
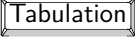





En fait, il faut lire la doc de `QWidget` - on clique sur ce mot dans *QtCreator* et . Dans la partie *Events* on peut voir tous les événements associés à un `QWidget` : `paintEvent()` (on verra cela plus tard), `mousePressEvent()`, `keyPressEvent()`... et les explications associées. En lisant on se rend compte que ces méthodes sont implantées par défaut (par exemple  ou la gestion du focus (est-ce que la souris survole ou a cliqué dans le widget), et que c'est à nous de spécifier d'autres comportements en ré-écrivant ces fonctions membres. C'est la clé (de tout ce TP) !

```
1 #include <QWidget>
2 #include <QKeyEvent>
3 #include <QMouseEvent>
4
5 class mainWidget : public QWidget
6 {
7     Q_OBJECT
8
9 public:
10     mainWidget(QWidget *parent = 0);
11     ~mainWidget();
12
13     //on surcharge les fonctions d'un widget (cf doc QT)
14     void keyPressEvent(QKeyEvent *);
15     void mouseMoveEvent(QMouseEvent *);
16     void mousePressEvent(QMouseEvent *);
17 };
```

### 3.1.2 Clavier

Par exemple, pour gérer un événement clavier, dans le widget `mainWidget`, on ré-implémente la fonction `void mainWidget::keyPressEvent(QKeyEvent * event)` (cf. « `mainwidget.cpp` »). L'événement reçu est un `QKeyEvent *` (c'est le prototype de la fonction) qui permet de savoir quelle touche a été frappée, s'il y en a eu plusieurs, son code Qt ou un texte correspondant... (allez voir la doc!). Nous voyons donc dans notre source que :

```
if( event->key() == Qt::Key_A )
```

Si la touche est un 'A', alors on déclenche un `qDebug()` différent, sinon on affiche le nom de la touche (fonctionne pour les chiffres et lettres). Si vous voulez tester la touche , il faut la tester explicitement :

```
if( event->key() == Qt::Key_F1 )
```

### 3.1.3 Souris

Pour la souris le code ré-écrit 2 fonctions d'un `QWidget` :

- `mouseMoveEvent(QMouseEvent *event)` est déclenché lorsque la souris se déplace, si on a décidé de suivre ce mouvement (cf `setMouseTracking(true)` dans le constructeur de `mainWidget`). On peut avoir la position de la souris dans le repère local du widget ((0,0) en haut à gauche), ou de l'application. On utilise ici `event->localPos()` qui donne les coordonnées *x* et *y* du pointeur de la souris. Notez que cette fonction renvoie un `QPointF`, type défini par Qt comme des coordonnées sous forme de nombre réel (que `qDebug()` sait afficher).
- `mousePressEvent(QMouseEvent *event)` est déclenché lorsque l'un des 3 boutons a été appuyé, on peut évidemment savoir quel bouton a été activé, via `event->button()` (mais aussi la position de la souris à ce moment-là, cf. la doc.).

## 3.2 Version 1 : gestion souris et clavier

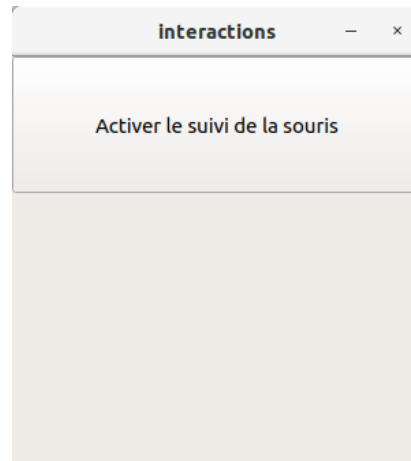


FIGURE 3.1 – Un bouton en plus

Facile, à vous donc de modifier cette application pour :

1. ajouter un bouton dans l'interface pour pouvoir suivre ou non le mouvement de la souris (voir figure 3.1). On utilise pour cela `setMouseTracking(bool)`. Il faut pour cela :
  - (a) construire l'interface graphique qui va avec. Il faut donc déclarer un `QPushButton *` dans « `mainWidget.h` » (il s'appelle `b_toggleMouseTracking` dans mon exemple), et de le définir dans le constructeur :

```
1 b_toggleMouseTracking = new QPushButton("", this); //pour l'instant il n
   'y a pas de texte sur le bouton
2
3 b_toggleMouseTracking->setFixedSize(300,100); //on lui donne la taille
   horizontale du widget (300), et 100 pixels de haut
4 b_toggleMouseTracking->setDisabled(true); //il n'est pas encore visible
   (cf plus loin)
```

