: {
        valeur = valeur == baseDeTemps-1 ? 0 : valeur+1
        if (valeur==0) tempsEcoule()
    }

    onValeurChanged: text = ""+valeur

    width: 70
    readOnly: true
    horizontalAlignment: "AlignHCenter"
    color: "darkred"

    anchors {
        top: parent.top
        topMargin: 30
    }

    font {
        pixelSize: 44
        family: "Dyuthi"
    }
    Text {
        text: identité
        font.pointSize: 12
    }
}
                
```

```
anchors {
    bottom: parent.bottom
    horizontalCenter: parent.horizontalCenter
}
}
```

main.qml

```
import QtQuick 2.9
import QtQuick.Window 2.2
import QtGraphicalEffects 1.0

Window {
    visible: true
    width: 400
    height: 180
    title: qsTr("Timer")
    color: "yellow"

    LinearGradient {
        anchors.fill: parent
        gradient: Gradient {
            GradientStop { position: 0.0; color: "yellow" }
            GradientStop { position: 1.0; color: "orange" }
        }
    }

    Bouton {
        onCheckedChanged: {
            horloge.running = checked
            text = checked ? "Arrêter" : "Démarrer"
        }
    }

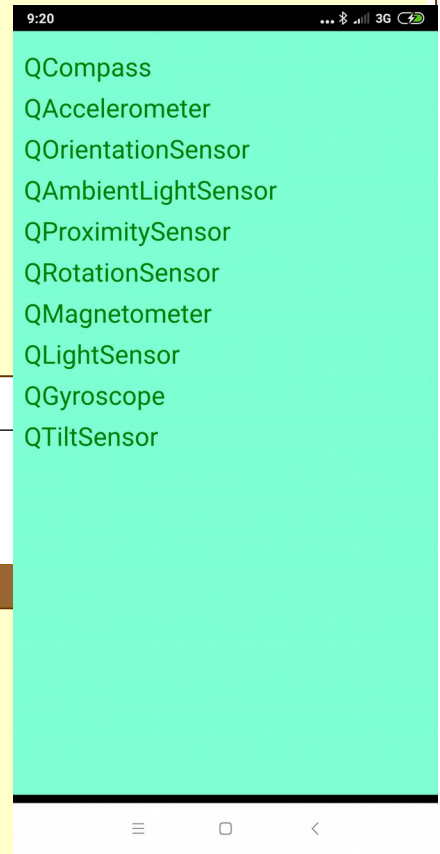
    Row {
        spacing: 20
        anchors.centerIn: parent

        Timer {
            id: horloge
            interval: 10
            repeat: true
            triggeredOnStart: true
            onTriggered: centiemmes.topHorloge()
        }

        LCD {
            id: minutes
            valeur: 0
            identité: "mn"
        }

        LCD {
            id: secondes
            valeur: 0
            identité: "s"
            onTempsEcoule: minutes.topHorloge()
        }

        LCD {
            id: centiemmes
            baseDeTemps: 100
            valeur: 0
            identité: "1/100"
            onTempsEcoule: secondes.topHorloge()
        }
    }
}
```



Gestion des capteurs sur un smartphone

Changeons de sujet et regardons ce que les smartphones offrent parmi tous les systèmes numériques, notamment ce qui concerne les capteurs intégrés. Dans un premier temps, je vous propose simplement de visualiser la liste des capteurs actifs dans votre propre smartphone.

main.qml

```
import QtQuick 2.9
import QtQuick.Window 2.2
import QtSensors 5.9
import QtQuick.Controls 2.4

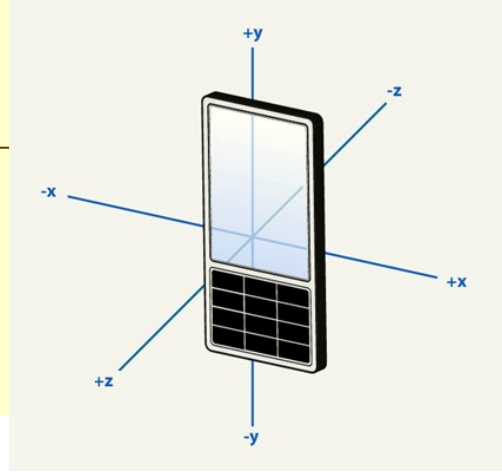
Window {
    visible: true
    width: 320
    height: 480
    color: 'black'
    title: qsTr("Tous les capteurs")
}
```

```

ScrollView {
    anchors {
        top: parent.top
        topMargin: 20
        left: parent.left
        leftMargin: 10
    }

    ListView {
        spacing: 10
        model: QmlSensors.sensorTypes()
        delegate: Text {
            font.pixelSize: 24
            color: 'green'
            text: modelData
        }
    }
}
    
```

Dans ce simple exemple, vous découvrez tous les capteurs présents sur mon smartphone. Vous remarquez qu'ils sont tous associés à une classe spécifique de Qt. Pour connaître cette liste, il suffit de faire appel à la méthode `sensorTypes()` de la classe `QmlSensors`. Dans cet exemple, vous remarquez, entre autre, la présence d'un accéléromètre « `QAccelerometer` » pour gérer tous les changements de direction du smartphone, la présence d'un gyroscope « `QGyroscope` » pour gérer l'orientation du smartphone suivant les trois axes, la présence d'une boussole « `QCompass` » pour savoir où se situe le nord magnétique, la présence d'un capteur qui mesure un champ magnétique « `QMagnetoMeter` », un capteur qui mesure l'inclinaison « `QOrientationSensor` » de votre smartphone également suivant les trois axes, etc.



### Tester l'accéléromètre

Je vous propose de tester un des capteurs pour bien voir les mesures effectuées. Nous allons vérifier que la pesanteur est bien de l'ordre de « 9,81 » en posant juste le smartphone sur la table grâce à l'accéléromètre intégré. La plus part des capteurs peuvent mesurer suivant les trois axes, nommés comme en mathématiques, « X, Y, et Z ».

Pour chaque type de capteur, vous disposez systématiquement d'une classe spécifique, ici `Accelerometer`. Il suffit ensuite d'activer ce capteur et de lancer la lecture, suivant les trois axes, grâce au slot `onReadingChanged`. Les processeurs modernes sont trop performants et le taux de rafraîchissement de la lecture est beaucoup trop rapide. Il est alors souhaitable de prévoir un `Timer` pour cadencer à la fréquence voulue.

