Exercice Qt n°1 : Création d'un widget permettant d'afficher et sélectionner les nuances de gris d'une couleur.

L'objet QML ColorPicker

L'objet ColorPicker permet d'afficher les nuances de gris d'une couleur et de sélectionner une nuance directement en cliquant dans l'item. Les composantes des couleurs sont manipulées directement puisqu'il n'est pas possible d'extraire les composantes d'une couleur donnée en QML. La couleur principale est définie par les variables main_red, main_green et main_blue. La couleur sélectionnée est récupérée grâce aux variables selected_red, selected_green et selected_blue dans le contexte ColorPickerContext.

La taille de l'item est fixée à 256x256 :

```
import Qt 4.7
Item
{
    width: 256
    height: 256
```

Pour dessiner les nuances de gris, on va utiliser deux gradients linéaires :

- le premier, horizontal, va du blanc à la couleur principale ;
- le second, vertical, va du transparent au noir.

Pour créer les gradients, on commence par définir un rectangle ayant les mêmes dimensions et position que l'item :

```
    Rectangle
    {
        anchors.fill: parent
    }
```

On utilise ensuite l'objet gradient pour remplir le rectangle avec le dégradé. Par exemple pour le second gradient :

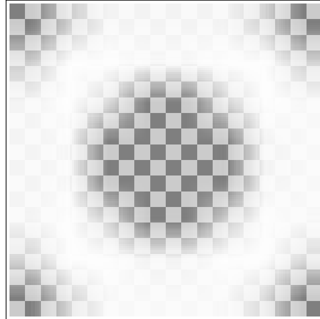
```
    Rectangle
    {
        anchors.fill: parent
        gradient: Gradient
        {
            GradientStop { position: 0.0; color: Qt.rgba(0, 0, 0, 0) }
            GradientStop { position: 1.0; color: Qt.rgba(0, 0, 0, 1) }
        }
    }
```

On obtient ainsi un gradient vertical allant du transparent au noir. Pour le premier gradient, il faut donc effectuer une rotation du rectangle de 90° après l'avoir correctement dimensionné et positionné :

```
    Rectangle
    {
        width: parent.height
        height: parent.width
        transform: Rotation { angle: 90}
        x: parent.width
        y: 0
        gradient: Gradient
        {
            GradientStop { position: 0.0; color:
```

```
                  Qt.rgba(main_red, main_green, main_blue, 1)}
            GradientStop { position: 1.0; color: "white" }
        }
    }
```

Pour visualiser la position de la couleur sélectionnée, on va afficher un cercle entourant la dernière position connue. Cependant, dessiner un cercle en QML est un peu complexe (il faut créer des objets path pour dessiner des courbes quadratiques et constituer le cercle). Pour simplifier, on va afficher une image de taille 8x8 pixels représentant un cercle avec le fond transparent :



On crée un objet image en indiquant la source du fichier. Par défaut, la taille de l'image est celle du fichier chargé :

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

Par défaut, on souhaite que le curseur soit placé au centre de l'item. Pour cela, on utilise les variables x et y :

```
    Image
    {
        x: width/2
        y: height/2
        source: "cursor.png"
    }
```

Pour déplacer, le curseur en fonction de la position de la souris, il faut pouvoir identifier cette image :

```
    Image
    {
        id: cursor
        x: width/2
        y: height/2
        source: "cursor.png"
    }
```

On pourra alors déplacer l'image en modifiant les variables x et y, en ajoutant le code suivant n'importe où dans notre item :

```
    cursor.x = 100
    cursor.y = 100
```

Pour terminer, il faut pouvoir récupérer les événements souris survenant sur notre item. Cela est réalisé en créer un objet MouseArea de même dimension que l'item :

```
    MouseArea
    {
        anchors.fill: parent
    }
```

Il faut ensuite demander à cette MouseArea de récupérer les clics sur le bouton gauche de la souris :

```
    MouseArea
    {
        anchors.fill: parent
        acceptedButtons: Qt.LeftButton
    }
```

On va réagir à deux types d'événements : un clic sur le bouton gauche, qui déclenche un événement de type onPressed, et le déplacement de la souris avec le bouton enfoncé, qui déclenche un événement de type onPositionChanged.
Lors de ces événements, on récupère la position de la souris grâce aux variables mouseX et mouseY que l'on affecte aux positions de l'image avec cursor.x et cursor.y. Cependant, il ne faut pas oublier que la position de cursor correspond au coin en haut à gauche de l'image alors que la souris correspond au centre de l'image. Il faut donc corriger en fonction des dimensions de l'image :

```
    cursor.x = mouseX - 4
    cursor.y = mouseY – 4
```

