4 - Les objets graphiques 2D en *Qt*

Plan du chapitre:

1- Généralités

Le graphisme numérique 2D est la manipulation numérique d'objets en deux dimensions, tels que que des f**ormes géométriques** (vectorielles) en 2D, des **textes** et des **images** « *raster* » (***pixmaps***). Il est utilisé dans les applications qui ont été développées à l'origine sur des technologies traditionnelles d'impression et de dessin, tels que la typographie, la cartographie, le dessin technique, la publicité mais aussi dans des applications de jeu vidéo 2D.

Qt offre 3 différentes API pour le graphisme 2D : *QPainter*/*QWidget*, *QGraphicsScene*/*QGraphicsView* et *QtQuick Scene Graph* :

- ***QPainter*** est une API de bas niveau pour le tracé (*drawing*) d'éléments graphiques vectoriels (lignes, rectangles, ellipses, arcs, polygones,...), de texte et d'images sur différentes surfaces (instances de sous-classes de *QPaintDevice*), notamment des instances de *QWidget*. Cette API est plus adaptée pour tracer des objets graphiques passifs (non interactifs). *QPainter* utilise par défaut le système de coordonnées de l'écran (pixels).
- Le ***framework « Graphics View »***, ainsi appelé par *Qt*, a deux classes principales : *QGraphicsScene* et *QGraphicsView*. La classe *QGraphicsScene* doit habituellement être dérivée dans une classe fille adaptée à l'application. Elle a pour but de contenir des objets (*items*) graphiques « intelligents » (sous-classes de *QGraphicsItem*) : lignes, rectangles, ellipses, textes, *pixmaps*,... qui représentent les objets du « monde réel » (de notre application). Ces *items* graphiques sont interactifs : ils ont toutes les méthodes pour réagir aux « évènements utilisateurs » (clic, « *drag & drop* »,...). On attribue en général à cette scène le système de coordonnées du « monde réel » (mètres, microns, années-lumière,... selon le domaine concerné). La classe *QGraphicsView* (sous-classe de *QWidget*) a pour rôle d'afficher la scène. Elle peut changer l'échelle, « *scroller* », faire une rotation de la scène. Cette vue a, par défaut, le système de coordonnées de l'écran (pixels). Entre scène et vue il y a un lien de type **Modèle-Vue** (cf. §4).

- ***QtQuick*** est la plus récente API graphique de *Qt*. Elle est adaptée à un développement rapide d'applications. Elle n'utilise plus C++ mais le **langage QML** pour la description des objets graphiques (inspirée de *CSS* et *JSON*) et ***JavaScript*** pour le comportement des objets. On sort ici complètement du champ du C++.

2- L'API *QPainter*

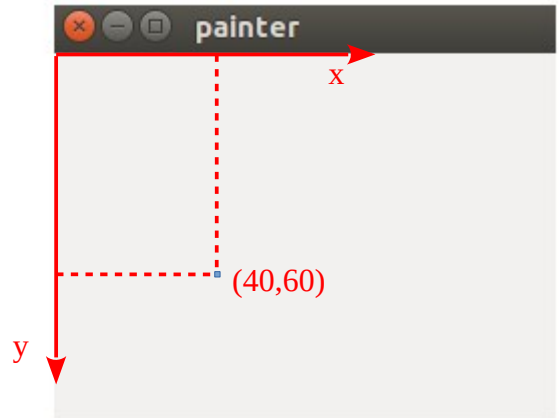
a) Paramétrage

QPainter a besoin, pour tracer les éléments graphiques, d'avoir configuré préalablement plusieurs caractéristiques de base, principalement :

- le **crayon** (classe *QPen*) : définit la couleur, l'épaisseur, le type (plein, pointillés,...) des traits (lignes, contours) (et aussi la couleur des textes). Il est modifié par la méthode *setPen()* qui prend un argument de type *QPen*, ex : `setPen(QPen(Qt::red,2));` La classe *QPainter* initialise un *QPen* par défaut noir, de 1 pixel d'épaisseur,
- la **brosse** (classe *QBrush*) : définit la couleur, le style (plein, hachuré, transparent = `Qt::NoBrush`,...) du remplissage des figures géométriques (rectangles, polygones, ellipses,...). Elle est modifiée par la méthode *setBrush()* qui prend un argument de type *QBrush*, ex : `setBrush(QBrush(Qt::red));` La classe *QPainter* initialise une *QBrush* par défaut, transparente,
- la **fonte** (classe *QFont*) : définit la fonte des textes : famille, taille, style (italique,...). Elle est modifiée par la méthode *setFont()* qui prend un argument de type *QFont*. Ex : `setFont(QFont("Courier",30));` La classe *QApplication* met en place une fonte par défaut spécifique du système (ex : « `Ubuntu 11` » sous *Ubuntu*).

b) Système de coordonnées

QPainter a, classiquement, un système de coordonnées orthonormées (x,y) dont l'origine est en haut à gauche du *QWidget* de tracé (axe Y orienté vers le bas) avec pour unité le pixel de l'écran sous-jacent.



Si on veut représenter des éléments graphiques du « monde réel », il est en général nécessaire de passer dans un autre système de coordonnées, avec habituellement un axe des Y vers le haut et des unités propres au domaine représenté. Pour cela, deux solutions :

- manuelle : faire soi-même le code (fonction de conversion) pour convertir les coordonnées du « monde réel » vers le *QPainter*
- utiliser les fonctions de transformation de *Qt* (cf. ci-dessous).

Qt propose :

- des **fonctions de transformation** de coordonnées : `scale()`, `rotate()`, `translate()`,... Ces fonctions s'appuient en arrière-plan sur une matrice de transformation.

la manipulation directe de la matrice de transformation (classe *QTransform*), que l'on obtient par la méthode `worldTransform()` de la classe *QPainter*. C'est une matrice 3x3 qui transforme un point (`x`,`y`) du plan en un point (`x'`,`y'`) dans l'autre système.