```

Information.qml
import QtQuick 2.0

Rectangle {
    property string identité: ""
    property real valeur: 0.0

    onValeurChanged: message.text = identité+" : "+Number(valeur).toPrecision(3)

    width: parent.width
    height: 50
    radius: 7
    color: 'burlywood'
    border.color: 'maroon'

    Text {
        id: message
        x: 12
        y: 12
        color: 'maroon'
        font.bold: true
        font.pixelSize: 24
    }
}

main.qml
import QtQuick 2.9
import QtQuick.Window 2.2
import QtSensors 5.9

Window {
    visible: true
    width: 320
    height: 480
    color: 'antiquewhite'
    title: qsTr("Tous les capteurs")

    Timer {
        running: true
        repeat: true
        interval: 250
        onTriggered: accéléromètre.start()
    }
}
    
```

9:46 📶 3G 🔋

X : 0.117

Y : -0.100

Z : 9.68

☰   □   <

```

Accelerometer {
    id: accéléromètre
    active: true
    onReadingChanged: {
        x.valeur = reading.x
        y.valeur = reading.y
        z.valeur = reading.z
        stop()
    }
}

Column {
    spacing: 20
    width: parent.width
    anchors {
        top: parent.top
        topMargin: 20
        left: parent.left
        leftMargin: 10
        right: parent.right
        rightMargin: 10
    }
}

Information {
    id: x
    identité: "X"
}

Information {
    id: y
    identité: "Y"
}

Information {
    id: z
    identité: "Z"
}
}
}
}

```

Posez votre smartphone sur la table et vérifiez que sur l'axe des « Z » vous avez bien l'accélération de **9,81** (à peu près). Prenez ensuite votre smartphone et mettez-le verticalement de sorte que cette accélération soit suivant l'axe des « Y ».

## Orientation du smartphone

Prenez un autre capteur qui permet de mesurer les angles entre le smartphone et les axes du système de coordonnées. Il s'agit du capteur d'orientation qui évalue les différents mouvements rotatifs suivant les trois axes. La classe qui gère ce capteur s'appelle tout simplement **RotationSensor**. L'ossature du programme est une copie conforme du projet précédent, seule le composant du système de capture est différent.

### main.qml

```

import QtQuick 2.9
import QtQuick.Window 2.2
import QtSensors 5.9

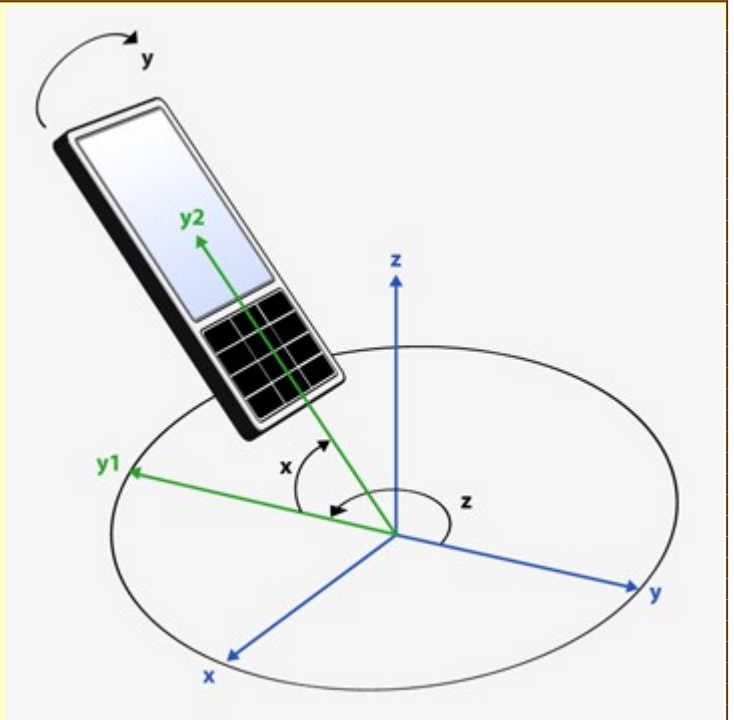
Window {
    visible: true
    width: 320
    height: 480
    color: 'antiquewhite'
    title: qsTr("Tous les capteurs")

    Timer {
        running: true
        repeat: true
        interval: 250
        onTriggered: rotation.start()
    }

    RotationSensor {
        id: accéléromètre
        active: true
        onReadingChanged: {
            x.valeur = reading.x
            y.valeur = reading.y
            z.valeur = reading.z
            stop()
        }
    }

    Column {
        spacing: 20
        width: parent.width
        anchors {
            top: parent.top
            topMargin: 20
            left: parent.left
        }
    }
}

```



```

    leftMargin: 10
    right: parent.right
    rightMargin: 10
}

Information {
    id: x
    identité: "X"
}

Information {
    id: y
    identité: "Y"
}

Information {
    id: z
    identité: "Z"
}
}
}

```

## Création d'une boussole numérique

Nous allons reprendre le capteur précédent mais cette fois-ci en tenant compte uniquement de l'axe des « Z » pour implémenter le fonctionnement d'une boussole. Il existe normalement le capteur « **Compass** » qui est prévu pour cela, mais il est plus délicat à mettre en œuvre.

*Pour la boussole, nous allons prendre une image qu'il suffit de trouver sur Internet et nous ferons tourner cette image en corrélation avec l'angle proposé par le capteur. Attention, ce capteur fournit bien un angle en degré, mais il propose des valeurs de  $-180^\circ$  à  $180^\circ$ . Il faut donc transformer cette mesure pour que la valeur s'échelonne de  $0^\circ$  à  $360^\circ$ .*

*La mesure est relativement fluctuante et donc vous verrez la boussole bouger constamment autour d'une valeur médiane qui indique finalement le cap actuel. Nous verrons par la suite comment adoucir cette fluctuation.*

*Par ailleurs, il faudra tenir votre smartphone de telle sorte qu'il soit toujours en mode « portrait ». Comme d'habitude, pensez à placer l'image de la boussole dans la ressource commune afin qu'elle soit automatiquement déployée sur votre smartphone.*

main.qml

