

# TP Interaction gestuelle

Gilles Bailly

## Objectif

L'objectif de ce TP est double :

- D'une part, il vise à mieux appréhender les classes **QGesture** et **QGestureRecognizer** afin que vous puissiez développer vos propres interfaces gestuelles en s'appuyant sur l'infrastructure Qt.
- D'autre part, il vise à se familiariser avec le 1\$ recognizer.

## Ressources

Les classes Qt utiles pour ce TP sont :

- **QGesture** : <http://doc.qt.io/qt-5/qgesture.html>
- **QGestureEvent** : <http://doc.qt.io/qt-5/qgestureevent.html>
- **QGestureRecognizer** : <http://doc.qt.io/qt-5/qgesturerecognizer.html>

Les détails de l'algorithme du 1\$ recognizer sont disponibles ici :

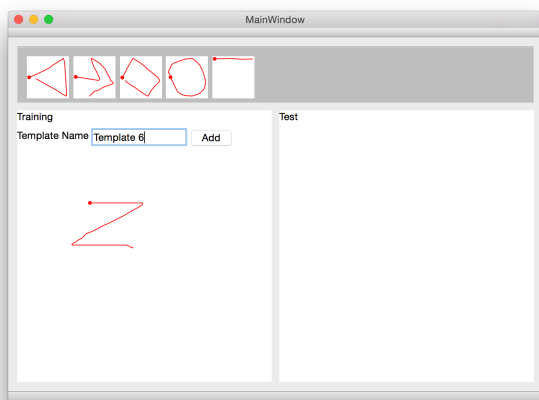
<http://faculty.washington.edu/wobbrock/pubs/uist-07.01.pdf>

Plusieurs implémentations sont disponibles ici :

<http://depts.washington.edu/aimgroup/proj/dollar/>

## Travail à réaliser

Vous devez réaliser une interface permettant à l'utilisateur de 1) créer ses propres gestes et 2) tester le 1\$ recognizer. La figure 1 montre à quoi pourra ressembler cette interface. Vous serez ensuite libre de vous appuyez sur cette interface pour implémenter davantage de fonctionnalités.



**Figure 1.** L'interface contenant une galerie de templates (haut), une zone pour définir ses propres gestes (gauche), une zone pour tester le système de reconnaissance (droite).

## Partie 1

### Étape 1 : créer le squelette de l'application

Créez une fenêtre qui hérite de **QMainWindow**. créez un **QWidget** qui servira de container pour les trois parties de l'interface. Attachez le à l'aide de la méthode **setCentralWidget()**. Vous utiliserez des layouts pour permettre un redimensionnement "intelligent" de l'interface. Vous vous aiderez des classes **QVBoxLayout** et **QHBoxLayout**.

- Pour *attacher* un layout à un **QWidget**, utilisez la méthode **QWidget::setLayout()** ;
- Pour *ajouter* un **QWidget** à un layout, utilisez la méthode **QLayout::addWidget()** ; Vous pouvez également ajouter un layout directement à un autre layout à l'aide de la méthode **QLayout::addLayout()**. Vous pourrez en particulier attachez le layout horizontal directement au layout vertical.

Pour tester la disposition et le redimensionnement de votre interface, vous utiliserez des **QLabel** ou des **QPushButton**. N'utilisez pas pour l'instant pas de **QWidget** car leur politique de redimensionnement n'est pas défini.

### Étape 2 : créer la galerie de templates

Pour créer une galerie de templates, vous utiliserez la classe **QScrollArea** qui permet de faire défiler horizontalement et verticalement le **QWidget** qu'il contient.

Vous ajouterez un **QWidget** au **QScrollArea** à l'aide de la méthode **QScrollArea::setWidget()**. Ce **QWidget** contiendra un **QHBoxLayout**. Appliquez à votre instance de **QScrollArea** la méthode **QScrollArea::setWidgetResizable(true)**. Les méthodes **setHorizontalScrollBarPolicy** et **setVerticalScrollBarPolicy** de **QScrollArea** permettent de définir les règles de défilement. Tester pour le moment votre galerie en insérant plusieurs boutons ou labels.