`m11`	`m12`	`m13`
`m21`	`m22`	`m23`
`m31` `dx`	`m32` `dy`	`m33`

`m11` et `m22` spécifient l'échelle en `x` et en `y` : `scale()`

`m31` (`dx`) et `m32` (`dy`) spécifient une translation : `translate()`

`m21` et `m12` spécifient un « cisaillement » en `x` et en `y` : `shear()`

`m13`, `m23` et `m33` spécifient une projection (effet de perspective)

Une rotation `rotate()` est obtenue en modifiant `m11`, `m22`, `m21` et `m12`

Formules de base :

```
x' = m11*x + m21*y + dx
y' = m22*y + m12*x + dy
```

La valeur par défaut de la matrice de transformation est la matrice « identité » :

`1`	`0`	`0`
`0`	`1`	`0`
`0`	`0`	`1`

c) Les fonctions de tracé

QPainter fournit les fonctions pour tracer (*draw*) :

- la plupart des primitives géométriques : `drawPoint()`, `drawPoints()`, `drawLine()`, `drawRect()`, `drawRoundedRect()`, `drawEllipse()`, `drawArc()`, `drawPie()`, `drawChord()`, `drawPolyline()`, `drawPolygon()`, ...
- les textes : `drawText()`, `drawStaticText()`. La seconde fonction est optimisée pour un texte fixe, c'est-à-dire un texte dont la mise en page ne doit pas être recalculée à chaque « *repaint* » (cf. §d ci-dessous).
- les images (*pixmaps*) : `drawPixmap()`, à partir d'un fichier au format `bmp`, `gif`, `jpg`, `png`,..

d) Comment utiliser *QPainter* dans un *QWidget*

Quand il s'agit de faire des tracés (*paint* : « peindre ») dans un *QWidget*, on ne peut utiliser *QPainter* que dans un gestionnaire d'évènement `paintEvent()`.

L'évènement `QPaintEvent` :

Il indique une requête pour « repeindre » (***repaint***) tout ou partie d'un *widget*. Cet événement est très fréquent. Il est déclenché (automatiquement ou volontairement) par plusieurs causes :

- la première fois que le *widget* s'affiche,
- après qu'une autre fenêtre qui masquait tout ou partie du *widget* a été déplacée par l'utilisateur,
- la fenêtre où se situe le *widget* est restaurée après avoir été mise en icône par l'utilisateur,
- la fenêtre où se situe le *widget* est redimensionnée par l'utilisateur,
- l'utilisateur a « scrollé » le *widget* (s'il possède des ascenseurs),
- la fonction `repaint()` ou `update()` a été appelée explicitement par votre programme, Elles ont le même rôle (retracer le *widget*) mais il est préférable d'utiliser `update()` car `repaint()` suspend tous les autres évènements (notamment les évènements utilisateur).

Le **gestionnaire d'évènement (*event handler*)** `paintEvent()` est défini (*virtual*) dans la classe *QWidget*. Pour tracer des éléments graphiques dans votre *widget*, il faut redéfinir (*override*) ce gestionnaire d'évènement. Ex. dans une classe `MyWidget` :

```
void MyWidget::paintEvent(QPaintEvent *event) override {
     QPainter painter(this); // instanciation
     painter.setPen (Qt::blue);
     painter.drawText ( 10, 20, "Test painter" );
     painter.drawText ( rect(), Qt::AlignCenter, "Test painter centré" );
}
```

Le nom du paramètre n'est pas utilisé dans le corps. Cela génère un *warning* de compilation

Mot-clé `override` conseillé

Positionnement « en dur » déconseillé (sera-t-il visible?)

`rect()` renvoie la géométrie du rectangle du *widget*

Ces 2 paramètres permettent de positionner le texte au centre du *widget*

e) Un exemple complet : tracé d'un mur (plan d'architecte)

Thème : le programme doit permettre à un architecte de représenter numériquement un plan (murs, portes, fenêtres, etc.). Les coordonnées du plan sont en mètres avec pour origine le coin SW du terrain bâti.

Pour simplifier l'exemple on ne tracera qu'un simple mur (ligne) et l'ensemble du code – `main()` et classe `MyWidget` – sera dans le même fichier.

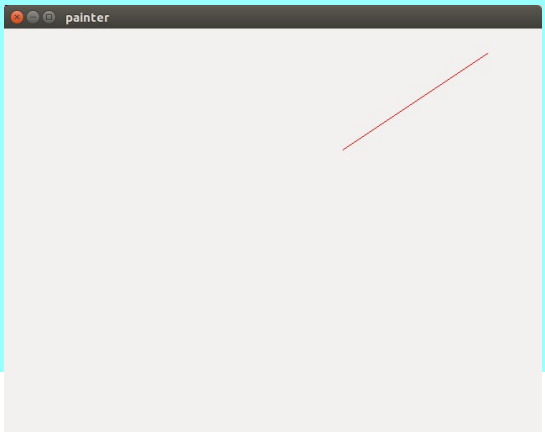
```
#include <QApplication>
#include <QWidget>
#include <QPainter>
#include <QDebug>

class MyWidget: public QWidget {
public:
  MyWidget( QWidget *parent = nullptr): QWidget(parent){ // constructeur
     //setMaximumSize( 200, 150); // On peut fixer une taille au widget
     // mais un widget (non pris dans un layout) a une taille par défaut:
     // souvent 640x480 (pixels) mais dépend de la plateforme.
  }
  void paintEvent ( QPaintEvent * ) override {
     QPainter painter(this);
     float x1 = 7, y1 = 3; // Point début du mur : (7,3) / coin SW
     float x2 = 10, y2 = 5; // Point fin du mur : (10,5) / coin SW
     // calcul de l'échelle : rapport dimension widget / coord. max plan
     float xmax = qMax(x1,x2);
     float ymax = qMax(y1,y2);
     float echelle = qMin ( width()/xmax, height()/ymax )*0.9;
```

Le nom du paramètre est omis ici

```
      // mise en place de l'échelle en (x, y)
      painter.scale ( echelle, -echelle ); // inversion des coordonnées Y
      painter.translate ( 0, -ymax*1.1); // translation des coordonnées Y
      // test : affichage de la matrice de transformation
      qDebug() << "worldTransform :" << painter.worldTransform();
      painter.setPen ( QPen(Qt::red, 0)); // couleur, épaisseur
      painter.drawLine(x1, y1, x2, y2);
  }
}; // fin définition de la classe MyWidget

int main(int argc, char **argv) {

  QApplication app(argc, argv);
  MyWidget w;
  w.show();
  return app.exec();
}
```

Bien noter le signe - pour l'échelle) en Y

0 indique que le trait fera toujours 1 pixel d'épaisseur (une épaisseur de 1 serait multipliée par l'échelle)



3- Le *framework « Graphics view »*

Il se place à un niveau conceptuel au-dessus de *QPainter* (qui reste le niveau sous-jacent). Une **scène** (classe dérivée de *QGraphicsScene*) contient des éléments (***items***) graphiques (classes descendantes de *QGraphicsItem*). Cette scène est visualisée par une ou plusieurs **vues** (classes dérivées de *QGraphicsView*).

a) La scène

La classe *QGraphicsScene* fournit un cadre pour gérer un grand nombre d'*items* graphiques 2D. Elle sert en fait de conteneur pour les *QGraphicsItems*. Elle possède des fonctionnalités permettant de déterminer la position des *items*, lesquels sont visibles, sélectionnés, ou en intersection avec une zone géométrique,...

