Module RAD
Bézier & B-Spline (1/2)

Nom : Laxenaire
Prénom : Flavian
2bts snir

Pour ces travaux, vous devez utiliser une machine équipée du SDK Qt version 1.6.x ou supérieure ou à minima du Qt Framework version 5.3.x ou supérieure ; l'usage de l'EDI Qt Creator est cependant recommandé.

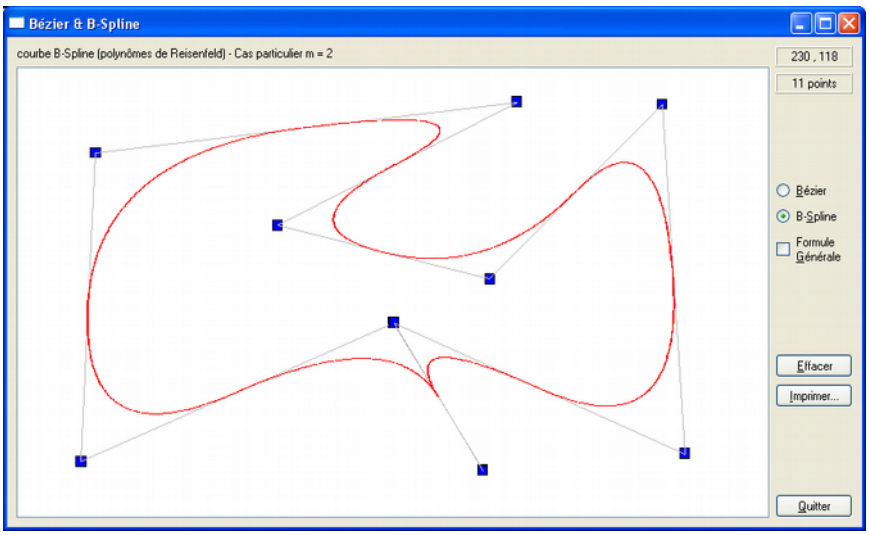
ATTENTION : vérifiez que Qt Creator est configuré en mode « multiple inheritance » pour la génération de classes...

1. Cahier des Charges

La finalité de ces travaux pratiques est la réalisation d'une application graphique permettant à l'utilisateur de construire des courbes gauches à partir d'une enveloppe segmentée.

L'application doit assurer la mise en oeuvre des courbes de Bézier ainsi que celle des B-Splines uniformes ; des annexes vous fourniront la théorie mathématique nécessaire au tracé des courbes.

Exemple de produit fini →



Le développement doit exclusivement être réalisé en programmation C++ avec la librairie Qt.
La réalisation va se décomposer en grandes 2 parties : la première est relative à la construction de l'IHM et au tracé des segments de l'enveloppe, la deuxième se charge des calculs mathématiques et du dessin des courbes.

Comme d'habitude, il est fortement conseillé de lire complètement le sujet avant de commencer à traiter les questions…
Pour toute référence à une classe Qt ou à ses membres, consulter sérieusement la documentation de la classe afin d'appréhender les fonctionnalités qu'elle est capable de fournir.

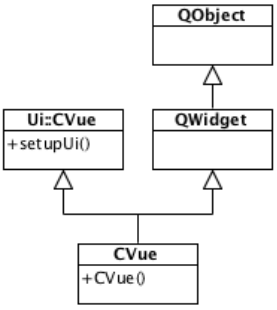
2. Construction de l'IHM

Utilisez Qt Creator pour créer un nouveau projet *Desktop* de type *Application Qt avec widgets* que vous nommerez `Courbes`.
Indiquez comme nom de classe `CVue` et comme classe parent `QWidget`, autorisez la génération de l'interface graphique.

Q1. Quel squelette de projet obtenez-vous (liste des fichiers créés) ?

les fichiers créés sont : courbes.pro ; cvue.h ; main.cpp ; cvue.ui ; cvue.cpp

Q2. Vous pouvez tester le programme… L'architecture de l'application est-elle conforme au diagramme de classes ci-contre ? Oui, le diagramme de classes est conforme. .....