- (b) ensuite il faut écrire la partie fonctionnelle : lorsque ce bouton est activé il faut 1) que `setMouseTracking()` soit appelée avec une valeur différente (`true` ou `false`); 2) que le bouton change de texte (par ex. "Activer le suivi souris" devient "Désactiver le suivi souris"). Donc pour la 1<sup>ère</sup> partie, on crée une variable booléenne `bool` dans le `mainWidget`, on lui donne une valeur par défaut dans le constructeur. Lorsque le bouton est activé, on reçoit ce signal dans un *slot*, appelons-le `toggleMouseTracking()`. Cela lance la fonction correspondante qui inverse la valeur du booléen (`monbooléen = !monbooléen`) et qui **demande** (via un nouveau signal, avec un `emit` la mise à jour du texte du bouton), c'est la 2<sup>de</sup> partie. Dans la fonction qui correspond au *slot* de ce signal on utilise `b_toggleMouseTracking -> setText("...")` pour modifier le texte sur le bouton. Le signal et le *slot* appartiennent donc à `mainWidget`, on peut mettre le signal en `private` si on veut.

Il nous font donc 2 `QObject::connect` dans le constructeur de `mainWidget` :

```
//pour modifier l'état du suivi (le booléen)
QObject::connect(b_toggleMouseTracking, SIGNAL(clicked()), this, SLOT(toggleMouseTracking()));

//pour modifier le texte du QPushButton
QObject::connect(this, SIGNAL(b_MouseTracking_update(bool)), this, SLOT(update_BMouseTracking(bool)));
```

2. ajouter un widget indépendant (dans le `main()`) : **Affichage**. Son rôle sera d'afficher les messages dus aux différents événements clavier et souris plutôt que d'utiliser un `qDebug()`.

On va faire comme dans le TP1, préparer un *slot* qui reçoit une chaîne de caractères (un `QString`) et qui l'affiche. On va faire un peu différent pour voir d'autres widgets, on utilisera un `QTextEdit` et un `QLCDNumber`.

Voilà à quoi ressemblera `Affichage.h` :

```
1 #include <QObject>
2 #include <QWidget>
3 #include <QTextEdit>
4 #include <QLCDNumber>
5 #include <QHBoxLayout>
6
7
8 class Affichage : public QWidget
9 {
10     Q_OBJECT
11 public:
12     explicit Affichage(QWidget *parent = nullptr);
13     QTextEdit * zoneTexte;
14     QLCDNumber * LCDnbEvents;
15     QHBoxLayout * Hlayout;
16     int nbEvents;
17
18 signals:
19     void LCDupdate(int);
20     void textAppend(QString);
21
22 public slots:
23     void recvData(QString);
24 };
```

On a besoin de ranger horizontalement les 2 widgets, donc il nous faut un `QHBoxLayout`, le `QLCDNumber` permet l'affichage de chiffres, on va l'utiliser pour afficher le nombre d'événements (donc le contenu de la variable `nbEvents`). On va utiliser le *slot* `recvData(QString)` qui recevra le texte à afficher.

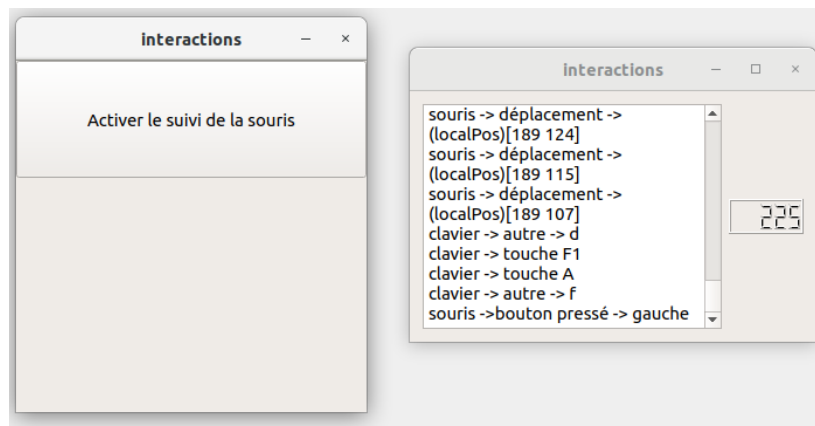


FIGURE 3.2 – 2 fenêtres : une pour l'interaction, une pour les informations

Quelques aides :