Elle n'a pas d'apparence visuelle : elle n'hérite pas de *QWidget*. Elle a besoin d'une *QGraphicsView* pour être visualisée.

En général, les coordonnées utilisées dans une scène sont celles du « monde réel » (par ex. X, Y orthonormés avec un axe Y orienté vers le haut) en unités « naturelles » du domaine applicatif.

b) Les *items* graphiques

La classe *QGraphicsItem* a plusieurs sous-classes permettant de créer les objets graphiques les plus courants :

- *QGraphicsEllipseItem* : une ellipse (notamment un cercle)
- *QGraphicsLineItem* : une ligne
- *QGraphicsPathItem* : une suite d'*items* (« *path* »)
- *QGraphicsPixmapItem* : un *pixmap*
- *QGraphicsPolygonItem* : un polygone
- *QGraphicsRectItem* : un rectangle
- *QGraphicsSimpleTextItem* : un texte simple
- *QGraphicsTextItem* : un texte avec des fonctionnalités avancées

Création (instanciation) d'un objet de type hérité de *QGraphicsItem* :

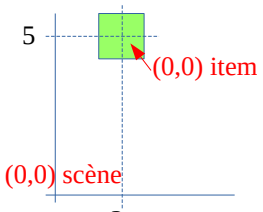
Les constructeurs de ces classes demandent en paramètre des caractéristiques géométriques : par ex. `x1`, `y1`, `x2`, `y2` pour un *QGraphicsLineItem*, `x`, `y`, `width`, `height` (ou un *QRectF*) pour un *QGraphicsRectItem* ou un *QGraphicsEllipseItem*, etc. Ces caractéristiques doivent être données en coordonnées « scène ».

Important :

Ces caractéristiques géométriques passées au constructeur correspondent à la définition de la « *bounding box* » de l'*item* et à la fixation de son origine = (0,0) *item*. La position de l'*item* dans la scène est fixée par l'appel à `setPos(x,y)` (à défaut l'*item* est positionné en (0,0) scène). Par ex., si on veut créer un carré de 2x2 centré sur la position (3,5), il faut écrire le code suivant :

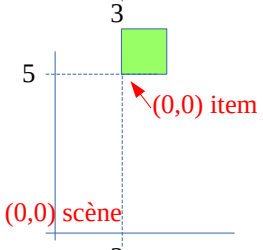
```
QGraphicsRectItem *carre =
          new QGraphicsRectItem ( -1, -1, 2, 2 );
carre->setPos(3,5);
```

x y width height



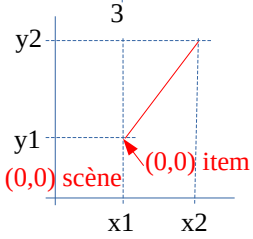
Par contre, si on veut créer le même carré à la position (3,5) mais avec l'origine en bas à gauche du carré :

```
QGraphicsRectItem *carre =
     new QGraphicsRectItem ( 0, 0, 2, 2 );
carre->setPos(3,5);
```



Pour une ligne dont on met l'origine sur le premier point (`x1,y1`) :

```
QGraphicsLineItem *ligne =
     new QGraphicsLineItem ( 0, 0, x2-x1, y2-y1 );
carre->setPos(x1,y1);
```



Les *items* graphiques ont donc leur propre système local de coordonnées (en unités de la scène) dont le (0,0) est fixé à la création de l'*item*. Dans la classe de l'*item* tout sera sera mesuré à partir de ce (0,0) et non pas de l'origine de la scène (cf. §d ci-dessous). Il peut avoir aussi sa propre matrice de transformation dont l'origine est (0,0) *item*. Par ex. une rotation va faire tourner l'*item* autour de son origine.

Ajout d'un *item* à la scène :

Il se fait simplement par la méthode `addItem()` de la classe *QGraphicsScene*. Cet ajout peut se faire à la création de la scène. Par exemple, en étant dans le constructeur de la scène, on a instancié un *QGraphicsLineItem* (variable `ligne`), l'instruction de rattachement est :

```
this->addItem(ligne);
```

Hiérarchie entre *items* :

Il est possible de construire une hiérarchie de type parent/enfant entre des *items*. Dans ce cas les *items* enfants sont rattachés à leur parent pour un déplacement ou une rotation par exemple. Ce rattachement se fait simplement avec la méthode `setParentItem()` de la classe *QGraphicsItem*, Par ex. on a créé un *item* pour représenter le point à l'extrémité d'une ligne (il n'y a pas de classe pour un point, mais on peut le représenter par un petit cercle par ex.). Chacun des 2 points créés sera rattaché à la ligne par :

```
point->setParentItem(ligne);
```

Cette instruction ajoute implicitement le point à la scène, à travers son parent. Dans ce cas, il ne faut pas faire `addItem()`.

Création des ses propres classes dérivées de *QGraphicsItem* :

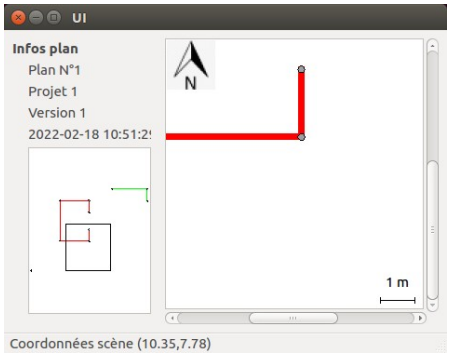
Les classes prédéfinies sont assez complètes si la scène est relativement passive, c'est-à-dire si on ne vient pas modifier les *items* après leur création. Elles permettent notamment d'afficher interactivement des informations sur les objets du « monde réel » sous-jacents aux *items* (à travers les *ToolTips*, cf. ex. plus bas). Elles permettent aussi dans une certaine mesure de bouger les *items*.