```

import QtQuick 2.9
import QtQuick.Window 2.11
import QtSensors 5.9
Window {
    visible: true
    width: 320
    height: 480
    color: 'antiquewhite'
    title: qsTr("Tous les capteurs")

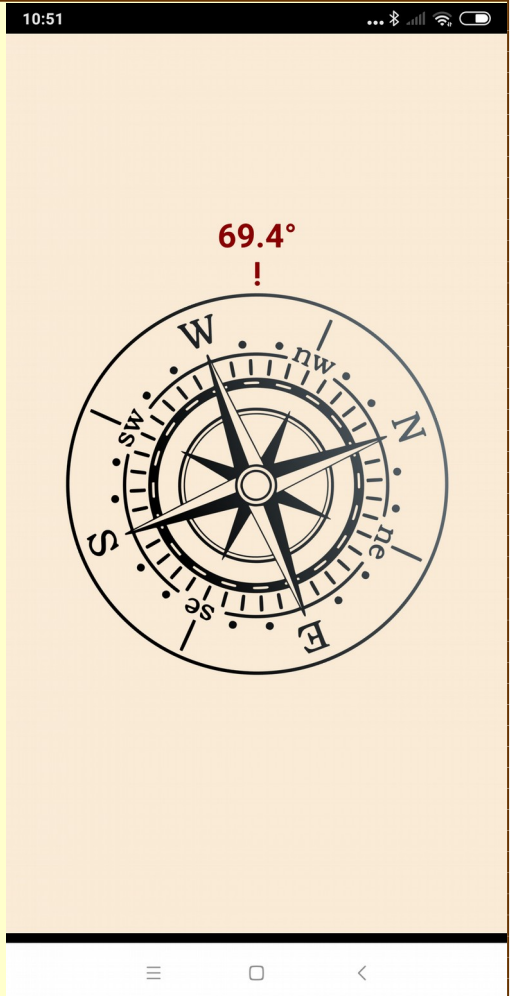
    Timer {
        running: true
        repeat: true
        interval: 250
        onTriggered: rotation.start()
    }

    RotationSensor {
        id: rotation
        active: true
        onReadingChanged: {
            angle.valeur = reading.z > 0 ? reading.z : 360 + reading.z
            boussole.rotation = angle.valeur
            stop()
        }
    }

    Text {
        id: angle
        property real valeur: 0.0
        onValueChanged: text = Number(valeur).toPrecision(3)+"°\n!"
        anchors {
            bottom: boussole.top
            horizontalCenter: parent.horizontalCenter
        }
        font.bold: true
        font.pixelSize: 26
        horizontalAlignment: Text.AlignHCenter
        color: "darkred"
    }

    Image {
        id: boussole
        width: 300
        height: 300
        source: "images/boussole.png"
        anchors.verticalCenter: parent.verticalCenter
        anchors.horizontalCenter: parent.horizontalCenter
    }
}
}

```





## Adoucir le mouvement de la boussole numérique

Modifions le code précédent afin que les mouvements soient moins saccadés. Pour cela, il suffit de prévoir une animation dont la durée correspond au temps d'acquisition, ainsi les mouvements et les changements de cap se réalise de façon plus fluides.

Nous utilisons l'animation **RotationAnimation** associée à la propriété **rotation**. Vous devez régler le début et la fin de la rotation avec respectivement l'ancienne valeur et la nouvelle valeur de cap. Attention, lorsque vous êtes près du nord, vous passez brusquement de la valeur 0° à 360° (ou inversement), il faut alors imposer la direction à suivre sinon la boussole fait un tour complet en sens inverse.

### main.qml

```
import QtQuick 2.9
import QtQuick.Window 2.11
import QtSensors 5.9

Window {
    visible: true
    width: 320
    height: 480
    color: 'antiquewhite'
    title: qsTr("Tous les capteurs")

    Timer {
        running: true
        repeat: true
        interval: 350
        onTriggered: rotation.start()
    }

    RotationSensor {
        id: rotation
        active: true
        onReadingChanged: {
            angle.valeur = reading.z > 0 ? reading.z : 360 + reading.z
            ralentir.from = boussole.rotation
            ralentir.to = angle.valeur
            if (ralentir.to-ralentir.from > 300) ralentir.direction = RotationAnimation.Counterclockwise
            else if (ralentir.from-ralentir.to > 300) ralentir.direction = RotationAnimation.Clockwise
            else ralentir.direction = RotationAnimation.Numerical
            ralentir.start()
            stop()
        }
    }

    Text {
        id: angle
        property real valeur: 0.0
        onValeurChanged: text = Number(valeur).toPrecision(3)+"°\n!"
        anchors {
            bottom: boussole.top
            horizontalCenter: parent.horizontalCenter
        }
        font.bold: true
        font.pixelSize: 26
        horizontalAlignment: Text.AlignHCenter
        color: "darkred"
    }

    Image {
        id: boussole
        width: 300
        height: 300
        source: "images/boussole.png"
        anchors.verticalCenter: parent.verticalCenter
        anchors.horizontalCenter: parent.horizontalCenter

        RotationAnimation on rotation {
            id: ralentir
            duration: 350
        }
    }
}
```

## Réaliser de tracés de courbes

La librairie Qt dispose de nombreux composants de hauts niveaux, notamment pour le tracé de courbes avec des systèmes d'axes prédéfinis. QtQuick dispose bien entendu également de ces compétences là.

Le composant principal s'appelle **ChartView**. Vous spécifiez ensuite les axes avec **ValueAxis**. Ensuite, vous rajoutez le type de courbe à visualiser grâce ici notamment à **LineSeries**. Dans le projet, il faut intégrer le module « **charts** ».

### quick-chart.pro