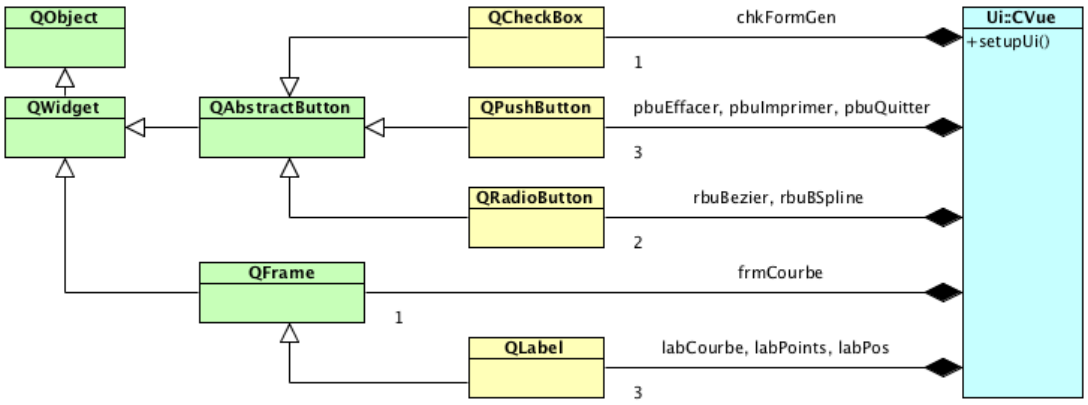


Q3. Quelle instruction pouvez-vous ajouter avant la projection de la fenêtre pour lui donner le titre « Bézier & B-Spline » ?

```
CVue w;
w.setWindowTitle("Bézier & B-Spline");
w.show();
```

.....

Utilisez le *designer* pour construire l'IHM de l'application en vous aidant du *screen shot* de la page précédente et du diagramme de classes ci-dessous (respectez les noms des widgets).



`frmCourbe` représente la zone de dessin et `labCourbe` le texte d'information situé juste au-dessus ; tous les autres éléments sont rangés dans la colonne à droite.

Q4. Combien de widgets enfants la vue possède-t-elle ? La vue possède 10 widgets enfants .....

Q5. Quelle propriété(s) de quel(s) widget(s) devez-vous modifier dans le *designer* pour sélectionner le mode « Bézier » par défaut ?

Il faut cocher la case "checked" du widget QRadioButton "rbuBezier" dans la propriété de la classe QAbstractButton. .....

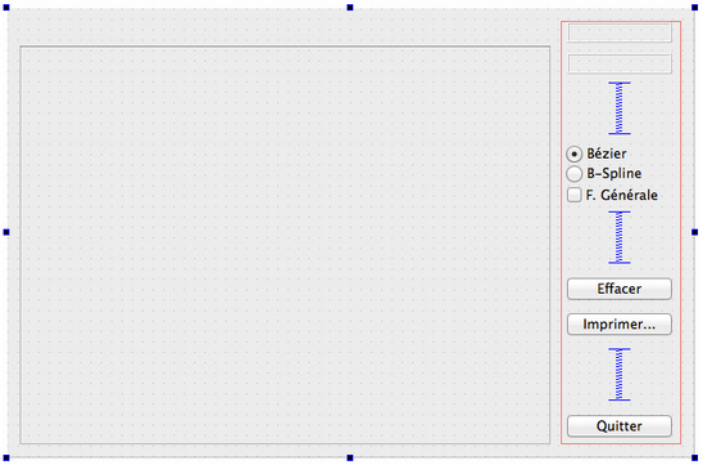
.....

Occupez-vous maintenant des arrangements (*layout*).

Vous devriez arriver à une solution correcte avec quelques contraintes et en utilisant judicieusement les éléments suivants :

- 3 x Vertical Spacer
- 1 x Vertical Layout
- 1 x Grid Layout

Ce dernier est bien sûr à appliquer à la fenêtre complète…



Pour en terminer avec le *designer*, mettez en place une connexion assurant l'issue du programme →

Émetteur	Signal	Receveur	Slot
pbuQuitter	clicked()	CVue	close()

Testez le programme, vérifiez notamment les arrangements lors de la distorsion de la fenêtre et la possibilité de quitter proprement l'application.

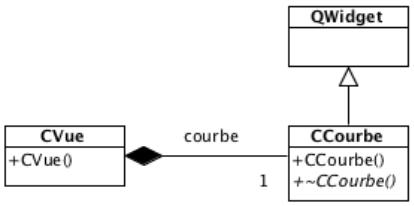
3. Zone de dessin

La zone de dessin, matérialisée par le composant `frmCourbe`, doit subir de nombreux traitements pour mener à bien le cahier des charges ; il est donc souhaitable de la spécialiser en créant une nouvelle classe `CCourbe`, dérivée uniquement de `QWidget`.