La dérivation d'une classe devient nécessaire si la gestion des *items* devient complexe, notamment dès qu'il faut répercuter les mises à jour des *items* vers une mise à jour des objets du « monde réel ».

⚠ **Remarque** : la classe *QGraphicsItem* n'hérite pas de *QObject*. Dans les classes dérivées on ne peut donc pas utiliser le système signal/*slot* (la macro Q_OBJECT n'est pas autorisée). On peut contourner cette limitation de plusieurs façons, notamment à l'aide de l'héritage multiple : la classe doit alors dériver de *QObject* et *QGraphicsItem*.

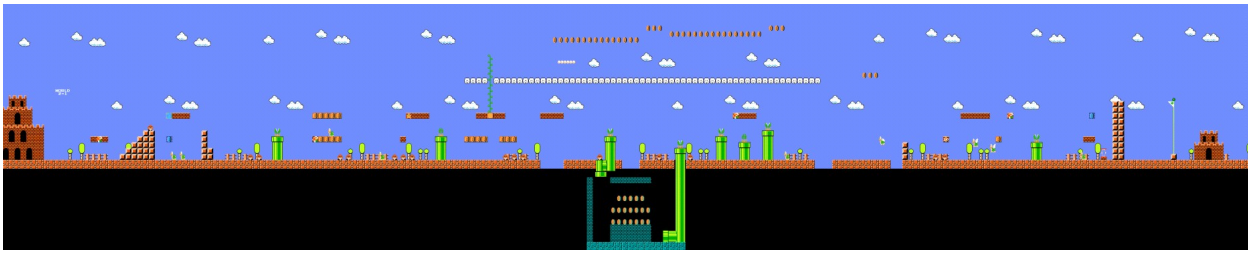
c) La vue

La classe *QGraphicsView* permet de visualiser le contenu d'une scène dans un *widget* « *scrollable* ». Il peut y avoir plusieurs vues différentes pour une scène. Il y en a obligatoirement une, sinon la scène n'est pas visible. Au départ la vue va se centrer automatiquement sur le centre de la scène et faire en sorte que tous les *items* soient visibles.



Exemples de scène/vues :

- Deux vues :
  une vue principale sur lequel on peut zoomer et une « mini-vue » non interactive qui prend toute la scène et indique le cadre visible dans la vue principale (ci-contre),
- Une vue qui se déplace :
  un jeu vidéo de plateforme où la scène représente l'ensemble du parcours et la vue représente une vue partielle qui se déplace avec la personnage (ci-dessous)



La scène

La vue 

Création d'une vue :

Elle prend obligatoirement la scène en premier paramètre du constructeur. Un second paramètre est le *widget* parent (`nullptr` par défaut) si la vue est à l'intérieur d'un autre *widget* (dans une *QMainWindow* par ex.).

```
myscene = new Scene();
myview = new QGraphicsView(myscene, this);
```

Si la vue n'a qu'un comportement simple (pas de zoom, pas de déplacement dans la scène, etc...) il n'est

pas nécessaire de dériver une sous-classe spécifique. Si vous en créez une, il faut en général redéfinir (*override*) certains « *event handlers* » selon vos besoins : *paintEvent* pour gérer un réaffichage de la vue, *resizeEvent* pour gérer une modification de taille du *widget*, *mouseMoveEvent* pour faire un *tracking* de la souris, *wheelEvent* pour gérer le zoom à la molette, *mousePressEvent* et *mouseReleaseEvent* pour des sélections à la souris, etc...

Deux méthodes de *QGraphicsView* sont intéressantes pour afficher des éléments graphiques « non-*items* », par ex. des éléments fixes liés à la vue mais pas à la scène : icônes, boutons, image de fond,... Il s'agit de `drawBackground()` et `drawForeground()`.

Si vous redéfinissez ces deux méthodes, un *QPainter* est passé automatiquement en premier argument : vous avez alors accès à l'API *QPainter* (cf. §2) pour réaliser des tracés spécifiques.
`drawBackground()` est appelé automatiquement (à chaque « *repaint* ») avant le tracé des *items* et `drawForeground()` après le tracé des *items*.

d) Le système de coordonnées du *framework*

Une des principales difficultés du *framework* « *Graphics View* » est la présence simultanée de trois systèmes de coordonnées : les coordonnées scène, vue et *items*. :

- C'est vous qui définissez le **système de la scène** (unité, échelle,...) : il doit correspondre à vos besoins applicatifs. Dans le code de la scène c'est celui qui est utilisé pour positionner les *items*.
- Le **système de la vue** est celui du périphérique sous-jacent (écran en général, mais ça peut aussi être une imprimante) : pour un écran l'origine est en haut à gauche et les unités en pixels. C'est dans ce système que vous récupérez la position des évènements « souris » de la vue par ex. Par contre, dans certaines méthodes de la vue, comme `drawBackground()` et `drawForeground()`, vous êtes en coordonnées « scène », sauf si vous déconnectez (temporairement) la matrice de transformation (`painter->setWorldMatrixEnabled(false);`).
- Le **système de chaque *item*** peut être différent. L'origine (0,0) est fixée au moment de la création (cf. plus haut §b), les unités et la matrice de transformation sont par défaut celles de la scène (mais il est possible d'appliquer une matrice spécifique à un *item)*. Quand vous gérez un événement souris dans le code d'un *item*, les positions sont en coordonnées « *item* ».

Pour permettre les conversions entre ces trois systèmes, le *framework* fournit des méthodes :

- dans la classe *QGraphicsView* : `mapFromScene()` et `mapToScene()`,
- dans la classe *QGraphicsItem* : `mapFromScene()`, `mapToScene()`, `mapFromItem()` et `mapToItem()`

Ex : pour transformer les coordonnées d'un événement souris en coordonnées « scène » dans un gestionnaire d'évènement de la vue :