- pour afficher un nombre dans un `QLCDNumber` :  
`LCDnbEvents->display (nbEvents);`
- pour modifier TOUT le contenu d'un `QTextEdit` :  
`zoneTexte->setText ("blabla");`
- pour AJOUTER du contenu dans un `QTextEdit` :  
`zoneTexte->append ("blabla");`

- pour spécifier une taille minimale à un widget (la largeur) :  
`LCDnbEvents->setMinimumWidth (10);`
- idem pour une dimension maximale (la hauteur ici) :  
`LCDnbEvents->setMaximumHeight(30);`
- pour spécifier un affichage en décimal dans un `QLCDNumber` :  
`LCDnbEvents->setMode (QLCDNumber::Dec);`  
(on peut aussi demander un affichage hexadécimal, octal, binaire)

Il vous reste à :

- mettre en place le `HBoxLayout` (cf figure),
- incrémenter le `nbEvents` à chaque `Affichage::recvData` et `DEMANDER` (un signal) la mise à jour du `QTextEdit` (qui fera un `append`),
- déclarer un signal dans `mainWidget` qui permettra d'envoyer une chaîne `QString` d'information à `Affichage`, à chaque fois qu'un événement clavier ou souris est reçu par le widget `mainWidget`,

Il faut manipuler dans `MouseXXXEvent` des `QString`, attention, il faut convertir les type `QPointF` en `QString`, par exemple :

```
QString n;
n.setNum (event->x ());
s += n;
```

Les `event->button()` ne se traduisent pas, il faut faire un *switch* (« selon cas ») pour gérer la chaîne à envoyer :

```
switch (event->button())
{
case Qt::LeftButton:
s += "gauche";
break;
case Qt::RightButton:
s += "droite";
break;
case Qt::MiddleButton:
s += "milieu";
break;
default:
s += "non identifié";
}
```

### 3.3 Version 2 : du dessin

Si vous vous rappelez, on peut utiliser un événement `paintEvent()`. Cet événement gère le dessin d'un widget (lorsque celui-ci reçoit le message `update()`). Nous allons nous en servir pour modifier l'application précédente. Lorsqu'un bouton de la souris sera enfoncé, on affichera un point bleu à cette position. On réécrit donc l'événement :

```
1 void mainWidget::paintEvent(QPaintEvent *)
2 {
3     QPainter painter(this);
4
5     QPen pen;
6     pen.setWidth(5);
```

```

7         pen.setBrush(Qt::blue);
8
9         painter.setPen(pen);
10        painter.drawPoint (mousePoint);
11
12        qDebug() << "paint";
13    }

```

On déclare un objet `QPainter` sur le widget courant. Cet objet gèrera la surface et les propriétés du dessin. On crée un crayon `QPen` pour dessiner des objets géométriques. Ici il sera de 5 pixels de large et bleu. On affecte ce crayon au `QPainter` et on demande le dessin d'un point (`.drawPoint(mousePoint)`). Cette méthode prend un `QPointF` en paramètre, pour l'utiliser dans notre widget (`mainWidget`) on va ajouter cette variable dans `mainWidget.h` :

```
QPointF mousePoint;
```

Lorsqu'un `mousePressEvent()` est déclenché on stocke les coordonnées locales du pointeur de souris :

```
mousePoint = event->localPos (); //on stocke les coordonnées locales de la souris
update (); //on demande le rafraîchissement du widget, qui rappelle le paintEvent()
```

Et voilà, c'est tout. Les sources « dessin0 » contiennent tout le code pour que cela fonctionne.