### Étape 3 : créer un classe qui affiche un geste

Pour préparer le contenu de la galerie de templates, créez une classe qui hérite de **QWidget** qui affiche un geste (stroke). Cette classe pourra s'appeler **StrokeDrawer** et aura plusieurs méthodes pour accéder au geste (**getStroke/setStroke**), le réinitialiser (**clear**) et l'afficher (**paintEvent**). Vous pouvez vous inspirer du TP précédent pour réaliser cette classe. Vous utiliserez la classe **QPolygonF** comme attribut de cette classe pour représenter le geste et la méthode **QPainter::drawPolyline** pour l'afficher.

Un **QWidget** peut indiquer sa taille « préférée » à l'aide de la méthode **QWidget::sizeHint()** et **QWidget::minimumSizeHint()**. C'est utile lorsque le widget est ajouté à un layout. Réimplémenter ces deux méthodes pour que la taille préférée de ce widget permette d'afficher la totalité du geste dans un carré. **Attention !!!** ces deux

méthodes sont « const ». Vous pourrez utiliser la méthode `QPolygonF::boundingRect()` et la méthode `qMax()` de `<qmath.h>` pour réimplémenter ces méthodes

Cette classe **StrokeDrawer** restera suffisamment **générique** pour être réutilisée ensuite pour la zone d'entraînement et la zone de test.

#### Étape 4 : créer le composant GestureTraining

Créer maintenant une classe permettant de créer un template. La classe pourra s'appeler **StrokeTraining** héritera de la classe **StrokeDrawer**. Cette classe écoutera les événements souris (comme vous l'avez fait dans les TPs précédents) et dessinera le geste correspondant. Ajouter quelques fonctionnalités à cette classe pour permettre, par exemple, à l'utilisateur de donner un nom au template à l'aide de la classe **QLineEdit** lorsqu'il est content du geste qu'il a effectué. Un bouton (**QPushButton**) permettra de valider et communiquer ce geste à votre fenêtre qui servira de contrôleur. Pour gagner du temps, disposez ces éléments directement dans le **QWidget** (sans utiliser de layout) à l'aide de la méthode `move()`.

#### Étape 5 : Utiliser la classe GestureTemplate 1/2

Téléchargez la classe **GestureTemplate** (.cpp, .h dans <http://www.gillesbailly.fr/qt/>) et ajoutez la à votre projet. Cette classe transforme le tracé de l'utilisateur en un template utilisable pour le reconnaître. La méthode `normalize()` effectue les trois premières étapes de l'algorithme décrit ici : <http://faculty.washington.edu/wobbrock/pubs/uist-07.01.pdf> (page 10). Cette méthode va re-échantillonner le geste, tourner le tracé pour que le point de départ soit le point le plus à gauche ; changer l'échelle et enfin le centrer par rapport à l'origine. Ceci facilitera la comparaison d'un nouveau geste avec la base des templates. Vous pouvez jeter un œil à l'algorithme et à l'implémentation.

#### Étape 6 : Utiliser la classe GestureTemplate 2/2

Ajoutez un `QVector<GestureTemplate>` comme attribut de la classe correspondant à votre fenêtre. Connectez maintenant, le composant **GestureTraining** et le vecteur de **GestureTemplate**. Pour cela, (1) vérifiez que le geste est correct (contient suffisamment de points) ; (2) normaliser le à l'aide `GestureTemplate::normalize()` avant de l'ajouter au vecteur.

Mettez ensuite à jour votre galerie lors de l'ajout d'un template. Utilisez la méthode `GestureTemplate::scaleToSquare()` pour obtenir une « vignette » de votre template avec une taille adéquate pour votre galerie et la méthode `QPolygonF::translate()` pour centrer votre geste dans la vignette.