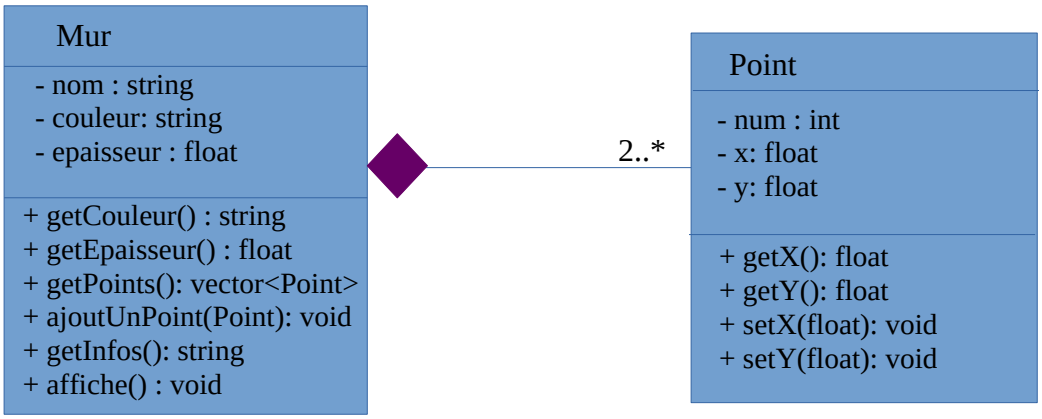
```
QPointF pos_scene = mapToScene(event->pos());
```

e) L'exemple du tracé d'un mur (plan d'architecte)

Reprendre l'exemple du §2 permettra de mettre en évidence les apports du *framework* « *Graphics View* ». Pour mieux formaliser l'exemple, voici un modèle UML simplifié du mur (vu par un bureau de dessin en architecture) : une classe ***Mur*** possède 4 attributs : un nom (*string*), une couleur (*string*), une épaisseur (*float*), un tableau de points (extrémités ou angles du mur : un *vector<Point>*). La classe ***Point*** possède 3 attributs : un numéro (donne l'ordre des points dans le tracé : *int*), un x et un y (*float*, en mètres depuis le coin SW du terrain). Dans une application réelle ces objets pourraient être lus dans une Base de Données.

Les méthodes nécessaires pour notre code sont :

- dans *Mur* : les *getters* `getCouleur()`, `getEpaisseur()`, `getPoints()`, un *setter* `ajoutUnPoint()` et une méthode `getInfos()` renvoyant une chaîne de caractères (*string*) des infos à afficher pour un mur.
- Dans *Point* : les *getters* `getX()`, `getY()` et les s*etters* `setX()`, `setY()`.



Pour simplifier l'exemple on ne tracera qu'un simple mur (ligne) comportant seulement deux points.

main.cpp

```
#include <QApplication>
#include "Mur.h"
#include "FenetrePrincipale.h"
int main(int argc, char **argv) {
      QApplication app(argc, argv);
      Mur mur( "A", "rouge", 0.2 );
      mur.ajoutUnPoint(Point( 1, 7.0, 3.0 ));
      mur.ajoutUnPoint(Point( 2, 10.0, 5.0 ));
      FenetrePrincipale mw ( mur );
      mw.show();
      return app.exec();
}
```

Création des objets du monde réel (ici : `mur`). Peut provenir d'une BD

Ils sont passés en paramètre de la *main window*, qui les transmettra à la scène

FenetrePrincipale.h

```
#include <QMainWindow>
#include <QGraphicsView>
#include "Scene.h"
#include "Mur.h"
class FenetrePrincipale : public QMainWindow {
      public:
            FenetrePrincipale(Mur &mur); // le mur en paramètre est passé ensuite à la scène
            ~FenetrePrincipale() override {}
      private:
            Scene *myscene; // classe dérivée de QGraphicsScene
            QGraphicsView *myview;
};
```

FenetrePrincipale.cpp

```
#include "FenetrePrincipale.h"

FenetrePrincipale::FenetrePrincipale(Mur &mur){   // constructeur
      myscene = new Scene(mur);  // création de la scène (on lui passe l'objet mur)
      myview = new QGraphicsView(myscene, this);  // création de la vue liée à la scène
      myview->scale(1,-1); // inversion des coordonnées Y
      myview->fitInView(myscene->sceneRect(),Qt::KeepAspectRatio);
      this->setCentralWidget(myview);  // la vue est le widget central de la main window
}
```

méthode de *QGraphicsView* qui déclenche le recalcul de l'échelle (*scale*) de la vue et la recentre pour que tous les *items* de la scène soient visibles

Scene.h

```
#include <QGraphicsScene>
#include <QGraphicsItem>
#include <QString>
#include <map>
#include "Mur.h"

class Scene : public QGraphicsScene {
      public :
            Scene(Mur & mur);  // le mur est récupéré en paramètre
      private:
            static std::map<std::string, QColor> tab_couleurs;
};
```

tableau associatif (`static`) qui fait le lien entre une couleur (*string*) et une *QColor* prédéfinie. Déclaré dans le `.h` et initialisé dans le `.cpp`

Scene.cpp

```
#include "Scene.h"

std::map<std::string, QColor> Scene::tab_couleurs =
 {{"rouge", Qt::red}, {"vert", Qt::green}};

Scene::Scene(Mur &mur) {   // constructeur
      // Ajout des items graphiques dans la scène
      QColor coul = tab_couleurs[mur.getCouleur()];
      qreal epais = mur.getEpaisseur();
      Point point1 = mur.getPoints()[0];
      Point point2 = mur.getPoints()[1];

      // item ligne
      QGraphicsLineItem *ligne = new QGraphicsLineItem (
            0, 0, point2.getX()-point1.getX(), point2.getY()-point1.getY());
      // l'origine de la ligne est le point 1
      ligne->setPos(point1.getX(), point1.getY());
      ligne->setPen(QPen(coul,epais,Qt::SolidLine));
```

Récupération des attributs des objets du monde réel (ici le `mur` avec ses points)

pour simplifier l'exemple on n'a gardé que 2 points

Création de l'*item* ligne. On passe sa *bounding box* relative au constructeur (cf. §b) et on fixe sa position dans la scène au point 1

caractéristiques graphiques de l'*item*

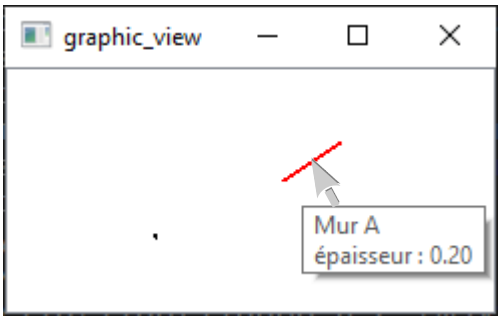
```
    std::string text_tooltip = mur.getInfos();
    ligne->setToolTip(QString::fromStdString(text_tooltip));

    this->addItem(ligne);

    // Matérialisation du point origine de la scène (taille=épaisseur mur)
    qreal taille_o = epais;
    QGraphicsEllipseItem *origine = new QGraphicsEllipseItem(
                    -taille_o/2,-taille_o/2,taille_o,taille_o);
    origine->setPos(0,0);   // le milieu du cercle est positionné au (0,0) de la scène
    origine->setPen(QPen(Qt::black,0,Qt::SolidLine));// épaisseur du trait 0 = 1 pixel
    this->addItem(origine);
}
```

