TP N°9
Une application graphique 2D sous Qt

Objectif

L'objectif de ce TP est de vous familiariser avec le *framework* « *Graphics View* » en créant une application graphique qui visualise des objets en 2D.

1. Sujet

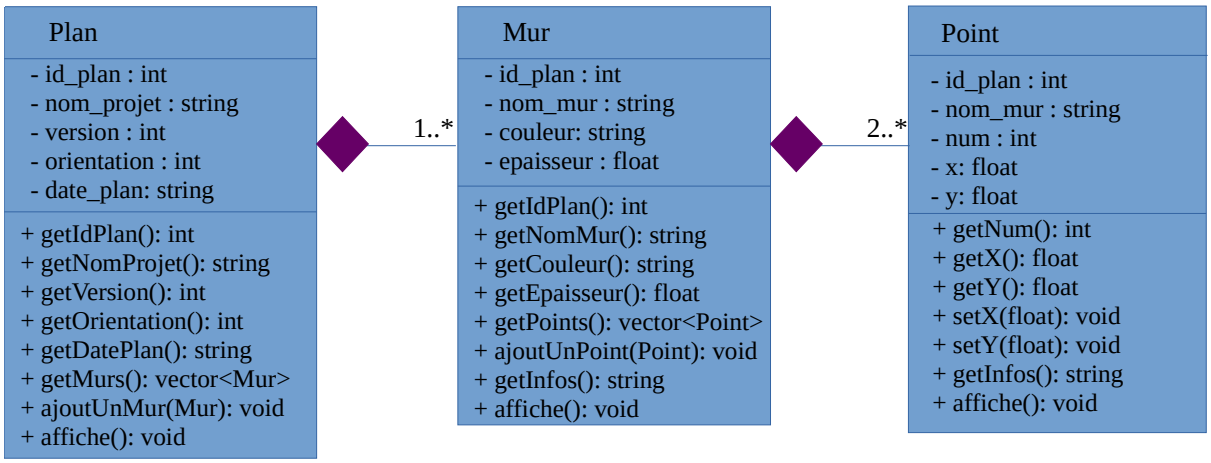
Le thème du programme est de permettre à un architecte de représenter numériquement un plan (murs, portes, fenêtres, etc.). Les coordonnées du plan sont en mètres avec pour origine une borne du terrain bâti. Dans le cadre de ce TP on représentera un plan qui ne contient que des murs.

Le plan est stocké dans une base de données *MySQL* (base de données *plans*). Le fichier *plans.sql* fourni permet de créer les tables et d'y charger des données de test. Les données peuvent être extraites de cette base de données à l'aide de fonctions C++ (nécessite *libmysqlcppconn*, cf. TP N°6) La librairie statique *libBDD.a* qui contient les binaires des fonctions (compilés pour Linux 64bits) et les *headers BDD.h*, *Plan.h*, *Mur.h* et *Point.h* sont fournis (*model.zip*, disponible sous *Moodle*). On conseille de copier ces fichiers dans un répertoire « *model* » (cf. §2). Ces fonctions permettent d'instancier un objet *Plan* qui contiendra toutes les données.

Un plan est constitué des classes suivantes :

- la classe ***Plan*** possède 6 attributs : *id* (*int*), *nom_projet* (*string*), *version* (*int*), *orientation* (*int*, en degrés par rapport au nord), *date_plan* (*string*), un tableau *murs* (*vector<Mur>*).
- la classe ***Mur*** possède 5 attributs : *id_plan* (*int*), *nom_mur* (*string*), *couleur* (*string*), *epaisseur* (*float*), un tableau *points* (extrémités ou angles du mur : *vector<Point>*).
- la classe ***Point*** possède 5 attributs : *id_plan* (*int*), *nom_mur* (*string*), *num* (donne l'ordre des points dans le tracé : *int*), un *x* et un *y* (*float*, en mètres depuis un coin du terrain).

Ci-dessous un UML simplifié du « modèle » (= modèle de données dans MVC) :



2. Mise en place des répertoires de développement et de la base de données

Dans un répertoire que vous avez créé pour ce TP (par ex. « *tpqt3* »), créez un sous-répertoire « *model* » dans lequel vous allez mettre les fichiers décompactés de *model.zip*.