#### Étape 7 (Optionel) : Sauvegarder les templates.

Implémentez une méthode permettant de sauvegarder les templates dans un fichier et une méthode qui charge ces templates depuis un fichier.

## Partie 2

Nous allons maintenant nous concentrer sur le composant **GestureTest** de notre interface en utilisant **Qt's Gesture framework**.

### Étape 8 QGesture : overview.

**QGesture** est la classe centrale du **Qt's Gesture Framework**. Elle contient un container pour les informations concernant le geste effectué par l'utilisateur. Cette classe sera étendue plus tard pour contenir les informations nécessaires pour le type de gestes que l'on souhaite reconnaître. Qt a implémenté trois classes qui héritent de **QGesture** : **QPanGesture**, **QPinchGesture** et **QSwipeGesture**. Regarder les différentes méthodes de ces quatre classes.

### Étape 9 : Créer le composant StrokeTest.

Avant de tester les gestes pour le 1\$ reconnaître qui nécessite l'implémentation de notre propre système de reconnaissance, nous allons d'abord nous familiariser avec l'utilisation des gestes qui sont déjà reconnus par Qt (Pan, Pinch et Swipe). Comme le geste Pinch, nécessite un dispositif multi-touch, nous allons utiliser **QSwipeGesture** (ou **QPanGesture** si vous le souhaitez).

Créer une classe **StrokeTest** qui hérite de **StrokeDrawer**. Dans le constructeur de **StrokeTest**, indiquez que ce **QWidget** s'abonne aux gestes « swipe » à l'aide de la méthode **QWidget::grabGesture()**.

### Étape 10 : récupérer les événements de type QSwipeGesture

Lorsque l'utilisateur fait un geste, un **QGestureEvent** est délivré au composant correspondant. Cet événement peut être récupéré en réimplémentant la méthode **QWidget::event()**. Cette méthode **event()** gère tous types d'événements comme les **QMouseEvent** ou les **QKeyEvent**. Pour cette raison, on ne s'intéressera qu'aux événements de type **QEvent::Gesture**. Pour la lisibilité du code, on implémentera une méthode :

**bool gestureEvent(QGestureEvent\*)** comme ci-dessous.

```
bool ImageWidget::event(QEvent *event)
{
    if (event->type() == QEvent::Gesture)
        return gestureEvent(static_cast<QGestureEvent*>(event)) ;
    return QWidget::event(event) ;
}
```

```
}  
.
```

Dans la méthode `gestureEvent(QGestureEvent*)`, il est possible de récupérer le geste (`QGesture`) associé à l'évènement à l'aide de la méthode `gesture()` de la classe `QGestureEvent`. Transformer ensuite ce `QGesture` en `QSwipeGesture` à l'aide :  
`static_cast<QSwipeGesture*>()`

**Attention !!** il faut bien vérifier que le geste est un `QSwipeEvent`. Pour cela vérifiez que le pointeur n'est pas null après le `static_casting`.

### Étape 11 : interpréter les événements de type `QSwipeGesture`

Récupérez l'état du geste avec la méthode `QGesture::state()` et afficher le dans la console. En particulier, le geste est terminé lorsque l'état est égal à `Qt::GestureFinished`. Indiquez dans un `qLabel` la commande qui doit être exécuté (ex : Previous et Next) en fonction de la direction du swipe (vous utiliserez les méthodes `horizontalDirection()` et `verticalDirection()`).

### Étape 12 : `QGestureRecognizer` : overview

Nous allons maintenant créer notre propre système de reconnaissance de gestes. Pour cela, nous allons créer dans les questions suivantes deux classes :

- **OneDollarRecognizer** qui héritera de `QGestureRecognizer`. Cette classe permettra de comparer le geste exécuté par l'utilisateur à la liste de templates que nous avons précédemment créé (partie 1).
- **OneDollarGesture** qui héritera de `QGesture` (comme `QSwipeGesture` hérite de `QGesture`). Cette classe permettra d'avoir des informations spécifiques au geste de l'utilisateur. En particulier, le template le plus proche ainsi qu'un score indiquant la similarité entre le geste et le template le plus proche.