Utilisez la commande *Fichier | Nouveau fichier ou projet…* de Qt Creator pour ajouter au projet une nouvelle classe C++ nommée `CCourbe` et dérivée de `QWidget`.

Vous obtenez normalement un nouveau point hash et un nouveau point c'est pépé qui sont automatiquement pris en compte par le fichier projet.

L'objet de classe `CVue` doit construire une instance de `CCourbe` et faire en sorte que celle-ci occupe en permanence l'aire de la zone de dessin.



Q6. Quelle méthode particulière devez-vous ajouter à la classe `CCourbe` pour être conforme au diagramme ci-contre ?

Il faut ajouter un destructeur : ~CCourbe();

Afin de satisfaire la composition apparaissant sur le diagramme, ajoutez à `CVue` une annonce de classe `CCourbe` et un membre privé `courbe` de type `CCourbe*`.

Incluez ensuite l'interface de la classe `CCourbe` dans l'implémentation de `CVue`.

Q7. Que faut-il ajouter dans le constructeur de `CVue` pour créer l'objet `courbe` avec pour parent `frmCourbe` ?

courbe = new CCourbe(ui->frmCourbe);

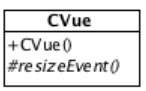
L'objet `courbe` fait ainsi partie de l'arborescence des widgets de l'UI. Il n'est donc pas nécessaire de se soucier de sa destruction, Qt s'en chargera…

Pour que `courbe` occupe en permanence l'aire totale de `frmCourbe`, il est nécessaire de détecter les variations de dimensions de la vue et, le cas échéant, de calquer les dimensions de `courbe` sur celles de `frmCourbe`.

Ajoutez à `CVue` une surcharge de la méthode protégée `resizeEvent()` héritée de `QWidget`.

Implémentez ensuite cette surcharge, une instruction suffit :

```
courbe->resize( frmCourbe->width(), frmCourbe->height() ) ;
```



Puisque vous avez étudié la documentation de `QWidget`, vous avez sans doute remarqué que cette classe dispose d'autres intercepteurs d'événements ; notamment en ce qui concerne la souris…

Ces intercepteurs vont nous être utiles pour gérer le tracé de l'enveloppe segmentée de nos courbes.

4. Gestion de la position de la souris

L'application, et plus particulièrement la zone de dessin, va nécessiter la réception en continu de la position de la souris (*mouse tracking*). Pour limiter la quantité d'événements à traiter, cette option n'est pas activée par défaut.

Activez cette option en ajoutant l'instruction suivante dans le constructeur de `CCourbe` :

```
setMouseTracking( true ) ;
```

Quelles que soient les dimensions réelles (en pixels) de la zone de dessin, nous allons faire en sorte que l'utilisateur travaille toujours sur une vue logique de taille 1000 x 1000. Les définitions suivantes, ajoutées à *ccourbe.h*, vont assurer par la suite le changement de repère :

```
1 #define LARGEURVUE  1000
2 #define HAUTEURVUE  1000
3 #define LX(X)       ( (int)( (double)LARGEURVUE / width() * (X) ) )
4 #define LY(Y)       ( (int)( (double)HAUTEURVUE / height() * (Y) ) )
```

Il s'agit de simples règles proportionnelles qui donnent les coordonnées d'un point dans la vue logique en fonction de ceux dans la vue physique.

`CCourbe` doit être capable d'informer l'IHM de tout changement de position de la souris, et accessoirement plus tard du nombre de points de construction, afin de permettre la mise à jour des widgets `labPos` et `labPoints`.

Quoi de mieux pour cela qu'une connexion signal-slot ?

Dotez la classe `CCourbe` d'un signal de prototype : `void posChange(QPoint, int ) ;`
Ajoutez à `CVue` un slot privé de prototype : `void majInfo(QPoint, int ) ;`

Analysez le code ci-contre destiné à implémenter le slot →

```
1 void CVue::majInfo(QPoint point, int nbPoints )
2 {
3     QString s ;
4     s = QString("%1 , %2").arg( point.x() ).arg( point.y() ) ;
5     labPos->setText( s ) ;
6     s = QString("%1 point").arg( nbPoints ) ;
7     if ( nbPoints ) s += "s" ;
8     labPoints->setText( s ) ;
9 }
```

Q8. Quelle instruction devez-vous placer dans le constructeur de `CVue` pour activer la connexion ?

connect(courbe, SIGNAL(posChange(QPoint,int)), this, SLOT(majInfo(QPoint,int))) ;

Q9. Quel intercepteur devez-vous surcharger dans `CCourbe` pour être informé en permanence de la position de la souris sur l'aire de dessin ?