Mise en place du *ToolTip* qui permet au survol de la ligne d'afficher les caractéristiques de l'objet du monde réel (`mur`)

Ajout de l'*item* de type *QGraphicsLineItem* à la scène

Ajout de l'*item* de type *QGraphicsEllipseItem* à la scène

Rendu à l'exécution (sous *Windows*), au survol de la souris sur la ligne :



Cet exemple permet de montrer l'**apport du *framework* « *Graphics View* »** par rapport à l'API *QPainter* :

- mise à l'échelle et positionnement automatique dans la vue des *items* de la scène
- interactivité : pour l'utilisateur, accès dans la vue aux caractéristiques de l'objet du monde réel sous-jacent à l'*item* graphique.

Cet apport a pour contrepartie une complexification du code, mais celle-ci est concentrée essentiellement dans la création des *items* graphiques dans la scène.

Améliorations possibles :

1. rendre la vue « zoomable » grâce à la molette de la souris : il faut dériver une sous-classe de *QGraphicsView*,
2. permettre à l'utilisateur de déplacer un *item* (ici la ligne) :
   a) modification assez simple, à faire à la création de l'*item* dans la scène,
   b) plus difficile si, après déplacement, on veut gérer la mise à jour des caractéristiques de l'objet sous-jacent (ici le mur),
3. améliorer la présentation graphique en visualisant les points d'extrémité de la ligne. Il n'existe pas de *QGraphicsItem* pour représenter un point : on peut le faire à l'aide d'un petit cercle ou d'un petit rectangle.

On va présenter ci-dessous les modifications de code pour les points 1 et 2a. Le point 3 ne présente pas de difficultés majeures : il faut créer dans la scène deux *QGraphicsEllipseItem* (ou rectangle) et les rattacher à l'*item* parent (ligne), cf. §b Hiérarchie entre *items*.

Il y a plusieurs possibilités pour réaliser le point 2b, qui ont trait aux principes MVC (modèle-vue-contrôleur) qui seront évoquées plus loin (§4).

Création d'une vue spécifique : classe `GrandeVue` (tout le code est *inline* dans le `.h` pour simplifier)

GrandeVue.h

```
#include <QGraphicsScene>
#include <QGraphicsView>
#include <QWheelEvent>
#include "Scene.h"
class GrandeVue : public QGraphicsView {
      public :
            GrandeVue(Scene *scene, QWidget *w) : QGraphicsView(scene, w){
                  Q_UNUSED(scene);
                  Q_UNUSED(w);
            }
            ~GrandeVue() override {}
      private:
            // redéfinition de gestionnaires d'évènements
            // Fit de la vue sur les limites de la scène
            void resizeEvent (QResizeEvent *) override {
                  if ( this->transform().m11() == 1 ){ // m11 : échelle x ds la matrice
                        this->fitInView(sceneRect(), Qt::KeepAspectRatio);
                  }
            }
            // événement déclenché par la molette
            void wheelEvent(QWheelEvent *event) override {
                  int angle = event->angleDelta().y();//angle donne le sens de la molette
                  qreal facteur_zoom = 1;
                  if (angle > 0 ){
                        facteur_zoom = 1.1;
                  } else {
                        facteur_zoom = 0.9;
                  }
                  centerOn(mapToScene(event->pos()));// position «vue» vers «scène»
                  scale ( facteur_zoom, facteur_zoom );
            }
};
```

On repasse les paramètres à la classe mère (*QGraphicsView*)

pour éviter les *warnings* à la compilation (*macro Qt*)

On appelle `fitInView()` au premier « *resize* » (quand l'échelle vaut 1)

On centre le zoom sur la position de la souris

Le code du constructeur de la *main window* devient :

```
FenetrePrincipale::FenetrePrincipale(Mur &mur){   // constructeur
      myscene = new Scene(mur); // création de la scène (on lui passe l'objet mur)
      myview = new GrandeVue(myscene, this); // création de la vue liée à la scène
      myview->scale(1,-1); // inversion des coordonnées Y
      this->setCentralWidget(myview); // la vue est le widget central de la main window
}
```

Ici plus d'appel à `fitInView()` : fait dans l'évènement « *resize* »

Pour **rendre un *item* déplaçable** (*movable*) par « *drag & drop* » il suffit de rajouter, juste après sa création dans la scène, l'instruction suivante (ex. ici sur la ligne) :

```
ligne->setFlags( QGraphicsItem::ItemIsMovable );
```

Remarque : la ligne peut être déplacée, mais sans changer d'orientation. Pour faire des déplacements plus complexes (rotation, allongement, etc.) il faut dériver une sous-classe pour le *QGraphicsItem* considéré.

4- Le *pattern* MVC (Modèle-Vue-Contrôleur)

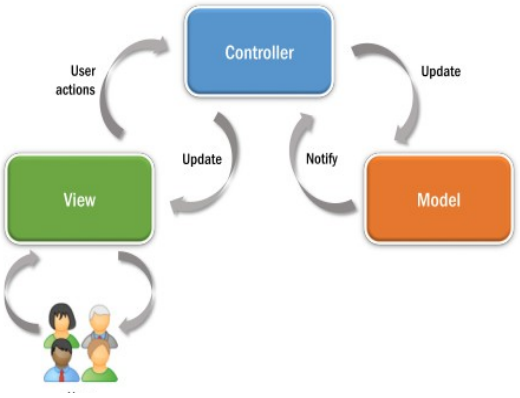
Dès les premiers ordinateurs proposant une interface graphique (recherches du *Xerox Palo Alto Research Center* dans les années 1970), les programmeurs se sont heurtés à la complexité de développement de ces interfaces quand l'application a beaucoup de fonctionnalités.

Un des idées phares pour améliorer la maintenabilité et la sécurité du code est appelée « *Separation of Concerns* » (*SoC*), le but étant de découper un programme en parties relativement indépendantes. Notamment, dans le développement d'IHM, l'idée est de séparer la partie présentation (les objets visibles sur l'écran) de la partie « métier » de l'application (les objets du « monde réel »).