Dans le répertoire « *tpqt3* », créez des fichiers (vides pour l'instant) *main.cpp*, *FenetrePrincipale.h*, *FenetrePrincipale.cpp*, *LoginDialog.h* et *LoginDialog.cpp*. La commande pour générer un fichier *.pro* (selon le nom du répertoire, par ex. *tpqt3.pro*) : `qmake –project –norecursive`

Dans ce fichier *tpqt3.pro* vous allez ajouter les lignes suivantes :

```
QT += widgets
QMAKE_CXXFLAGS += -std=c++11 -g
LIBS += model/libBDD.a
LIBS += -lmysqlcppconn
```

La base de données :

Le fichier *plans.sql* est fourni sous *Moodle*. Connectez-vous à *mysql* et créez une base de données « *plans* » et entrez dans cette base puis créez les tables et les données :

```
create database plans;
use plans
source plans.sql
```

Trois tables ont été créées (*plan*, *mur* et *point*). Vous pouvez faire des « *select* » pour voir ce qu'elles contiennent (pour information) puis quitter *mysql*.

3. Écriture du `main()` et du « *Dialog box* » de saisie des infos de connexion à la base de données

Ci-dessous le code du `main()` contenant :

- l'appel au « *dialog box* » dans lequel l'utilisateur saisit les infos pour la connexion à la base
- l'appel aux fonctions de « *model* » pour ouvrir la base et y lire les données du plan.

```
// includes VIEW
#include <QApplication>
#include <QMessageBox>
#include "LoginDialog.h"
// includes MODEL
#include <cppconn/exception.h>
#include "model/Plan.h"
#include "model/BDD.h"

int main(int argc, char **argv) {
      QApplication app(argc, argv);

      LoginDialog dlg;
      std::string host, base, user, pwd;

      if ( ! dlg.exec() ){ // Ouverture de la boite de dialogue
            std::cout << "Sortie de l'application\n";
            return 1;
      }
```

```
	// Récupération des saisies après fermeture de la Dialog box
	dlg.getResult(host, base, user, pwd);
	std::cout << "Lecture base plans" << std::endl;
	Plan plan;

	try {
		// Connexion BD
		BDD bdd("tcp://"+host+":3306", base, user, pwd);
		// Récupération du plan
		plan = bdd.getPlan(1);
		plan.affiche();
	}
	catch (sql::SQLException &e) {
		std::cout << "Erreur MySQL. Sortie de l'application\n";
		QMessageBox msg( QMessageBox::Critical, "Erreur mySQL",
				e.what());
		msg.exec();
		return 1;
	}
	return 0;
}
```

Le code qu'il vous reste à écrire :

La classe *LoginDialog* est une classe fille de *QDialog*. Vous allez mettre en place un *layout* de type « *QGridLayout* », qui contient :

- 4 *QLabel*,
- 4 *QLineEdit*,
- 2 *QPushButton*

- Un clic sur le bouton « *Login* » déclenche un *slot* (à écrire) qui lit les 4 champs de type *QLineEdit* et stocke le contenu dans 4 *QString* puis appelle la méthode *accept*() de *QDialog* (qui ferme la boîte de dialogue en retournant *true*).

  Si l'un des champs est vide, vous devez ouvrir une *popup* (*QMessageBox* de type « *warning* ») à l'aide du code ci-dessous, mais ne pas quitter la boîte de dialogue :

```
QMessageBox::warning( this, "Warning",
                              "Tous les champs doivent être remplis");
```

- Un clic sur le bouton « *Annuler* » déclenche le *slot* prédéfini *close*() de *QDialog* (qui ferme la boîte de dialogue en retournant *false*).

Une dernière méthode à écrire pour votre classe *LoginDialog* : `getResult()`, qui a 4 paramètres de sortie de type *std::string* correspondant aux *QString* des 4 champs *QLineEdit* (à convertir en *std::string*, cf. méthode to*StdString()* de la classe *QString*).

Le rendu de cette boite de dialogue :

4. Écriture de la « *main window*» (classe *FenetrePrincipale*)

Ci-dessous la suite du code du `main()` contenant :

- l'instanciation d'une « *Fenetre Principale* » qui prend en paramètre le plan lu dans la base,
- l'appel à `app.exec()` qui démarre la boucle d'attente d'évènements de l'application.

