TP3: Interaction gestuelle

Objectif

L'objectif de ce TP est triple:

- Mieux appréhender les **QLayout**
- Mieux appréhender les classes **QGesture** et **QGestureRecognizer** afin que vous puissiez développer vos propres interfaces gestuelles en s'appuyant sur l'infrastructure Qt.
- Se familiariser avec le 1$ recognizer que vous avez vu en cours la semaine dernière.

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

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.

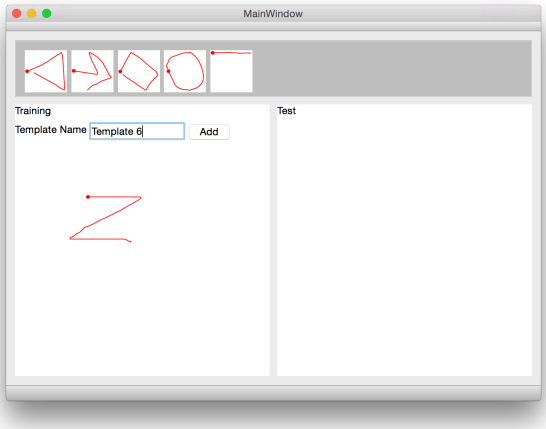

Figure 1. L'interface contenant une galerie de templates (haut), une zone pour définir ses propres gestes (gauche), une zone pour tester le système de reconnaissance (droite).

Partie 1

Étape 1 : créer le squelette de l'application

La classe MainWindow (fournie) hérite de **QMainWindow**. Un container (**QWidget)** contiendra les trois parties de l'interface. Vous lui ajouterez des layouts pour permettre un redimensionnement "intelligent" de l'interface à l'aide de **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 attacher 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 (haut de la figure 1), vous utiliserez la classe **QScrollArea 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 le **QScrollArea** ::**setWidget**(). Ce **QWidget** contiendra un **QHBoxLayout**. Appliquer 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

Complétez la class GestureDrawer qui permet d'afficher un geste (stroke). Elle a plusieurs méthodes pour accéder au geste (getStroke/setStroke), le réinitialiser (clear) et l'afficher (paintEvent). Complétez la méthode drawWidget à l'aide de 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. Vous pourrez utiliser la méthode **QPolygonF** ::**boundingRect**() et la méthode qMax() de **<qmath.h>** pour réimplémenter ces méthodes

Cette clase est utilisée ensuite pour la zone d'entrainement et la zone de test.

Étape 4 : Le composant GestureTraining

La classe **GestureTraining** (hérite de GestureDrawer) permet de créer un template. Cette classe écoutera les évènements souris (comme vous l'avez fait dans le TP2) et dessinera le geste correspondant. Complétez cette classe pour que le bouton (**QPushButton**) valide et communique le geste à votre fenêtre qui servira de contrôleur.

Étape 5 : Utiliser la classe GestureTemplate 1/2

La classe **GestureTemplate** transforme le tracé de l'utilisateur en un template utilisable pour le 1$ recognizer. La méthode **normalize**() effectue les quatre premières étapes de l'algorithme décrit ici : http://faculty.washington.edu/wobbrock/pubs/uist-07.01.pdf (page 10). Cette méthode va

- re-echantilloner le geste ;
- tourner le tracé pour que le point de départ soit le point le plus à gauche ;
- changer l'échelle
- 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.

Regardez la classe **GestureTemplate**. Implémentez la méthode **normalize** en vous appuyant sur les différentes méthodes de la classe GestureTemplate.

Étape 6 : Utiliser la classe GestureTemplate 2/2

Votre MainWindow contient un **QVector<GestureTemplate>**. 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 : Le composant GestureTest : récupérer les évenements de type gesture

La classe **GestureTest** hérite de **GestureDrawer**. 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'évènements comme les **QMouseEvent** ou les **QKeyEvent**. Pour cette raison, on ne s'intéressera qu'aux évènements 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**.

Étape 11 (si vous avez un touchpad): interpréter les événements de type QSwipeGesture

Transformer le **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.

Il faut aussi dans le constructeur de **GestureTest**, indiquer que ce **QWidget** s'abonne aux gestes « swipe » à l'aide de la méthode **QWidget** ::**grabGesture**().

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 exécutée (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 utiliser 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 créée (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 : OneDollarGesture

La classe **OneDollarGesture** hérite de **QGesture** avec quatre attributs :

- le tracé du geste de l'utilisateur sous forme d'un **QPolygonF**,
- le **GestureTemplate** le plus proche
- un score comme mesure de similarité entre le geste effectué par l'utilisateur et le template le plus proche.
- La rotation pour « matcher » au mieux au template.

Regardez cette classe

Étape 14 : La classe 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 évènements 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**.

La classe **OneDollarUtil** contient plusieurs méthodes utiles pour le 1$ recognizer. C'est pour cela qu'elle hérite également de **OneDollarUtil**.

Étape 15 : Implémenter la méthode create()

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

Étape 16 : Implémenter la méthode reset()

Dans la méthode **reset**(), réinitiliasisez 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()

Complétez la méthode recognize. En particulier, vous appelerez la méthode **recognizeStroke()** de la classe **OneDollarUtil**.

Étape 17 : Implémenter recognizeStroke()

La comparaison entre un nouveau geste et un template consiste à calculer la somme des distances entre les points pris deux à deux des deux gestes. Cette comparaison est faite en déterminant l'angle de rotation entre les deux figures qui minimise cette distance. Cette recherche de l'angle optimal est obtenue par l'utilisation de la technique Golden Section Search pour minimiser le nombre d'itérations vers la solution optimale (10 itérations au maximum avec cette technique).

Ecriver la méthode **recognizeStroke**() de la classe OneDollarUtile en vous appuyant sur l'algorithme correspondant.

```
RECOGNIZE(points, templates)
1  b ← +∞
2  foreach template T in templates do
3     d ← DISTANCE-AT-BEST-ANGLE(points, T, -θ, θ, θΔ)
4     if d < b then
5        b ← d
6        T' ← T
7  score ← 1 – b / 0.5√(size² + size²)
8  return ⟨T', score⟩
```

Étape 18 : Implémenter GestureTest

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

1. Dans le constructeur créé une instance de OneDollarRecognizer.

2. Enregistrer votre instance à l'aide de la méthode **QGestureRecognizer ::registerRecognizer**().
3. Abonner 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**. Récupérez l'état du geste avec la méthode **QGesture ::state**(). Lorsque sont état est égal à **Qt ::GestureUpdated,** dessiner la trace à l'écran. Lorsque l'état est égal à **Qt ::GestureFinished,** récupérer le résultat du 1$ recognizer.

Modifiez également le code dans MainWindow et de GestureTest afin d'informer **OneDollarGestureRecognizer** de la liste des templates à utiliser.

Étape 19 : Etape optionelle

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.