```
QT += quick charts
CONFIG += c++11

DEFINES += QT_DEPRECATED_WARNINGS
HEADERS += controleur.h
```

```

SOURCES += main.cpp controleur.cpp
RESOURCES += qml.qrc

# Default rules for deployment.
qnx: target.path = /tmp/${TARGET}/bin
else: unix:!android: target.path = /opt/${TARGET}/bin
!isEmpty(target.path): INSTALLS += target

```

### main.cpp

```

#include <QApplication>
#include <QQmlApplicationEngine>
#include <controleur.h>
#include <QtQml>

int main(int argc, char *argv[])
{
    QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
    QApplication app(argc, argv);

    QQmlApplicationEngine engine;
    Controleur controleur;

    engine.rootContext()->setContextProperty("controleur", &controleur);
    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));

    if (engine.rootObjects().isEmpty()) return -1;

    return app.exec();
}

```

Attention, pour ces composants spécifiques puissent être intégrés dans l'application, il faut, pour une fois, prendre la classe **QApplication** en lieu et place de **QGuiApplication** (intégration de widgets classique dans une application QML).

### controleur.h

```

#ifndef CONTROLEUR_H
#define CONTROLEUR_H

#include <QObject>
#include <QLineSeries>

using namespace QtCharts;

class Controleur : public QObject
{
    Q_OBJECT
public:
    explicit Controleur(QObject *parent = nullptr) : QObject(parent) {}
public slots:
    void selectionFichier(const QString& nomFichier, QLineSeries* courbe);
};

#endif // CONTROLEUR_H

```

Attention, vous devez rajouter l'espace de nom « **QtCharts** ».

### controleur.cpp

```

#include "controleur.h"
#include <QFile>
#include <Qurl>

void Controleur::selectionFichier(const QString& nomFichier, QLineSeries* courbe)
{
    QFile fichier(QUrl(nomFichier).toLocalFile());
    if (fichier.open(QIODevice::ReadOnly)) {
        QByteArray contenuECG = fichier.readAll();
        double pas = 0.0025;
        double x = 0;
        courbe->clear();
        courbe->append(0, 0);
        for (auto octet : contenuECG) {
            x += pas;
            double y = ((octet>=0 ? octet : octet+256) - 128)/256.0;
            courbe->append(x, y);
        }
    }
}

```

Pour une fois, le contrôleur se met directement en relation avec un composant utilisé dans la vue. Ici, il s'agit de **QLineSeries**.

### main.qml

```

import QtQuick 2.9
import QtQuick.Window 2.2
import QtCharts 2.2

```



```

import QtQuick.Controls 2.4
import QtQuick.Dialogs 1.2

Window {
    visible: true
    width: 840
    height: 480
    title: qsTr("Graphes")

    ChartView {
        title: "Électrocardiogramme"
        titleFont.pixelSize: 24
        titleColor: "darkred"
        anchors.fill: parent
        antialiasing: true

        ValueAxis {
            id: axeDesX
            titleText: "Temps (s)"
            min: 0
            max: 6
        }

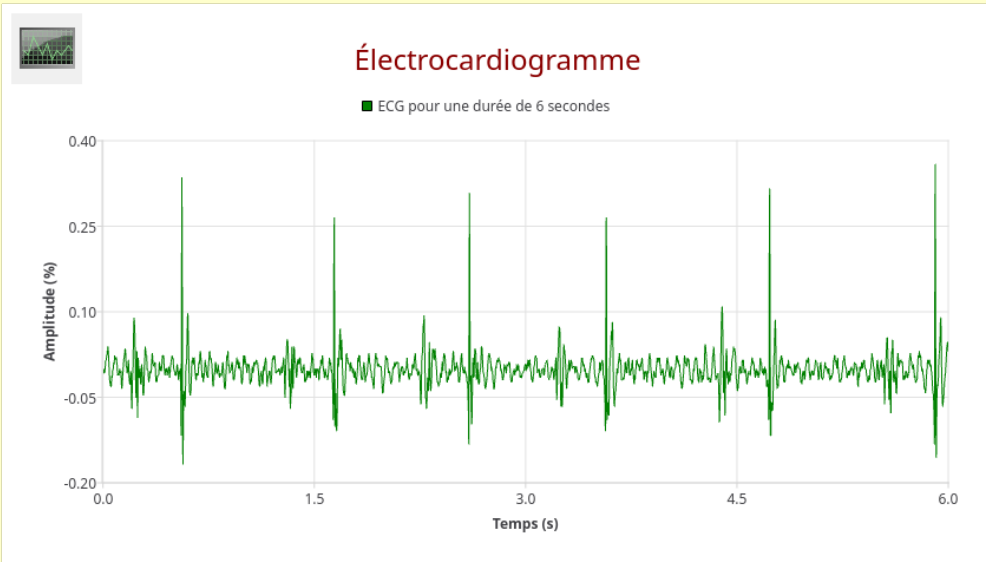
        ValueAxis {
            id: axeDesY
            titleText: "Amplitude (%)"
            min: -0.2
            max: 0.4
        }

        LineSeries {
            id: courbe
            name: "ECG pour une durée de 6 secondes"
            axisX: axeDesX
            axisY: axeDesY
            color: "green"
            useOpenGL: true
        }
    }

    ToolButton {
        icon {
            source: "qrc:/icone/courbe.png"
            color: "transparent"
            width: 48
            height: 48
        }
        anchors {
            top: parent.top
            topMargin: 8
            left: parent.left
            leftMargin: 8
        }
        onClicked: fichier.open()
    }