Il faut surchager : void QWidget::mouseMoveEvent(QMouseEvent *event);

Q10. Complétez l'implémentation de cette surcharge →

```
1 void CCourbe::mouseMoveEvent(QMouseEvent *event)
2 {
3     QPoint point ;
4     point.setX( LX( event->pos().x() ) ) ;
5     point.setY( LY( event->pos().y() ) ) ;
6     emit posChange( point, 0 ) ;
7 }
```

Testez le programme. Les coordonnées de la souris doivent être affichées en permanence sur l'IHM, ils doivent varier entre 0 et 999 quelles que soient les dimensions de la fenêtre.

5. Points de construction des courbes

Les courbes à construire le sont à partir d'une enveloppe segmentée définie par une liste de points.

En début de construction, la liste est vide. Chaque appui sur le bouton gauche de la souris doit entraîner le stockage dans la liste des coordonnées logiques de la position du pointeur souris (à condition que celui-ci soit bien évidemment dans la zone de dessin). Le dernier point est déterminé par un appui sur le bouton droit.

Qt fournit des modèles de classes conteneur et itérateur parfaitement adaptés à notre besoin. Notre liste de points peut facilement être gérée par une instance de classe… `QList<QPoint>`.

Ajoutez à `CCourbe` les membres privés :

```
QList<QPoint>           points ;
QList<QPoint>::iterator itPoint ;
```

Le programme est régi par deux modes de fonctionnement : le mode par défaut « Construction » tant que le dernier point de la liste n'a pas été spécifié, et un mode « Modification » lorsque la liste est complète (ce mode permettra d'envisager la reprise des points d'une courbe déjà construite).

Ajoutez à `CCourbe` l'énumération publique :

```
enum Mode { Construction, Modification } ;
```

et le membre privé :

```
Mode mode ;
```

Q11. Que faut-il ajouter dans le constructeur de `CCourbe` pour initialiser correctement ce dernier membre ?

mode = Construction;

Mettez en place dans la classe `CCourbe` les surdéfinitions nécessaires à la détection des appuis et relâchements des boutons de la souris (`void mouse.....Event(QMouseEvent* event )`).

Dans l'implémentation de la méthode de détection des appuis, récupérez la position de la souris dans une instance locale de classe `QPoint` de la même manière qu'en Q10.

Chaque point sera matérialisé graphiquement par une « poignée » de 10 x 10 unités logiques.

Ajoutez à `CCourbe` une méthode privée de prototype : `QRect poignee(const QPoint& point ) ;`

Q12. Consultez la documentation de la classe `QRect` et complétez l'implémentation de la méthode →

```
1  QRect CCourbe::poignee(const QPoint& point )
2  {
3      int size = 10 ;
4      int x = point.x() - LX( size / 2 ) ;
5      int y = point.y() - LY( size / 2 ) ;
6      return QRect( x, y, size, size ) ;
7  }
```

Pour le confort visuel de l'utilisateur, les points de construction vont être reliés par des segments en traits fins pointillés. Cela suppose de toujours avoir à disposition le dernier point validé par un clic gauche, mais aussi le point courant du pointeur souris dont le dernier segment doit suivre les déplacements (comme une sorte de ligne élastique entre le dernier point et la souris…).

Récapitulons le traitement à effectuer lors des mouvements et des clics souris sur l'aire de dessin :

- en mode construction : chaque clic gauche place un point sur le dessin et crée un point supplémentaire qui suit les mouvements de la souris ; un clic droit fixe le dernier point et bascule en mode modification.
- en mode modification : un appui gauche sur un point existant attache celui-ci aux mouvements de la souris, le point est fixé à la position du relâchement du bouton gauche.

  Le retour en mode construction est fait par action sur le bouton « Effacer » de l'IHM qui purge la liste de points enregistrée.