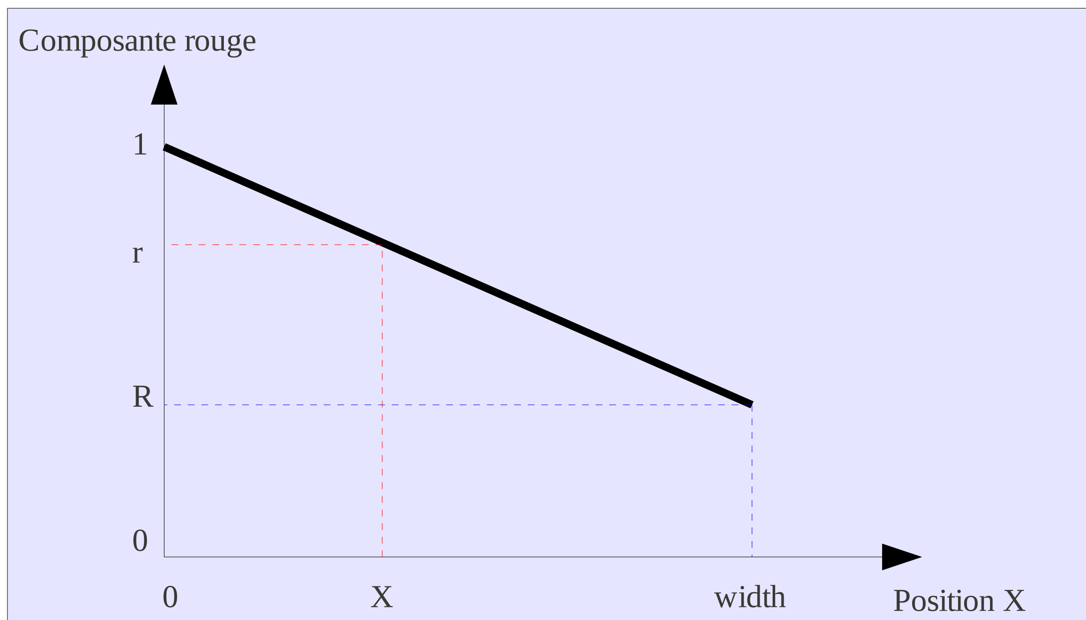
Pour terminer, après un changement de la couleur sélectionnée, il faut recalculer les variables selected_red, selected_green et selected_blue. Pour cela, on appelle un fonction JavaScript updateSelectedColor(), qu'il faudra créer :

```
    MouseArea
    {
        anchors.fill: parent
        acceptedButtons: Qt.LeftButton

        onPressed:
        {
            cursor.x = mouseX - 4
            cursor.y = mouseY - 4
            updateSelectedColor()
        }
        onPositionChanged:
        {
            cursor.x = mouseX - 4
            cursor.y = mouseY - 4
            updateSelectedColor()
        }
    }
```

Pour connaître la couleur sélectionnée, on ne dispose pas de fonction équivalente à Qimage::pixel(). Les seuls éléments que l'on dispose sont : la position (x, y) de la couleur sélectionnée dans le widget de taille (width, height) et les composantes RGB de la couleur principale.
Il faut donc calculer les composantes RGB de la couleur sélectionnée à partir de ces éléments et de la méthode utilisée pour dessiner l'item.

Lorsque l'on dessine le premier gradient, on va du blanc (1, 1, 1) à la couleur principale (R, G, B). Le rapport entre la position cursor.x et la largeur est équivalent au rapport entre la composante de la couleur principale et le blanc :



On obtient alors l'équivalence suivante :

$$\frac{r-1}{X} = \frac{R-1}{width} = pente\,de\,la\,droite$$

Alors :

$$r = 1 + (R-1)\frac{X}{width}$$

De même pour les autres composantes :

$$g = 1 + (G-1)\frac{X}{width}$$

$$b = 1 + (B-1)\frac{X}{width}$$

Lorsque l'on dessine le second gradient, on va du transparent (alpha = 0) au noir (0, 0, 0, 1). La couleur final s'écrit :

$$couleur\,finale = couleur\,initiale*(1-alpha) + noir*alpha = couleur\,initiale*(1-alpha)$$