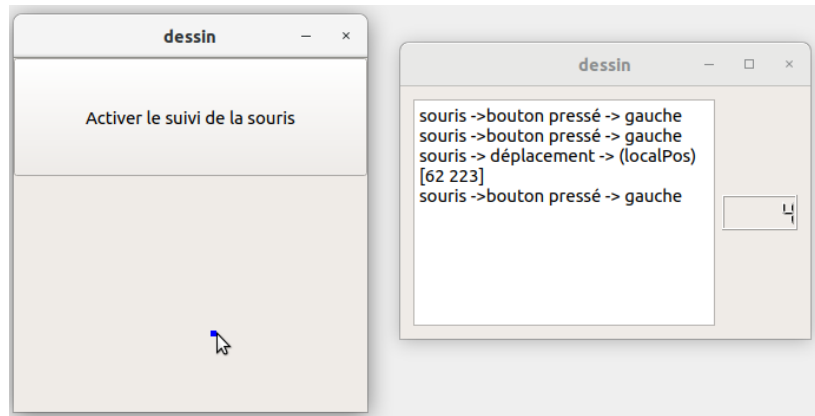


FIGURE 3.3 – Un point bleu sous le clic de souris

La cerise sur le gâteau du TP, que vous devez apporter à ces sources, est de conserver le tracé des points bleus au clic de souris ET de relier ces points par des lignes rouges (cf. fig 3.4). Il faut :

1. stocker les `QPointF` issus des `event->localPos` de la souris dans une liste :

- on déclare un `QVector<QPointF> mousePoint;` dans `mainWidget.h`,
- on ajoute les coordonnées au fur et à mesure :  
`mousePoint.append (event->localPos ());`

2. dans le `paintEvent` on trace comme avant les points bleus (mais tous) :

```
painter.drawPoints (mousePoint.data (), mousePoint.size ());
```

La fonction `.data()` de `QVector` permet de renvoyer directement un tableau `QPoint *` et c'est exactement ce que veut `.drawPoints`.

3. et on trace les lignes. C'est un peu plus compliqué, il faut créer un `QVector` de `QLineF` - les points sont en coordonnées décimales, y ajouter les lignes une par une, en récupérant des paires de `QPointF` dans la liste de points, puis passer à la fonction `.drawLines` cette liste de `QLineF`

Allez, je vous donne la partie compliquée mais faites l'effort de comprendre comment cela fonctionne.

```

1 QVector<QLineF> segments;
2 QPointF A, B;
3
4 for (QVector<QPointF>::iterator i = mousePoint.begin (); i < mousePoint.end
   () - 1; ++i)
5 {
6     A = *i;
7     B = *(i+1);
8
9     QLineF l;
10    l.setPoints(A, B);
11    segments.append(l);
12 }

```

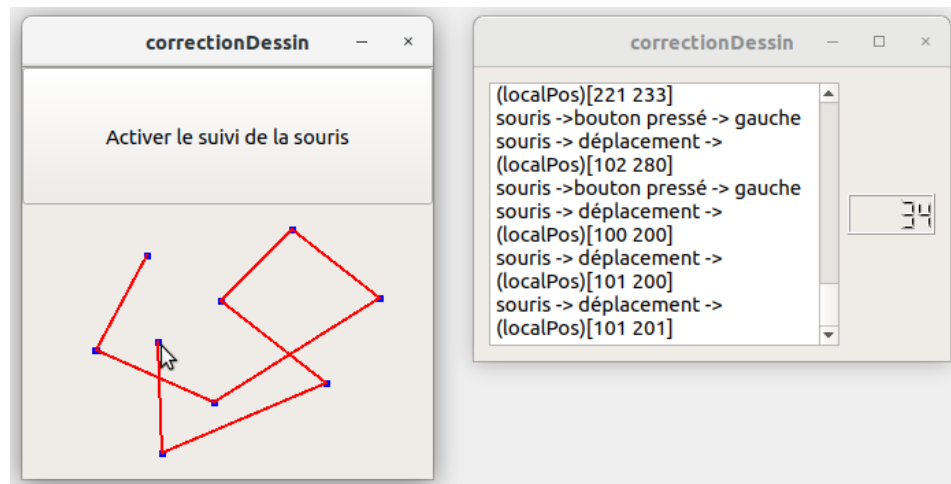


FIGURE 3.4 – C'est beau! (et c'est fini!)