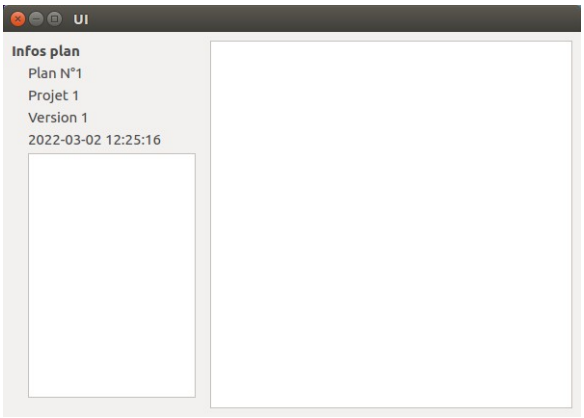
```
FenetrePrincipale mw ( plan );
mw.show();

return app.exec();
```

La classe *FenetrePrincipale* (sous-classe de *QMainWindow*) va pour l'instant seulement mettre en place son *layout* basique, avec une scène vide. On va fixer une *minimumSize* à la fenêtre. Les éléments à visualiser :

- à gauche une « *Group Box* » portant le titre « Infos plans » contenant 4 *QLabel* indiquant l'*id_plan*, le *nom_projet*, la *version* et la *date_plan*, dont le texte est récupéré par les *getters* de la classe Plan. En dessous de ces *labels*, elle contient une *QGraphicsView* (qui sera destinée à contenir une « mini-vue » de la scène). On va donner à la « *Group Box* » une *maximumWidth* égale au 1/3 de la largeur de la « *Fenetre Principale* »
- à droite une *QGraphicsView* qui sera destinée à contenir la vue principale (zoomable) de la scène.

Ci-dessous le rendu de cette « *Fenetre Principale* » :



5. Écriture de la scène (classe *Scene*)

Il s'agit d'écrire la classe *ScenePlan* (sous-classe de *QGraphicsScene*) qui va créer tous les *QGraphicsItem* correspondant aux objets Mur du modèle de données. Son constructeur prend le plan en paramètre.

Dans un premier temps vous pouvez créer uniquement des *QGraphicsLineItem* (de la couleur et de l'épaisseur et à la position obtenus par les *getters* des objets Mur et Point). Vous pouvez vous inspirer du cours *Qt4* pp 10 et 11, sachant qu'ici les murs n'ont pas seulement 2 points mais un tableau de points. Il n'existe pas de *Polyline* en tant que *QGraphicsItem*, vous devrez donc créer plusieurs *QGraphicsLineItem* pour un seul mur.

Le texte des *ToolTips* d'un mur sera obtenu par la méthode `getInfos()` de la classe Mur.

Dans un second temps vous pouvez ajouter à votre code des *QGraphicsEllipseItem* pour représenter les points du mur. Le texte des *ToolTips* de chaque point sera obtenu par la méthode `getInfos()` de la classe Point.

Pour créer la scène, dans *FenetrePrincipale.cpp* :

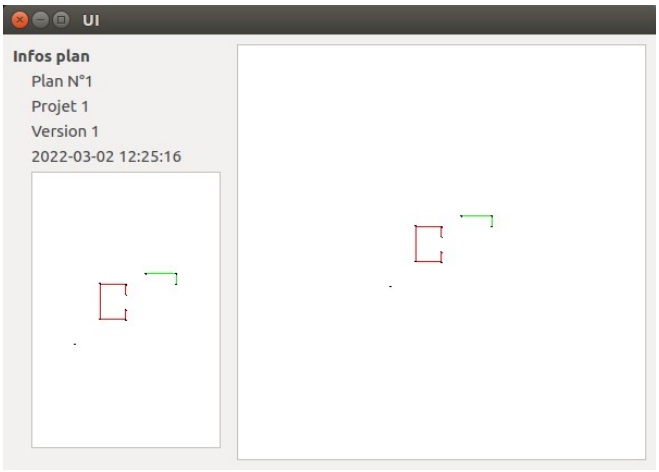
```
myscene = new ScenePlan(plan);
```

Les vues étaient déjà rattachées à la scène vide. L'affichage de la scène (sur les deux vues) va donc se faire automatiquement.

Pour inverser les Y et redimensionner la scène dans la vue il faut ajouter (pour les 2 vues) le code suivant dans *FenetrePrincipale.cpp* :

```
myview1->scale(1,-1); // inversion des coordonnées Y
myview1->fitInView(myscene->sceneRect(),Qt::KeepAspectRatioByExpanding);
```

Le rendu de la « *Fenetre Principale* » :



Le redimensionnement n'est pas complètement satisfaisant : la scène apparaît trop petite. Une solution à ce problème sera apporté au § suivant, en dérivant une sous-classe de *QGraphicsView*.