    FileDialog {
        id: fichier
        title: "Choisissez votre courbe à visualiser"
        folder: shortcuts.home
        onAccepted: controleur.selectionFichier(fileUrl, courbe)
    }
}

```



## Création complète d'un composant graphique personnalisé

Avant de clôturer cette étude, je vous propose de fabriquer un composant graphique personnalisé qui permet de tracer des formes simples comme des cercles, des carrés et des triangles, juste à l'endroit où nous cliquons avec la souris, en prévoyant des couleurs de bord et de fond adaptés au type de forme choisi.

*Nous profitons de l'occasion pour voir comment traiter les énumérations c++ afin qu'elles puissent être directement utilisables dans le document QML correspondant avec des réglages spécifiques prévues côté vue. À ce titre, nous verrons comment implémenter correctement ce composant pour qu'il soit en interaction complète avec les autres composants visuels de l'interface graphique.*

Nous avons déjà implémenté des composants personnalisés qui avaient le plus souvent le rôle de **contrôleur**. Il devait donc systématiquement hériter de la classe **QObject**, afin qu'il puisse prendre en compte la gestion événementielle. De plus, pour permettre l'interaction avec tous les éléments visuels, ces composants devaient proposer des **propriétés** et des **slots** adaptés, d'une part pour échanger les données et d'autre part pour activer des actions spécifiques, côté **QML**.

*Dans le cas de la création d'un composant visuel, c'est légèrement différent dans le sens où vous devez proposer un tracé totalement personnalisé comme tous les composants visuels de QML qui, nous le savons, héritent tous de la classe **Item**, côté QML. À noter que ce composant visuel de base s'appelle **QuickItem** côté c++.*

*Finalement, pour créer un composant visuel d'un point de vue QML, vous devez proposer une nouvelle classe qui hérite au moins de la classe **QuickItem**. L'idéal, est d'hériter plutôt d'une classe fille qui se nomme **QuickPaintedItem** qui implémente directement la classe **QPainter** que nous avons largement utilisée lorsque nous avons créé nos propres composants visuels de type **QWidget**. Rappelons que la classe **QPainter** implémente tous les caractéristiques relatives aux tracés en permettant de choisir le type de crayon et de pinceau.*

Souvenez-vous que pour que votre classe personnalisée soit connue d'un document **QML**, elle doit être enregistrée au moyen de la fonction `qmlRegisterType<Classe>()` en spécifiant bien l'importation et le numéro de version.

## quick-chart.pro

```

QT += quick
CONFIG += c++11

DEFINES += QT_DEPRECATED_WARNINGS
SOURCES += graph.cpp main.cpp
HEADERS += graph.h
RESOURCES += qml.qrc

# Default rules for deployment.
qnx: target.path = /tmp/${TARGET}/bin
else: unix:!android: target.path = /opt/${TARGET}/bin
!isEmpty(target.path): INSTALLS += target

```

## main.cpp

```

#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include <QtQml>
#include <graph.h>

int main(int argc, char *argv[])
{
    QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
    QGuiApplication app(argc, argv);

    qmlRegisterType<Graph>("Formes", 1, 0, "Graph");

    QQmlApplicationEngine engine;
    const QUrl url(QStringLiteral("qrc:/main.qml"));
    QObject::connect(&engine, &QQmlApplicationEngine::objectCreated,
                    &app, [url](QObject *obj, const QUrl &objUrl) {
        if (!obj && url == objUrl)
            QCoreApplication::exit(-1);
    }, Qt::QueuedConnection);
    engine.load(url);
    return app.exec();
}

```

## graph.h

```

#ifndef GRAPH_H
#define GRAPH_H

#include <QQuickPaintedItem>
#include <QPainter>
#include <vector>
using namespace std;

class Graph : public QQuickPaintedItem
{
    Q_OBJECT
public:
    enum TypeForme {Cercle, Carre, Triangle};

    struct Forme
    {
        int x, y, largeur;
        TypeForme type;
    };

    Q_ENUM(TypeForme)
    Q_PROPERTY(int largeur MEMBER largeur)
    Q_PROPERTY(TypeForme type MEMBER type)

    Graph();

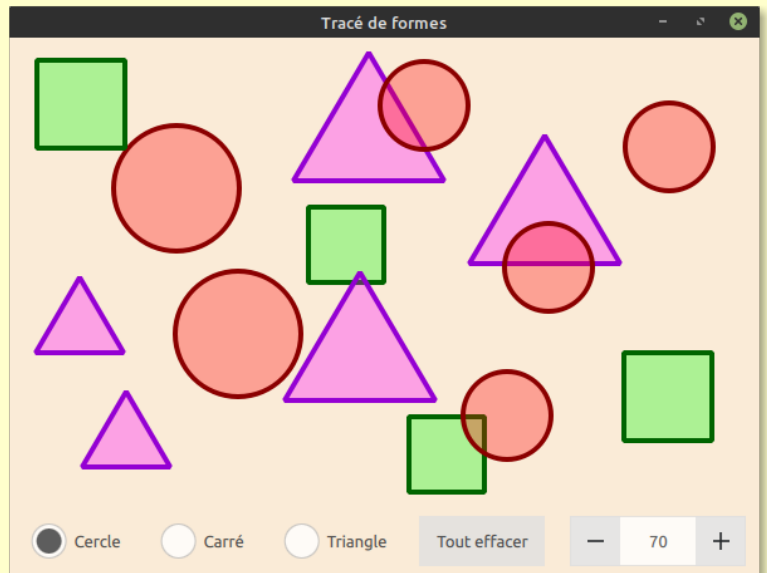
protected:
    void paint(QPainter* calque) override;
    void mousePressEvent(QMouseEvent* souris) override;

public slots:
    void effacer() { formes.clear(); update(); }

private:
    int largeur=100;
    TypeForme type=Cercle;
    vector<Forme> formes;

    QPen crayonRouge = {QColor("darkred"), 4};
    QPen crayonVert = {QColor("darkgreen"), 4};
    QPen crayonViolet = {QColor("darkviolet"), 4};
    QBrush pinceauRouge = QColor(255, 0, 0, 80);
    QBrush pinceauVert = QColor(0, 255, 0, 80);
    QBrush pinceauViolet = QColor(255, 0, 255, 80);
};
#endif // GRAPH_H

```



Pour qu'une énumération puisse être accessible côté QML, elle doit être associée à une propriété comme bien d'autres types, mais le type de l'énumération doit également être déclaré au moyen de la directive `Q_ENUM()`. De plus, puisque nous passons par la fonction `qmlRegisterType<Classe>()`, nous devons placer cette énumération au sein même de la classe que nous déployons.

## graph.cpp

