Qt - Module IHM
TP n°4

par Thierry Vaira © v.1.00

Sommaire

Les TP d'acquisition des fondamentaux visent à construire un socle de connaissances de base, à appréhender un concept, des notions et des modèles qui sont fondamentaux. Ce sont des étapes indispensables pour aborder d'autres apprentissages. Les TP sont conduits de manière fortement guidée pour vous placer le plus souvent dans une situation de découverte et d'apprentissage.

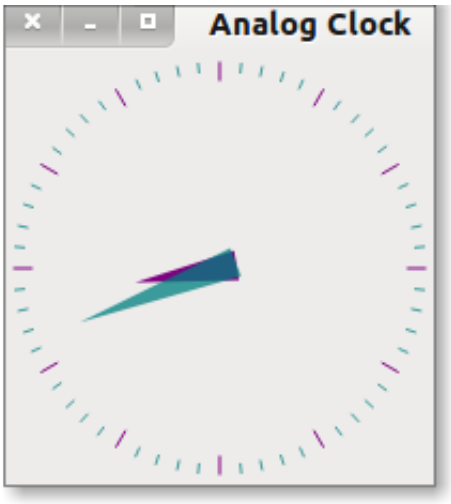
Les objectifs de ce tp sont de découvrir la programmation d'IHM sous Qt.

Une horloge analogique

L'objectif de ce TP est de découvrir les bases du dessin 2D avec **Qt**.

Tous les concepts utilisés dans ce TP sont décrits dans le support de cours qt-dessin.pdf.

On désire réaliser une horloge analogique où l'on verra les différentes grandeurs (heures et minutes) sous forme d'aiguilles dans une montre :



L'horloge analogique

Le code source de l'application :

- Qt5 : http ://doc.qt.io/qt-5/qtgui-analogclock-example.html
- Qt4 : http ://qt.developpez.com/doc/4.7/widgets-analogclock/

Télécharger le code source fourni : http ://tvaira.free.fr/info1/src-analogclock.zip

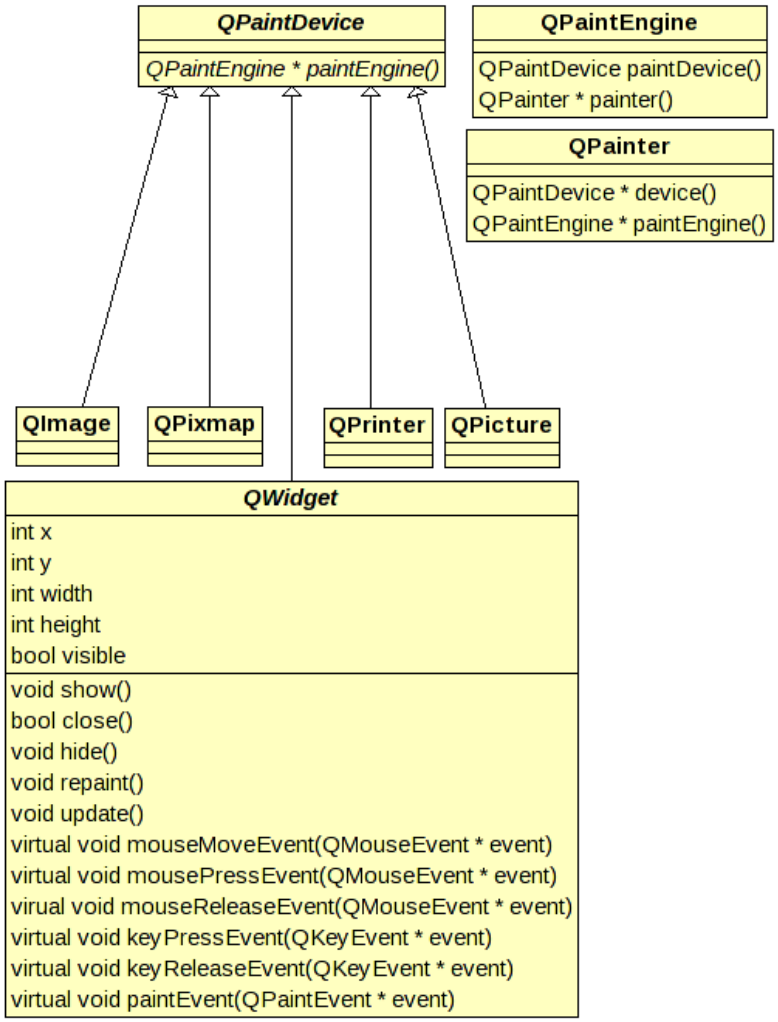
Le dessin 2D

Il existe deux approches pour dessiner en 2D dans Qt :