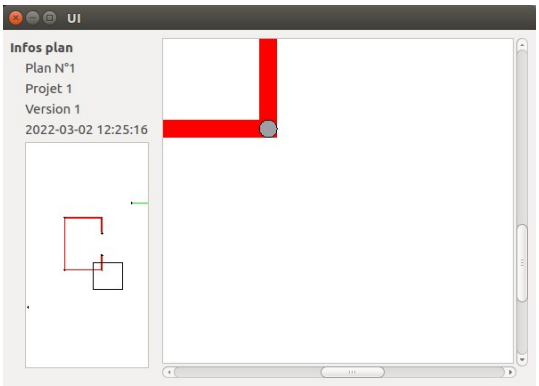
6. Écriture de classes spécialisées pour les deux vues (classes *GrandeVue et MiniVue*)

La classe *GrandeVue* (sous-classe de *QGraphicsView*) va ajouter une possibilité de zoom (à l'aide de la molette de la souris). Pour cela, dans cette classe, il faudra redéfinir les « *events handlers* » `resizeEvent()` et `wheelEvent()`, cf. p.12 du cours *Qt4*.

La classe *MiniVue* (aussi sous-classe de *QGraphicsView*) n'aura pas de zoom mais devra afficher le cadre du *viewport* (c'est-à-dire la zone visible de la scène) en cours dans la *GrandeVue*. Pour cela *GrandeVue* devra émettre un signal spécifique (à partir de `wheelEvent()`) qui envoie les infos de taille du *viewport*.

C'est la « *Fenetre Principale* » qui se chargera de connecter ce signal émis par *GrandeVue* à un *slot* spécifique de *MiniVue*, qui tracera le *viewport*.

Rendu avec la « *Grande Vue* » dans un état « zoomé »



7. Ajout de fonctionnalités supplémentaires dans la « *Grande Vue* »

Les éléments à rajouter à la classe *GrandeVue* :

a. « *Tracking* » de la souris

La position du curseur de la souris, dans le système de coordonnées de la scène, devra s'afficher dans la barre de statut (« *status bar* », en bas de la fenêtre) de la « *Fenetre Principale* ». Pour cela, il faut :

- ajouter un attribut de type *QStatusBar* dans *FenetrePrincipale.h* :

```
QStatusBar *barre_statut;
```

- affecter à cet attribut la barre de statut de *FenetrePrincipale* (dans le constructeur) :

```
barre_statut = statusBar();
```

- redéfinir l'« *event handler* » `mouseMoveEvent()` dans la classe *GrandeVue*. Dans celui-ci, on émettra un signal spécifique qui prend en paramètre un *QPointF* (position de la souris en coordonnées scène). La déclaration de la méthode :

```
void mouseMoveEvent(QMouseEvent *event) override;
```

- dans le constructeur de la classe *GrandeVue*, il faut autoriser le « *tracking* » de la souris (par défaut, il est désactivé). Si on ne l'active pas, les événements de type `mouseMoveEvent` ne sont pas déclenchés, donc on ne rentrera jamais dans l'« *event handler* » `mouseMoveEvent()`. L'instruction à ajouter dans le constructeur :

```
setMouseTracking(true);
```

- ajouter un *slot* dans *FenetrePrincipale.h*. Il prend en paramètre un *QPointF* :

```
void affiche_pos_scene( QPointF p);
```

- écrire le code de ce *slot* dans *FenetrePrincipale.cpp*. Il va écrire un message « texte » indiquant les coordonnées (x,y) de p et passer ce message à la barre de statut :

```
barre_statut->showMessage(msg);
```

- dans le constructeur de *FenetrePrincipale.cpp* connecter le signal émis par *GrandeVue* au *slot* `affiche_pos_scene()`.

b. Ajouter à la « Grande Vue » une flèche indiquant le Nord

Une image « *north.png* » vous est fournie sous *Moodle*, mais vous pouvez en changer.

Cette image doit être positionnée en haut à gauche dans la « *Grande Vue* » et elle doit être visible quel que soit le zoom et le « *scroll* ».

Sa mise en place se fera dans une fonction de la classe *GrandeVue* qui redéfinira la méthode `drawBackground()` de *QGraphicsView*, cf. p.8 du cours *Qt4*.

La flèche devra être orientée selon l'angle donné par l'attribut `orientation` du Plan (*getter* `getOrientation()`). Cette valeur devra donc être passée en paramètre au constructeur de la classe *GrandeVue*.

c. Ajouter une échelle à la « Grande Vue »

Une échelle, trait d'un mètre en coordonnées scène, accompagné du texte « 1 m », doit être positionnée en bas à droite dans la « *Grande Vue* » et elle doit être visible quel que soit le zoom et le « *scroll* ».

Sa mise en place se fera dans une fonction de la classe *GrandeVue* qui redéfinira la méthode `drawForeground()` de *QGraphicsView*, cf. p.8 du cours *Qt4*.

Après ajout de ces trois fonctionnalités, voici le rendu de la « *Grande Vue* »dans un état « zoomé » :