### Étape 13 : Implémenter `OneDollarGesture`

Créer une classe **OneDollarGesture** qui hérite de `QGesture` avec trois attributs : le tracé du geste de l'utilisateur sous forme d'un `QPolygonF`, le **GestureTemplate** le plus proche et un score comme mesure de similarité entre le geste effectué par l'utilisateur et le template le plus proche. Implémenter également les méthodes permettant d'accéder (get/set) à ces attributs et une méthode pour réinitialiser les attributs (clear).

### Étape 14 : Implémenter `OneDollarRecognizer`

Pour créer un système de reconnaissance de gestes en Qt, il est nécessaire de créer une classe qui hérite de `QGestureRecognizer` et de ré-implémenter trois méthodes :

- **recognize()** qui est chargé de recevoir les événements en entrée (en l'occurrence les événements souris), de les filtrer et de déterminer si ces événements génèrent un geste qui est pris en charge par le système de reconnaissance. (voir question suivante)
- **create()** qui permet de créer des instances de gestes qui remplacent les **QGesture** par défaut. Dans notre cas, cette méthode créera un **OneDollarGesture**.
- **reset()** pour réinitialiser un geste, typiquement sur un événement de type Cancel. Dans notre cas, cette méthode appellera d'abord la méthode **clear()** de **OneDollarGesture** puis la méthode **reset()** de **QGestureRecognizer**.

Téléchargez la classe **OneDollarUtil** qui contient plusieurs méthodes utiles pour le 1\$ reconnaître. Créez une classe **OneDollarGestureRecognizer** qui hérite à la fois de **QGestureRecognizer** et **OneDollarUtil**.

### Étape 15 : Implémenter la méthode create()

Dans le corps de la méthode **create()**, créez et retournez un **OneDollarGesture\***

### Étape 16 : Implémenter la méthode reset()

Dans la méthode **reset()**, réinitialisez le **OneDollarGesture** avec la méthode **clear()**. Pour cela, utilisez la fonction `static_cast<OneDollarGesture*>` pour transformer un **Gesture\*** en **OneDollarGesture\***. Appelez ensuite la méthode **QGestureRecognizer::reset()**.

### Étape 17 : Implémenter la méthode recognize()

Pour accélérer l'implémentation, on vous donne une partie du corps de cette méthode. Pour cela, téléchargez le fichier `OneDollarRecognizer_recognize.txt` et remplacez les **TODO** par le contenu correspondant.

### Étape 18 : GestureTest (seconde version)

Il s'agit maintenant de mettre à jour **GestureTest** pour qu'il puisse utiliser le 1\$ reconnaître :

1. Dans le constructeur créez une instance de **OneDollarRecognizer**.
2. Enregistrez votre instance à l'aide de la méthode **QGestureRecognizer::registerRecognizer()**.
3. Abonnez votre widget à recevoir événements « 1\$ dollar » à l'aide de la méthode **QWidget::grabGesture()**. Le type de geste (**Qt::GestureType**) nécessaire en paramètre est la valeur retournée par **QGestureRecognizer::register()**.
4. Modifiez votre méthode **GestureEvent()** pour traiter les **QGestureEvent**. En particulier, lorsque l'état est égal à **Qt::GestureUpdated** (pour dessiner la trace à l'écran) et **Qt::GestureFinished** pour récupérer le résultat du 1\$ reconnaître.

Modifiez également le code dans votre fenêtre et de **GestureTest** pour informer **OneDollarGestureRecognizer** de la liste des templates à utiliser.

## Étape 19 : Etape optionnelle

Ajouter toutes sortes de décoration qui pourrait améliorer votre application. L'application la plus réussie sera candidate pour la réalisation d'une vidéo qui sera postée sur le site de l'UE.