- un modèle fonctionnel basé sur `QPainter`
- un modèle objet basé sur le *framework* `Graphics View`

Le *framework* `Graphics View` n'est pas étudié dans ce TP.

En collaboration avec les classes `QPaintDevice` et `QPaintEngine`, `QPainter` est la base du système de dessin de Qt.



`QPaintDevice` représente un dispositif qui peut être "peint" en utilisant un `QPainter`.

`QPaintEngine` fournit le moteur de rendu et l'interface (abstraite) que `QPainter` utilise pour dessiner sur différents types de dispositifs suivant la plate-forme utilisée.

`QPainter` est la classe utilisée pour effectuer les opérations de dessin. `QPainter` peut fonctionner sur n'importe quel objet qui hérite de la classe `QPaintDevice`.

La classe `QPainter` est donc la classe de base de dessin bas niveau sur les *widgets* et les autres dispositifs de dessins.

`QPainter` fournit des fonctions hautement optimisées : il peut tout dessiner des lignes simples à des formes complexes.

L'utilisation courante de `QPainter` est à l'intérieur de la méthode `paintEvent()` d'un *widget* en suivant la démarche suivante :

a) construire un objet `QPainter` ;
b) personnaliser (par exemple le pinceau) ;
c) dessiner ;
d) détruire l'objet `QPainter` après le dessin.

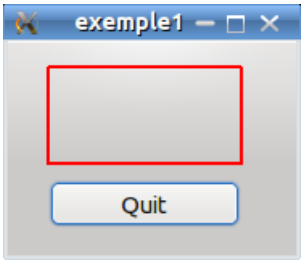
Un widget est "repeint" :

- lorsque une fenêtre passe au dessus ;
- lorsque l'on déplace le composant ;
- ... ;
- lorsque l'on le lui demande explicitement en appelant :
  - `repaint()` entraîne un rafraichissement immédiat
  - `update()` met une demande de rafraîchissement en file d'attente

Dans tous les cas, c'est la méthode `paintEvent()` qui est appelée :
`void paintEvent(QPaintEvent* e);`

Pour dessiner dans un widget, il faut donc redéfinir `QWidget::paintEvent()`.

```
class MyWidget : public QWidget
{
  public:
   MyWidget( QWidget *parent = 0 ) : QWidget( parent ) {}

   void paintEvent(QPaintEvent* e)
   {
     QWidget::paintEvent(e); // effectue le comportement standard

     QPainter painter(this); // a) construire un objet QPainter

     painter.setPen( QPen(Qt::red, 2) ); // b) personnaliser (par exemple le stylo)

     painter.drawRect( 25, 15, 120, 60 ); // c) dessiner (par exemple un rectangle)

   } // d) détruire l'objet Painter après le dessin
};
```



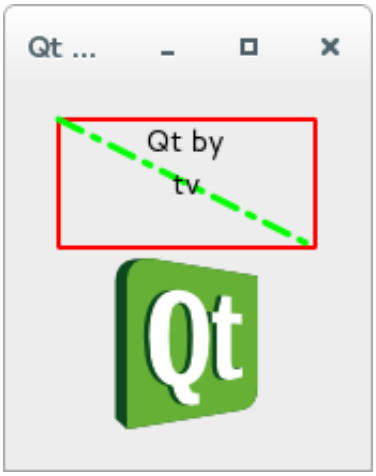
La classe `QPainter` fournit de nombreuses méthodes :