Le *pattern* MVC (*Model View Controller*) a été proposé dans les années 1980 par les créateurs (aussi au *Xerox PARC*) du langage *Smalltalk*, un des premiers langages à cibler le développement graphique.

Les idées directrices :

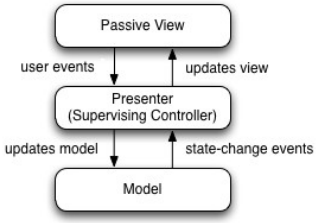
- le **modèle** : indépendant de l'IHM, il gère directement les données (*data*) et la logique « métier »
- la **vue** : la présentation à l'utilisateur. Plusieurs vues distinctes sont possibles pour la même donnée
- le **contrôleur** : il reçoit les entrées utilisateur et les convertit en commandes (notamment *updates*) pour le modèle ou la vue.



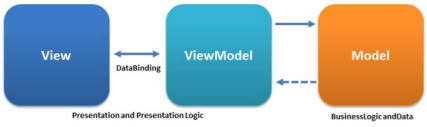
Source : https://fr.wikipedia.org/wiki/Modèle-vue-contrôleur

C'est un concept général qui peut être décliné en plusieurs variantes (dont la description peut parfois être totalement contradictoire selon les sites...) :

- dans le *pattern* MVC d'origine (*Smalltalk*), il existe un lien direct entre la vue et le modèle : la vue peut se mettre à jour à partir des informations du modèle sans passer par le contrôleur.
- la variante **MVP** (*Model-View-Presenter*) ou « Modèle-Vue - Présentateur » : toutes les informations entre la vue et le modèle passent obligatoirement par le *presenter* (présentateur). Ce *pattern* a lui même plusieurs variantes (avec vue passive ou non).



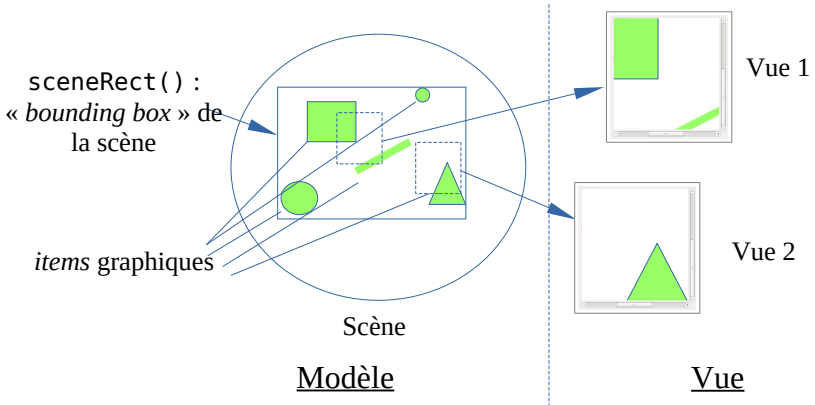
- la variante **MVVM** (*Model-View-ViewModel*) ou « Modèle-Vue - Modèle de Vue » : le « modèle de vue » (*View Model*) est une abstraction de la vue. Son lien avec la vue est « automatique » : les évènements se passant dans la vue déclenchent un *callback* dans le « modèle de vue ».



– il existe même une variante **MV** (*Model View*), ou la communication est directe entre les deux parties, sans intermédiaire.

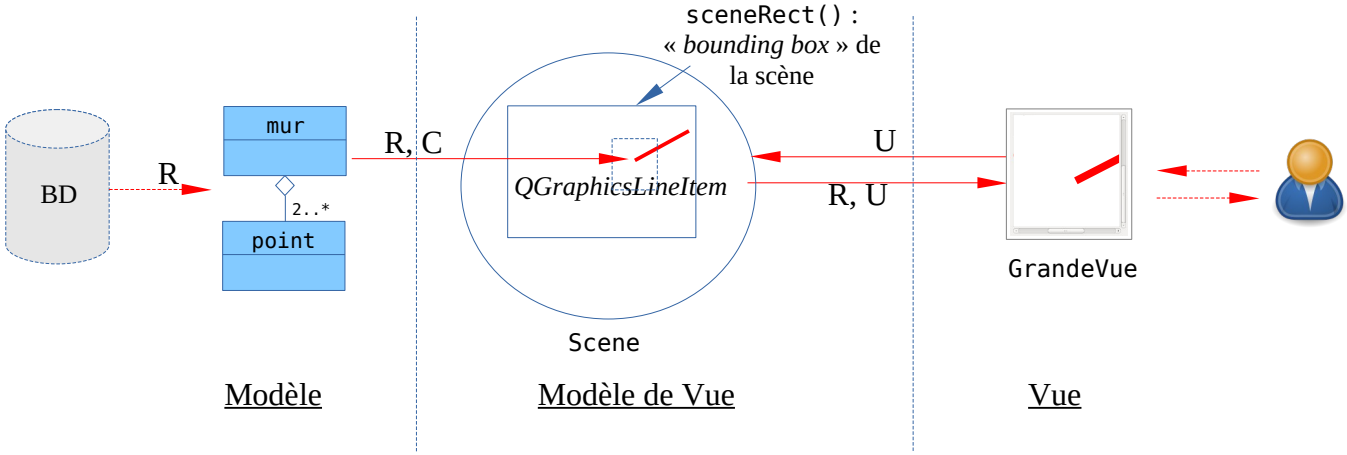
a) Utilisation du pattern MVC dans notre exemple (tracé d'un mur)

Le ***pattern* MVC** n'est pas un *framework* logiciel, c'est plutôt un **cadre conceptuel** qui permet de réfléchir avec du recul et de rationaliser la conception d'applications. On peut essayer de voir comment le *framework* « *Graphics View* » peut se situer dans ce cadre.



De base, sans les données du « monde réel », le *framework* est compatible avec un pattern **Modèle-Vue** : le modèle est constitué de la scène et des *QGraphicsItems* contenus et la vue est constituée des *QGraphicsView*.

Essayons de représenter notre exemple du §3e dans le cadre d'une conception MVC. Les transferts d'information (ou de données) entre les différents blocs seront représentées selon le formalisme CRUD : C (*Create* : création ou ajout d'objet), R (*Retrieve* ou *Read* : récupération d'informations), U (*Update* : mise à jour), D (*Delete* : suppression d'objet). Dans les schémas ci-dessous les lignes rouges indiquent un transfert d'information ou de données, le sens de la flèche indique le sens du transfert.