Le code de la procédure d'interception des appuis sur les boutons de la souris, après récupération des coordonnées logiques, devient donc le suivant :

```
1     if ( mode == Construction ) {
2         if ( ! points.isEmpty() )points.removeLast() ;
3         points << point ;
4         if ( event->button() == Qt::LeftButton ) {
5             points << point ;
6         }
7         if ( event->button() == Qt::RightButton ) {
8             itPoint = points.end() ;
9             mode = Modification ;
10        }
11    }
12    if ( mode == Modification ) {
13        if ( event->button() == Qt::LeftButton ) {
14            QList<QPoint>::iterator     it ;
15            for ( it = points.begin() ; it != points.end() ; ++it ) {
16                if ( poignee( *it ).contains(point) ) itPoint = it ;
17            }
18        }
19    }
20    update() ;
```

Mode construction : **ligne 2** : supprime le point provisoire précédent (sauf en cas de liste vide)

ligne 3 : ajoute le point courant de l'appui dans la liste

ligne 4 à 6 : ce n'est pas le dernier point, alors ajout d'un nouveau point provisoire

ligne 7 à 10 : c'est le dernier point, fixe l'itérateur et bascule en mode modification.

Mode modification : **ligne 13 à18** : si c'est un appui gauche, parcours de la liste de points pour trouver celui à reprendre ; le cas échéant, mémorise le point repris dans l'itérateur membre de la classe.

ligne 20 : force à redessiner la scène pour prendre en compte les modifications.

Q13. Quel est le prototype exact de la méthode `contains()` utilisée ligne 16 ?

bool QRect::contains(const QPoint &point, bool proper = false) const

Notez lignes 8 et 15 que la méthode `QList::end()` retourne un itérateur sur un élément fictif après la fin de liste. Fixer `itPoint` sur cette valeur signifie donc qu'aucun point n'est en cours de reprise ; c'est donc ce qu'il faut faire lors du relâchement du bouton gauche en mode modification…

Q14. En conséquence, quelle(s) instruction(s) devez-vous placer dans l'intercepteur de détection des relâchements des boutons de la souris ?

```
QPoint point;
point.setX(LX(event->pos().x()));
point.setY(LX(event->pos().y()));
```

Le code de la procédure d'interception des mouvements de la souris, après récupération des coordonnées logiques, doit être complété tel que ci-contre →

Les lignes 9 et 10 mettent à jour les coordonnées du point en cours de reprise.

```
1     if ( points.isEmpty() ) return ;
2
3     if ( mode == Construction ) {
4         points.removeLast() ;
5         points << point ;
6     }
7     if ( mode == Modification ) {
8         if ( itPoint == points.end() ) return ;
9         (*itPoint).setX( point.x() ) ;
10        (*itPoint).setY( point.y() ) ;
11    }
12    update() ;
```

Q15. Comment devez-vous modifier l'instruction ci-dessous pour une mise à jour correcte du nombre de points ?

`emit posChange(` point, points.count() `) ;`

6. Tracé des poignées et segments

Toute modification effectuée sur la liste de points va obliger à repeindre en tout ou partie la zone de dessin grâce à la méthode `update()` héritée de la classe `QWidget`.

Nous allons maintenant récupérer les événements qui demandent à repeindre notre widget de manière à mettre ne place nos propres instructions de dessin.

☑ Ajoutez à la classe `CCourbe` une surcharge de la méthode `paintEvent()` et une méthode privée de prototype `void dessine(QPainter& paint, bool final = false ) ;`

☑ Implémentez `paintEvent()` →

```
1  void CCourbe::paintEvent(QPaintEvent* event )
2  {
3      Q_UNUSED(event) ;
4
5      QPainter paint( this ) ;
6      dessine( paint ) ;
7  }
```

Le regroupement de toutes les instructions de dessin dans la fonction indépendante `dessine()` permettra par la suite de la réutiliser directement sur un autre *device* (par exemple une imprimante…). L'argument `final` sera d'ailleurs utilisé pour dessiner la courbe seule sans l'enveloppe segmentée lors des impressions.

☑ Consultez la documentation de `QPainter` et renseignez la méthode de dessin comme ci-dessous :

```
1      paint.save() ;
2      paint.setWindow(0, 0, LARGEURVUE, HAUTEURVUE ) ;
3
4      QList<QPoint>::iterator it ;
5      QPoint  pt ;
6
7      if ( !final ) for ( it = points.begin() ; it != points.end() ; ++it ) {
8          paint.setPen( Qt::SolidLine ) ;
9          paint.setBrush( it == itPoint ? Qt::red : Qt::blue ) ;
10         paint.drawRect( .............. ) ;
11         paint.setPen( Qt::DotLine ) ;
12         paint.setPen( Qt::lightGray ) ;
13         if ( ! pt.isNull() )      paint.drawLine(pt, *it ) ;
14         pt = *it ;
15     }
16
17     paint.restore() ;
```

Q16. Que devez-vous écrire ligne 10 pour dessiner la poignée du point donné par l'itérateur `it` ?