- `setPen()` : pour les lignes et contours
- `setBrush()` : pour le remplissage
- `setFont()` : pour la police de texte
- `setTransform()`, etc. : transformations affines setClipRect/Path/Region() : *clipping* (découpage)
- `setCompositionMode()` : composition
- `drawPoint()`, `drawPoints()`, `drawLine()`, `drawLines()`, `drawRect()`, `drawRects()`, `drawArc()`, `drawEllipse()`, `drawPolygon()`, `drawPolyline()`, etc ... et `drawPath()` : dessin de lignes et contours
- `fillRect()`, `fillPath()` : remplissage
- `drawText()`, `drawPixmap()`, `drawImage()`, `drawPicture()` : texte et images

Conjointement à la classe `QPainter`, on utilise de nombreuses autres classes utiles :

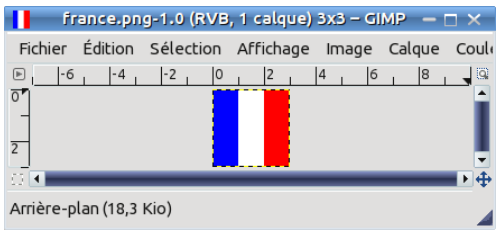
- Entiers : `QPoint`, `QLine`, `QRect`, `QPolygon`
- Flottants : `QPointF`, `QLineF`, ...
- Chemin complexe : `QPainterPath`
- Zone d'affichage : `QRegion`
- Stylo (trait) : `QPen`
- Pinceau (remplissage) : `QBrush`
- Couleur : `QColor`

```
void MyWidget::paintEvent(QPaintEvent* e)
{
  QPainter painter(this);
  painter.setRenderHint(QPainter::Antialiasing);

  QPen pen;

  // dessine un rectangle rouge
  painter.setPen( QPen(Qt::red, 2) );
  painter.drawRect( 25, 15, 120, 60 );

  // dessine une ligne pointillée verte
  pen.setStyle(Qt::DashDotLine);
  pen.setWidth(3);
  pen.setBrush(Qt::green);
  pen.setCapStyle(Qt::RoundCap);
  pen.setJoinStyle(Qt::RoundJoin);
  painter.setPen(pen);
  painter.drawLine( 25, 15, 145, 75 );

  // écrit un texte en noir
  painter.setPen( QPen(Qt::black, 2) );
  QRect rect(25, 15, 120, 60);
  painter.drawText(rect, Qt::AlignCenter, "Qt by\ntv\n\n");

  // affiche une image
  QPixmap pixmap;
  pixmap.load("qt-logo.png");
  painter.drawPixmap(45, 80, pixmap);
}
```



`QImage` fournit une représentation d'image indépendante du matériel qui permet un accès direct aux pixels.

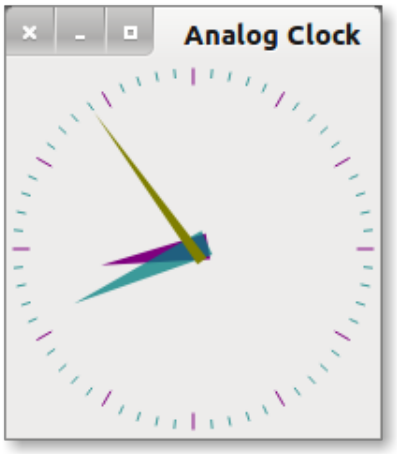
```
#include <QImage>

int main(int argc, char **argv)
{
   QImage image(3, 3, QImage::Format_RGB32);
   QRgb value;

   value = qRgb(0, 0, 255); // 0x0000FF
   for(int i = 0; i < 3; i++)
      image.setPixel(0, i, value);

   value = qRgb(255, 255, 255); // blanc
   for(int i = 0; i < 3; i++)
      image.setPixel(1, i, value);

   value = qRgb(255, 0, 0);
   for(int i = 0; i < 3; i++)
      image.setPixel(2, i, value);

   image.save("france.png", "PNG");

   return 0;
}
```



Travail demandé

Itération 1. Tester l'application fournie.

Itération 2. Modifier l'application existante pour y intégrer la gestion et l'affichage des secondes.



Remarque : il serait intéressant d'assurer un affichage "fluide" des secondes (comme pour les minutes).

Itération 3. Modifier l'application existante pour y intégrer un texte et une image.