```

#include "graph.h"
#include <QMouseEvent>
#include <math.h>

Graph::Graph()
{
    setAntialiasing(true);
    setAcceptedMouseButtons(Qt::AllButtons);
    setFillColor(QColor("antiquewhite"));
}

void Graph::paint(QPainter *calque)
{
    for (Forme forme : formes)
        switch (forme.type) {
            case Cercle:
                calque->setPen(crayonRouge);
                calque->setBrush(pinceauRouge);
                calque->drawEllipse(forme.x, forme.y, forme.largeur, forme.largeur);
                break;
            case Carre:
                calque->setPen(crayonVert);
                calque->setBrush(pinceauVert);
                calque->drawRect(forme.x, forme.y, forme.largeur, forme.largeur);
                break;
            case Triangle:
                int l = forme.largeur/2;
                int b = l*sqrt(3)/3;
                int h = 2*b;
                int x = forme.x+l;
                int y = forme.y+l;
                QPoint points[3] = {QPoint(x-l, y+b), QPoint(x, y-h), QPoint(x+l, y+b)};
                calque->setPen(crayonViolet);
                calque->setBrush(pinceauViolet);
                calque->drawPolygon(points, 3);
                break;
        }
}

void Graph::mousePressEvent(QMouseEvent *souris)
{
    Forme forme = {souris->x()-largeur/2, souris->y()-largeur/2, largeur, type};
    formes.push_back(forme);
    update();
}

```

Comme pour le développement en **QWidget**, nous retrouvons la même pratique, nous devons redéfinir la méthode **paint()** - au lieu de **paintEvent()** - qui prend en paramètre un objet de type **QPainter**. Grâce à ce paramètre, nous pouvons réaliser tous les tracés que nous avons l'habitude de pratiquer jusqu'à présent, ce qui nous évite d'avoir à acquérir des compétences supplémentaires.

## main.qml

```

import QtQuick 2.12
import QtQuick.Window 2.12
import QtQuick.Controls 2.12
import Formes 1.0

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Tracé de formes")

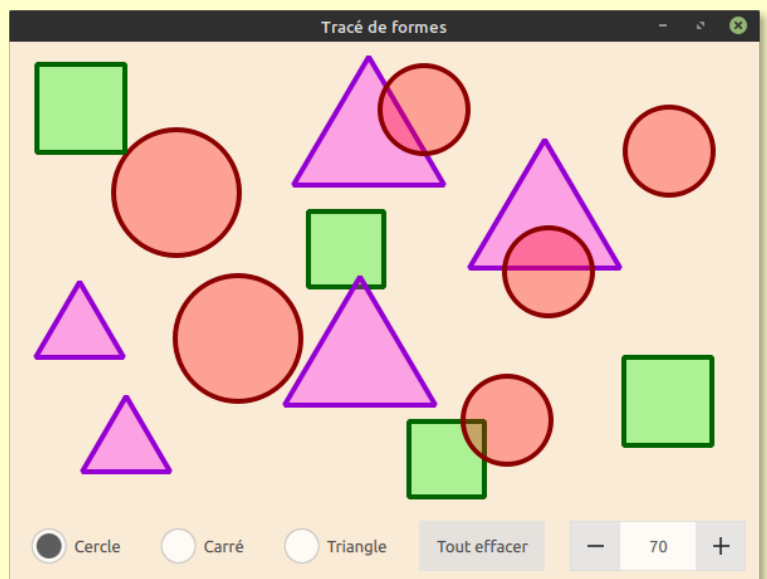
    Graph {
        id: dessin
        anchors.fill: parent
    }

    Row {
        anchors {
            horizontalCenter: parent.horizontalCenter
            bottom: parent.bottom
            bottomMargin: 10
        }
        spacing: 20

        RadioButton {
            text: "Cercle"
            checked: true
            opacity: 0.8
            onCheckedChanged: if (checked) dessin.type = Graph.Cercle
        }

        RadioButton {
            text: "Carré"
            opacity: 0.8
            onCheckedChanged: if (checked) dessin.type = Graph.Carre
        }
    }
}

```



```

RadioButton {
    text: "Triangle"
    opacity: 0.8
    onCheckedChanged: if (checked) dessin.type = Graph.Triangle
}
Button {
    text: "Tout effacer"
    opacity: 0.8
    onClicked: dessin.effacer()
}
SpinBox {
    from: 50
    to: 150
    stepSize: 10
    value: 100
    opacity: 0.8
    onValueChanged: dessin.largeur = value
}
}
}

```

Dans le document QML, nous retrouvons notre énumération où chaque élément doit être préfixé de la classe de support à savoir Graph.

### Création complète d'un composant graphique personnalisé – Gestion des formes

Nous allons rajouter quelques fonctionnalités au projet précédent qui permet de supprimer les formes déjà placées ou de les déplacer à volonté.

Je rappelle que pour qu'une énumération puisse être accessible côté QML, elle doit être associée à une propriété comme bien d'autres types, mais le type de l'énumération doit également être déclaré au moyen de la directive Q\_ENUM(). De plus, puisque nous passons par la fonction qmlRegisterType<Classe>(), nous devons placer cette énumération au sein même de la classe que nous déployons.

Pour que ce projet puisse se faire, nous rajoutons une autre énumération qui nous permettra de choisir le type de traitement à faire pour la gestion des formes. Nous en profiterons pour personnaliser l'aspect des « boutons radios ».

#### graph.h