`paint.drawRect(` poignee(pt) ..... `) ;`

☑ Testez le programme ; la construction et le tracé de l'enveloppe et des poignées doivent fonctionner !

☑ Ajoutez maintenant le slot public ci-contre →

Cette méthode réinitialise la construction de l'enveloppe segmentée.

```
1  void CCourbe::raz()
2  {
3      points.clear() ;
4      mode = Construction ;
5      update() ;
6      emit posChange(QPoint(0,0), 0 ) ;
7  }
```

Q17. Quelle instruction faut-il ajouter dans le constructeur de `CVue` pour que le code de la méthode précédente soit exécuté lors d'une action sur le bouton « Effacer » de l'IHM ?

connect(ui->pbuEffacer, SIGNAL(clicked()), courbe, SLOT(raz())) ; .....

.....

☑ Testez à nouveau le programme ; le bouton « Effacer » doit être opérationnel.

7. Mise à jour de la ligne d'information

La ligne d'information en haut à gauche de la fenêtre va servir à indiquer le type de courbe en cours. Son contenu dépend de l'état des composants `rbuBezier`, `rbuBSpline` et `chkFormGen` :

Bézier B-Spline Formule Générale	« courbe de Bézier - Formule générale »
Bézier B-Spline Formule Générale	« courbe de Bézier - Méthode des barycentres (sur 4 points max.) »
Bézier B-Spline Formule Générale	« courbe B-Spline (polynômes de Reisenfeld) - Formule générale m = 3 »
Bézier B-Spline Formule Générale	« courbe B-Spline (polynômes de Reisenfeld) - Cas particulier m = 2 »

L'objet `CCourbe` doit maintenir en interne cette configuration, par exemple au moyen de 2 attributs booléens `isBezier` et `isFormGen` initialisés dans le constructeur.

Il doit aussi disposer d'une méthode publique du style `changeTypeCourbe(bool bezier, bool formGen )` afin que l'IHM puisse lui transmettre la configuration courante.

Ajoutez à `CCourbe` les membres `isBezier`, `isFormGen` et `changeTypeCourbe()`.
Implémentez ce dernier de manière à affecter les membres privés avec les valeurs des arguments.

Pour que l'IHM soit en mesure de modifier le texte d'information, `CCourbe` doit aussi proposer dans son interface publique une méthode du genre `QString typeCourbe()` susceptible de fournir une des 4 chaines de caractères correspondant au type courant maintenu par `isBezier` et `isFormGen`.

Implémentez cette nouvelle méthode en respectant le modèle ci-contre →

```
1  QString CCourbe::typeCourbe()
2  {
3      QString s ;
4      if ( isBezier ) {
5          // TODO
6      }
7      else {
8          // TODO
9      }
10     return s ;
11 }
```

Que nous manque-t-il ?

Coté IHM, il faut mettre en place un slot d'interception des actions de l'utilisateur sur le *checkbox* et les *radiobutton*. Comme les 2 boutons radio sont exclusifs, il suffit d'en tester un seul !

Enrichissez la classe `CVue` d'un slot privé nommé aussi `changeTypeCourbe()` et qui contiendra les instructions :

```
courbe->changeTypeCourbe( rbuBezier->isChecked(), chkFormGen->isChecked() ) ;
labCourbe->setText( courbe->typeCourbe() ) ;
```

Q18. Quel signal des éléments `rbuBezier` et `chkFormGen` devez-vous connecter sur ce slot ?

J'ai fais deux connections pour qu'il soit plus réactif:
connect(ui->rbuBezier, SIGNAL(toggled(bool)), this, SLOT(changeTypeCourbe())) ;
connect(ui->chkFormGen, SIGNAL(toggled(bool)), this, SLOT(changeTypeCourbe())) ;

Q19. Pour finir, que faut-il ajouter au constructeur de `CVue` pour initialiser correctement l'IHM ?

```
ui->labPoints->setText("0 point") ;
ui->labPos->setText("0 , 0") ;
ui->labCourbe->setText(courbe->typeCourbe());
```

Vérifiez la conformité de votre projet avec le diagramme de classes ci-contre.

Prenez 5 minutes pour tester votre superbe IHM de manière exhaustive.

à suivre…