Ce qui semble le mieux correspondre est le pattern MVVM : le Modèle correspond aux données (*data*) du « monde réel » et la Vue à la vue, bien sûr. Il n'y a pas de lien direct entre Vue et Modèle, on se situe donc dans le concept MVP ou MVVM. La scène (objet *Scene* contenant les *QGraphicsItem*) se rapproche assez d'une « abstraction de la vue », elle représente donc plutôt une « *View Model* » qu'un « *Presenter* ».

Ce qui manque pour que le *pattern* soit complet est la possibilité de mise à jour (*update*) de la scène vers le modèle (transfert « U » vers la gauche). Concrètement, dans notre exemple, quand l'utilisateur déplace la ligne à l'écran la position de l'objet *QGraphicsLineItem* est mise à jour, mais pas les positions des points du mur (*data*).

Pour pouvoir réaliser cet « *update* », plusieurs options sont envisageables :

1. dans la classe `Mur` ajouter une référence à l'objet *QGraphicsLineItem* lié. Cette option n'est pas satisfaisante car elle introduit une notion de « présentation » dans les objets « métier », ce qui contredit un des concepts majeurs à l'origine du MVC, le découplage : il est souhaitable que le modèle ignore tout de la façon dont il est présenté .
2. dans la classe *QGraphicsLineItem* ajouter une référence à l'objet `Mur` lié : cette option est un peu plus acceptable dans la mesure ou l'objet *QGraphicsLineItem* est inclus dans la partie centrale, pas dans la visualisation proprement dite. Cette option implique une dérivation vers une sous-classe adaptée de *QGraphicsLineItem*.
3. ajouter une classe intermédiaire qui a une référence vers les deux objets : c'est la solution qui assure le meilleur découplage entre données et présentation. Elle implique cependant un niveau supplémentaire et une plus grande complexité du code. Cette classe intermédiaire jouerait alors un rôle de présentateur (ou contrôleur).

b) Réalisation de l'« *update* » du mur Option 2 :

Qt a prévu pour la classe mère *QGraphicsItem* un attribut « *custom* » d'usage libre par le développeur. Cet attribut s'appelle `data` :

- le *setter* correspondant, `setData()` prend en paramètres un entier et un *QVariant*. Un objet *QVariant* peut contenir une valeur dont le type T est défini par le programmeur. Cela permet d'utiliser `data` de façon générique, selon le besoin. Le paramètre entier peut indiquer par ex. quel est le type T, si `data` est utilisé plusieurs fois dans des contextes différents. Ici nous mettrons simplement 0.
- le getter `data()` prend en paramètre le même entier (ici 0) et retourne un *QVariant*.

Nous allons mettre dans `data` un pointeur générique (`void *`) qui stockera l'adresse de l'objet `Mur` sous-jacent à l'*item* ligne. La syntaxe est un peu (!!) compliquée :

- stockage de l'adresse de `mur` :

création d'un *QVariant*

```
QVariant var (QVariant::fromValue(static_cast<void *>(&mur)));
ligne->setData(0, var);
```

- récupération de l'adresse de `mur` : conversion du *QVariant* en `Mur *`

```
QVariant var = data(0);
Mur *mur = static_cast<Mur *>(var.value<void *>());
```

Création d'une classe d'*item* spécifique : classe `Ligne` dérivée de *QGraphicsLineItem*

Ligne.h

```
#include <QGraphicsScene>
#include <QGraphicsItem>
#include <QGraphicsSceneMouseEvent>
#include "Mur.h"
class Ligne: public QGraphicsLineItem {
public:
     Ligne(qreal x1, qreal y1, qreal x2, qreal y2):
                                          QGraphicsLineItem(x1,y1,x2,y2) {}
     ~Ligne (){}
```

On repasse les paramètres à la classe mère (*QGraphicsLineItem*)

```
    // redéfinition de gestionnaires d'évènements
    void mouseReleaseEvent(QGraphicsSceneMouseEvent *event) override;
    void mouseMoveEvent(QGraphicsSceneMouseEvent *event) override;
private :
    bool drag = false;  // false => pas de « drag and drop » en cours
};
```

Ligne.cpp

```
#include "Ligne.h"
void Ligne::mouseMoveEvent(QGraphicsSceneMouseEvent *event) {
    drag = true;   // en cours de « drag and drop »
    // re-propage l'évènement
    QGraphicsItem::mouseMoveEvent(event);
}
void Ligne::mouseReleaseEvent(QGraphicsSceneMouseEvent*event){//gestion du «drop»
    if ( !drag ) return;   // clic de souris sans « drag and drop »
    qreal x1 = pos().x(); // coord Scene
    qreal y1 = pos().y(); // coord Scene
    qreal x2 = pos().x()+ line().x2(); // coord Scene
    qreal y2 = pos().y()+ line().y2(); // coord Scene
    QVariant var = this->data(0);
    Mur *mur = static_cast<Mur *>(var.value<void *>());
    // Récupération des 2 points du mur : getPoints()
    Point &pt1 = mur->getPoints()[0]; // référence sur le point d'origine dans mur
    Point &pt2 = mur->getPoints()[1]; // référence
    pt1.setX(x1);
    pt1.setY(y1);
    pt2.setX(x2);
    pt2.setY(y2);
    drag = false; // fin du « drag and drop »
    // re-propage l'évènement
    QGraphicsItem::mouseReleaseEvent(event);
}
```

Point d'origine : (0,0) de l'*item*

Récupération des positions des points d'extrémité de l'*item Ligne* : `pos()` donne les coordonnées scène)

Récupération de l'adr. du `mur` par `data()`

update des points du `mur` dans le modèle (données)

Le code de la création de `ligne` dans `Scene.cpp` est modifié :

au lieu de *QGraphicsLineItem*

```
    Ligne *ligne = new Ligne (
        0, 0, point2.getX()-point1.getX(), point2.getY()-point1.getY());
    ...
    ...
    QVariant var(QVariant::fromValue(static_cast<void *>(&mur)));
    ligne->setData(0, var);
```

Passage de l'adr. du `mur` à `data()`

Pour vérifier que le `mur` est bien modifié à la fin du `main()` :

```
    ...
    ...
    int ret = app.exec();
    qDebug() << "Fin prog";
    mur.affiche();
    return ret;
}
```