La transparence alpha est liée à la position cursor.y par :

$$alpha = \frac{Y}{height}$$

On obtient donc la formule suivante pour la composante rouge :

$$r' = r*(1-\frac{Y}{height})$$

$$r' = (1+(R-1)\frac{X}{width})*(1-\frac{Y}{height})$$

De même pour les autres composantes :

$$g' = (1+(G-1)\frac{X}{width})*(1-\frac{Y}{height})$$
$$b' = (1+(B-1)\frac{X}{width})*(1-\frac{Y}{height})$$

La fonction JavaScript s'écrit alors, en utilisant ces formules :

```
    function updateSelectedColor()
    {
        ColorPickerContext.selected_red =
                (1 - (cursor.y / height)) *
                (1 + (cursor.x / width) * (main_red - 1))
        ColorPickerContext.selected_green =
                (1 - (cursor.y / height)) *
                (1 + (cursor.x / width) * (main_green - 1))
        ColorPickerContext.selected_blue =
                (1 - (cursor.y / height)) *
                (1 + (cursor.x / width) * (main_blue - 1))
    }
```

Le code complet de l'item ColorPicker est donc celui-ci :

```
import Qt 4.7
Item
{
    width: 256
    height: 256

    function updateSelectedColor()
    {
        ColorPickerContext.selected_red =
                (1 - (cursor.y / height)) *
                (1 + (cursor.x / width) * (main_red - 1))
        ColorPickerContext.selected_green =
                (1 - (cursor.y / height)) *
                (1 + (cursor.x / width) * (main_green - 1))
        ColorPickerContext.selected_blue =
                (1 - (cursor.y / height)) *
                (1 + (cursor.x / width) * (main_blue - 1))
    }

    Rectangle
    {
        width: parent.height
        height: parent.width
        transform: Rotation { angle: 90}
        x: parent.width
        y: 0
        gradient: Gradient
        {
            GradientStop { position: 0.0; color:
                   Qt.rgba(main_red, main_green, main_blue, 1)}
            GradientStop { position: 1.0; color: "white" }
        }
    }

    Rectangle
    {
```

```
        anchors.fill: parent
        gradient: Gradient
        {
            GradientStop { position: 0.0; color: Qt.rgba(0, 0, 0, 0) }
            GradientStop { position: 1.0; color: Qt.rgba(0, 0, 0, 1) }
        }
    }

    Image
    {
        id: cursor
        x: width/2
        y: height/2
        source: "cursor.png"
    }

    MouseArea
    {
        acceptedButtons: Qt.LeftButton
        anchors.fill: parent
        onPressed:
        {
            cursor.x = mouseX - 4
            cursor.y = mouseY - 4
            updateSelectedColor()
        }
        onPositionChanged:
        {
            cursor.x = mouseX - 4
            cursor.y = mouseY - 4
            updateSelectedColor()
        }
    }
}
```

La classe GradientWidget

Pour utiliser cet item QML dans du code C++, il est nécessaire d'écrire un wrapper pour récupérer les signaux et slots de l'item. Cette classe n'est pas un widget et dérive donc de QObject (pour pouvoir utiliser le système de signaux et slots) :

```
class GradientWidget : public QObject
{
    Q_OBJECT
```

Pour récupérer la couleur sélectionnée, il faut créer une variable pour chaque composante et utiliser la macro Q_PROPERTY pour la rendre accessible en QML. Il faut également définir les fonctions d'écriture (pour modifier la variable depuis le QML) et de lecture (pour lire la variable à l'extérieur de la classe) :

```
public:
    Q_PROPERTY(float    selected_red
               READ     selectedRed
               WRITE    setSelectedRed)

private:
    float   selected_red;
    float   selectedRed() const { return selected_red; }
    void    setSelectedRed(const float red)
                { selected_red = red; selectedColorChanged(); }
```

Après avoir mis à jour la couleur avec la fonction setSelectedColor(), il faut émettre un signal contenant la couleur créée à partir des composantes :

```
signals:
    void    colorSelected(const QColor &color);

private:
    void    selectedColorChanged()
    {
        emit colorSelected(
                QColor::fromRgbF(selected_red, selected_green, selected_blue));
    }
```

Il faut également définir les variables et fonctions pour les deux autres composantes :

```
    Q_PROPERTY(float    selected_green
               READ     selectedGreen
               WRITE    setSelectedGreen)
    Q_PROPERTY(float    selected_blue
               READ     selectedBlue
               WRITE    setSelectedBlue)
```