```

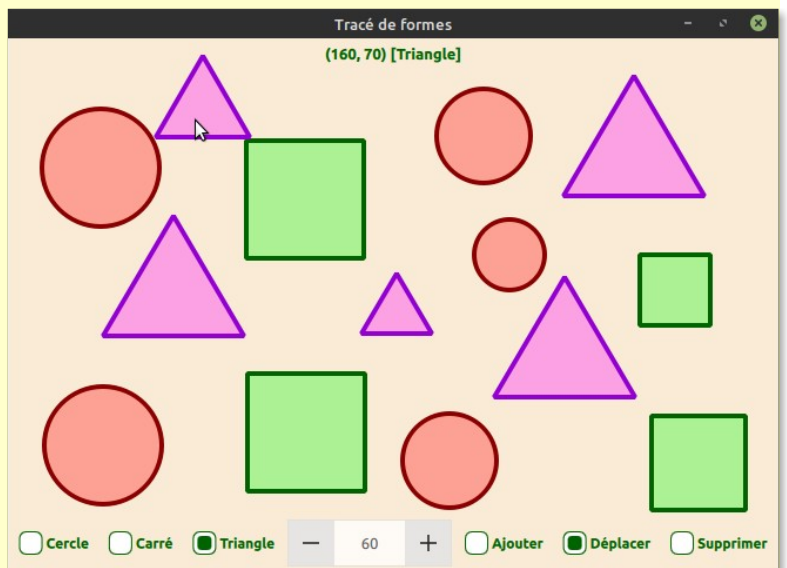
#include <QPainter>
#include <list>
using namespace std;

class Graph : public QQuickPaintedItem
{
    Q_OBJECT
public:
    enum TypeForme {Cercle, Carre, Triangle};
    enum Traitement {Ajouter, Deplacer, Supprimer};

    struct Forme
    {
        int x, y, largeur;
        TypeForme type;
        QString nom;
        bool operator==(const Forme& f) {
            return x==f.x && y==f.y && largeur==f.largeur && type==f.type;
        }
    };

    Q_ENUM(TypeForme)
    Q_ENUM(Traitement)
    Q_PROPERTY(int largeur MEMBER largeur)
    Q_PROPERTY(TypeForme type MEMBER type)
    Q_PROPERTY(QString nom MEMBER nom)
    Q_PROPERTY(Traitement fonction MEMBER fonction)
    Q_PROPERTY(QString position MEMBER position)
    Graph();
protected:
    void paint(QPainter* calque) override;
    void mousePressEvent(QMouseEvent* souris) override;
    void mouseMoveEvent(QMouseEvent* souris) override;
    void hoverMoveEvent(QHoverEvent* souris) override;
signals:
    void sourisChanged();
private:
    bool rechercheForme(int x, int y);
private:
    int largeur=100, posX=0, posY=0;
    TypeForme type=Cercle;
    QString nom="Cercle";
    Traitement fonction=Ajouter;
    list<Forme> formes;
    Forme* formActive;
    QPoint souris;
    QString position;
    QPen crayonRouge = {QColor("darkred"), 4};
    QPen crayonVert = {QColor("darkgreen"), 4};
    QPen crayonViolet = {QColor("darkviolet"), 4};
    QBrush pinceauRouge = QColor(255, 0, 0, 80);
    QBrush pinceauVert = QColor(0, 255, 0, 80);
    QBrush pinceauViolet = QColor(255, 0, 255, 80);
};

```



Dans ce projet, nous visualisons en temps réel les coordonnées de la souris et contrôlons si elle passe au dessus d'une forme particulière afin de la recenser. Pour cela, nous rajoutons un champ supplémentaire « nom » de la structure « Forme ». Nous redéfinissons également l'opérateur d'égalité afin de pouvoir contrôler qu'elle forme se trouve au-dessous du curseur de la souris. Nous modifions la collection des formes en prenant une liste en lieu et place d'un vecteur pour que la suppression puisse se faire directement.

Deux méthodes spécifiques doivent être redéfinies afin de prendre en compte le déplacement simple de la souris et la gestion du glisser-déposer, il s'agit respectivement de « hoverMoveEvent() » et de « mouseMoveEvent() ».

## graph.cpp

```
#include "graph.h"
#include <QMouseEvent>
#include <math.h>
#include <QDebug>

Graph::Graph()
{
    setAntialiasing(true);
    setAcceptHoverEvents(true);
    setAcceptedMouseButtons(Qt::LeftButton);
    setFillColor(QColor("antiquewhite"));
}

void Graph::paint(QPainter *calque)
{
    for (Forme forme : formes)
        switch (forme.type) {
            case Cercle:
                calque->setPen(crayonRouge);
                calque->setBrush(pinceauRouge);
                calque->drawEllipse(forme.x, forme.y, forme.largeur, forme.largeur);
                break;
            case Carre:
                calque->setPen(crayonVert);
                calque->setBrush(pinceauVert);
                calque->setBrush(pinceauVert);
                calque->drawRect(forme.x, forme.y, forme.largeur, forme.largeur);
                break;
            case Triangle:
                int l = forme.largeur/2;
                int b = 1*sqrt(3)/3;
                int h = 2*b;
                int x = forme.x+1;
                int y = forme.y+1;
                QPoint points[3] = {QPoint(x-1, y+b), QPoint(x, y-h), QPoint(x+1, y+b)};
                calque->setPen(crayonViolet);
                calque->setBrush(pinceauViolet);
                calque->drawPolygon(points, 3);
                break;
        }
}

void Graph::mousePressEvent(QMouseEvent *souris)
{
    posX = souris->x();
    posY = souris->y();
    switch (fonction) {
        case Ajouter: {
            Forme forme = {souris->x()-largeur/2, souris->y()-largeur/2, largeur, type, nom};
            formes.push_back(forme);
            break;
        }
        case Deplacer:
            break;
        case Supprimer:
            if (rechercheForme(souris->x(), souris->y())) formes.remove(*formActive);
            break;
    }
    update();
}

void Graph::mouseMoveEvent(QMouseEvent *souris)
{
    if (fonction==Deplacer) {
        if (rechercheForme(souris->x(), souris->y())) {
            formActive->x += souris->x()-posX;
            formActive->y += souris->y()-posY;
            posX = souris->x();
            posY = souris->y();
            update();
        }
    }
    sourisChanged();
}

void Graph::hoverMoveEvent(QHoverEvent *souris)
{
    int x=souris->pos().x();
    int y=souris->pos().y();
    if (rechercheForme(x, y)) position = QString("%1, %2 [%3]").arg(x).arg(y).arg(formActive->nom);
}
```

```

else position = QString("%1, %2) [Aucune forme]").arg(x).arg(y);
sourisChanged();
}

bool Graph::rechercheForme(int x, int y)
{
    bool auDessusForme=false;
    for (Forme& f : formes) {
        int l = f.largeur;
        if (x>f.x && x<f.x+l && y>f.y && y<f.y+l) {
            auDessusForme=true;
            formActive=&f;
            break;
        }
    }
    return auDessusForme;
}

```

Si vous souhaitez redéfinir un « bouton radio », vous devez vous préoccuper de la propriété « *indicator* » et aussi « *contentItem* », comme dans l'exemple ci-dessous

BoutonRadio.qml

```

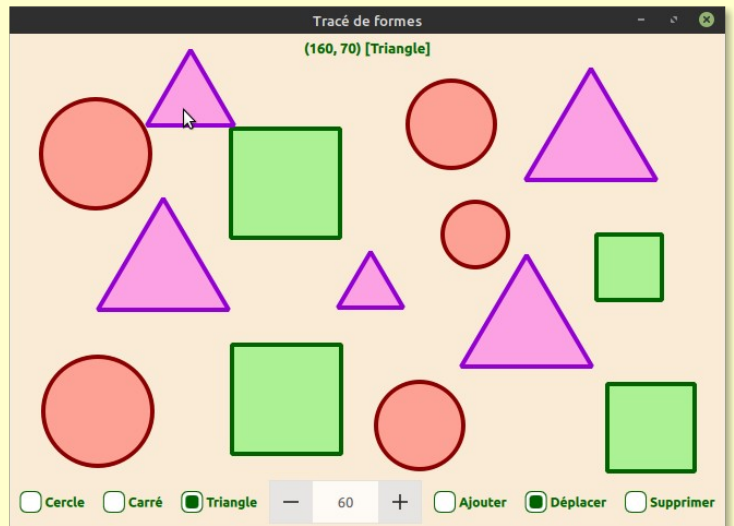
import QtQuick 2.12
import QtQuick.Controls 2.12

RadioButton {
    id: control
    spacing: 3
    anchors.verticalCenter: parent.verticalCenter

    indicator: Rectangle {
        implicitWidth: 20
        implicitHeight: 20
        x: control.leftPadding
        y: parent.height/2 - height/2
        radius: 6
        border.color: "darkgreen"
        Rectangle {
            width: 12
            height: 12
            x: 4
            y: 4
            radius: 3
            color: "darkgreen"
            visible: control.checked
        }
    }

    contentItem: Text {
        text: control.text
        font {
            bold: true
            pixelSize: 12
        }
        color: "darkgreen"
        verticalAlignment: Text.AlignVCenter
        leftPadding: control.indicator.width + control.spacing
    }
}

```



main.qml

```

import QtQuick.Controls 2.12
import QtQuick.Window 2.12
import QtQuick.Layouts 1.12
import Formes 1.0

Window {
    visible: true
    width: 720
    height: 480
    title: qsTr("Tracé de formes")

    Graph {
        id: dessin
        anchors.fill: parent
        onSourisChanged : {
            coordonnées.text = position
        }
    }

    Text {
        id: coordonnées
        anchors {
            horizontalCenter: parent.horizontalCenter
            top: parent.top
            topMargin: 5
        }
        font.bold: true
        color: "darkgreen"
    }
}

```



```

Row {
    id: boutons
    anchors {
        horizontalCenter: parent.horizontalCenter
        bottom: parent.bottom
        bottomMargin: 5
    }
    spacing: 5

    BoutonRadio {
        text: "Cercle"
        checked: true
        onCheckedChanged: if (checked) {
            dessin.type = Graph.Cercle
            dessin.nom = "Cercle"
        }
    }

    BoutonRadio {
        text: "Carré"
        onCheckedChanged: if (checked) {
            dessin.type = Graph.Carre
            dessin.nom = "Carré"
        }
    }

    BoutonRadio {
        text: "Triangle"
        onCheckedChanged: if (checked) {
            dessin.type = Graph.Triangle
            dessin.nom = "Triangle"
        }
    }

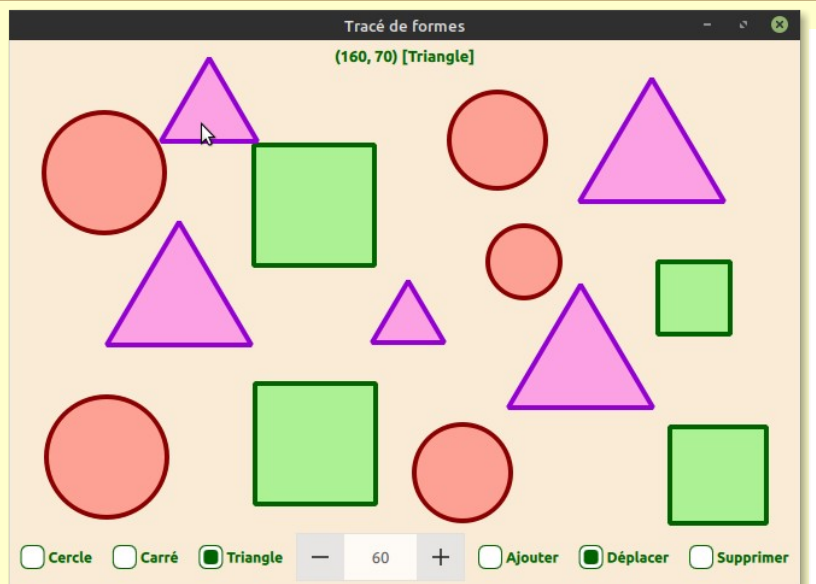
    SpinBox {
        id: largeur
        from: 50
        to: 150
        stepSize: 10
        value: 100
        opacity: 0.8
        onValueChanged: dessin.largeur = value
    }

    Row {
        anchors.verticalCenter: parent.verticalCenter
        spacing: 5

        BoutonRadio {
            text: "Ajouter"
            checked: true
            onCheckedChanged: if (checked) dessin.fonction = Graph.Ajouter
        }

        BoutonRadio {
            text: "Déplacer"
            onCheckedChanged: if (checked) dessin.fonction = Graph.Deplacer
        }

        BoutonRadio {
            text: "Supprimer"
            onCheckedChanged: if (checked) dessin.fonction = Graph.Supprimer
        }
    }
}
    
```



Dans le document QML, nous retrouvons notre énumération où chaque élément doit être préfixé de la classe de support à savoir **Graph**.