La création de l'item ColorPicker est réalisée dans le constructeur de la classe :

```
public:
    GradientWidget(QWidget *parent = 0);
```

Dans ce constructeur, il faut créer un objet QDeclarativeView pour afficher l'item QML puis fournir le code QML à l'aide la fonction setSource. La taille est fixée à 256x256, comme dans le code QML. Dans cet exemple, le code QML est fournit dans un fichier .qml et référencé dans un fichier ressource .qrc :

```
GradientWidget::GradientWidget(QWidget *parent)
    : QObject(parent)
{
```

```
    view = new QdeclarativeView();
    view->resize(256, 256);
    view->setSource(QUrl("qrc:/qml/colorpicker.qml"));
```

Pour permettre au code QML de transmettre les variables contenant les composantes de la couleur sélectionnée, il faut récupérer le contexte par défaut de la vue et définir une propriété ColorPickerContext dans celui-ci. Cette propriété permet d'accéder à l'objet dans le code QML à partir de la variable ColorPickerContext et permettre ainsi d'accéder aux propriétés définies ci-dessus :

```
    context = view->rootContext();
    context->setContextProperty("ColorPickerContext", this);
```

Les variables view et context sont définies dans l'en-tête de la classe par :

```
private:
    QDeclarativeView*       view;
    QDeclarativeContext*    context;
```

Lors de la création de l'item, on attribue également une couleur principale par défaut à l'aide de la fonction setMainColor() :

```
    setMainColor(Qt::red);
}
```

Cette fonction setMainColor() est un slot ayant un paramètre, la couleur a attribuer :

```
public slots:
    void    setMainColor(const QColor &color);
```

Pour transmettre cette couleur au code QML, on utilise la fonction setContextProperty(), qui permet d'attribuer une valeur à une variable QML :

```
void GradientWidget::setMainColor(const QColor &color)
{
    context->setContextProperty("main_red", color.redF());
    context->setContextProperty("main_green", color.greenF());
    context->setContextProperty("main_blue", color.blueF());
}
```

Pour terminer, il faut créer les fonctions show() et move() pour pouvoir afficher et déplacer la vue :

```
public:
    void    show();
    void    move(int x, int y);


void GradientWidget::show()
{
    view->show();
}

void GradientWidget::move(int x, int y)
{
    view->move(x, y);
}
```

Le code complet de l'en-tête est :

```
class GradientWidget : public QObject
```

```
{
    Q_OBJECT

    Q_PROPERTY(float    selected_red
               READ     selectedRed
               WRITE    setSelectedRed)
    Q_PROPERTY(float    selected_green
               READ     selectedGreen
               WRITE    setSelectedGreen)
    Q_PROPERTY(float    selected_blue
               READ     selectedBlue
               WRITE    setSelectedBlue)

public:
    GradientWidget(QWidget *parent = 0);

    void    show();
    void    move(int x, int y);

signals:
    void    colorSelected(const QColor &color);

public slots:
    void    setMainColor(const QColor &color);

private:

    void    selectedColorChanged()
    {
        emit colorSelected(QColor::fromRgbF(selected_red, selected_green,
selected_blue));
    }

    QDeclarativeView*       view;
    QDeclarativeContext*    context;

    float   selected_red;
    float   selected_green;
    float   selected_blue;

    float   selectedRed() const { return selected_red; }
    float   selectedGreen() const { return selected_green; }
    float   selectedBlue() const { return selected_blue; }

    void    setSelectedRed(const float red) { selected_red = red;
selectedColorChanged(); }
    void    setSelectedGreen(const float green) { selected_green = green;
selectedColorChanged(); }
    void    setSelectedBlue(const float blue) { selected_blue = blue;
selectedColorChanged(); }
};
```

Et le code complet de l'implémentation :

```
GradientWidget::GradientWidget(QWidget *parent)
    : QObject(parent)
{
    view = new QDeclarativeView();
    view->resize(256, 256);
    view->setSource(QUrl("qrc:/qml/colorpicker.qml"));
```

```
    context = view->rootContext();
    context->setContextProperty("ColorPickerContext", this);
    setMainColor(Qt::green);
}

void GradientWidget::show()
{
    view->show();
}

void GradientWidget::move(int x, int y)
{
    view->move(x, y);
}

void GradientWidget::setMainColor(const QColor &color)
{
    context->setContextProperty("main_red", color.redF());
    context->setContextProperty("main_green", color.greenF());
    context->setContextProperty("main_blue", color.blueF());
}
```