Débuter dans la création d'interfaces graphiques avec Qt 4

Traduction du tutoriel pour debutant de Nokia / Qt Software

Par qt.developpez.com - dourouc05

Date de publication : 18 janvier 2009

Ce tutoriel fournit une introduction à la programmation d'interfaces graphiques (IHM, Interface Homme-Machine) utilisant la boîte à outils Qt. Il ne couvre pas tout ; l'accent est mis sur la philosophie de la programmation IHM, et les fonctionnalités de Qt sont introduites au fur et à mesure. Certaines fonctionnalités, qui sont souvent utilisées, ne sont pas couvertes par ce tutoriel d'introduction à Qt.

Ce tutoriel est une traduction d'un des tutoriels écrits par **Nokia Corporation and/or its subsidiary(-ies)** incluse la documentation Qt 4.4.x en anglais. Les éventuels problèmes résultant d'une mauvaise traduction ne sont pas imputables à Nokia ou Qt Software. Vous pouvez trouver le tutoriel original **ici**.
Nous remercions Nokia et Qt Software pour leur confiance tout au long de ce travail de traduction.

Pour toutes questions ou remarques retouvez ce tutoriel sur le forum

0 - Remerciements

Voici terminée la première traduction de l'équipe Qt de developpez.com. Nous espérons tous que débutants et non débutants trouveront ici une base solide pour leur apprentissage et compréhension de ce Framework.

Merci à **Alp Mestan** et **Yan Verdavaine** pour l'encadrement et la mise en place de ce projet.

Merci tout particulier à **dourouc05** et **difredo** pour leur participation et relecture tout au long de la traduction

Et surtout **félicitation et un grand bravo** aux acteurs principaux de cette traduction. Par chapitre :

- Tutoriaux Qt : **John42**
- Hello World : **haraelendil**
- Terminer proprement l'application : **Spout**
- Valeurs familières : **Spout**
- Laissons les être des widgets : **Spout**
- Les composants : **amoweb**
- Les composants en abondance : **buggen25**
- Une chose mène à une autre : **buggen25**
- Préparation à la guerre : **buggen25**
- Avec un canon, vous pouvez : **Spout**
- Lisse comme la soie : **buggen25**
- Tirons : **amoweb**
- Accrochons dans l'air des briques : **Spout**
- Game over : **dourouc05**
- Face au mur : **signix**

00 - Présentation

Ce tutoriel fournit une introduction à la programmation d'interfaces graphiques (IHM, Interface Homme-Machine) utilisant la boîte à outils Qt. Il ne couvre pas tout ; l'accent est mis sur la philosophie de la programmation IHM, et les fonctionnalités de Qt sont introduites au fur et à mesure. Certaines fonctionnalités, qui sont souvent utilisées, ne sont pas couvertes par ce tutoriel d'introduction à Qt

Le premier chapitre commence avec un programme minimal de type «hello world» et les suivants introduisent de nouveaux concepts. Jusqu'au chapitre 14, le «hello world» du premier chapitre va se transformer en un jeu de 650 lignes de code.

Si vous débutez complètement avec Qt, commencez par lire **"Comment apprendre Qt"** si vous ne l'avez pas déjà fait.

Le code source du tutoriel est situé dans le répertoire examples/tutorials/tutorial de Qt-4.4.x
Pour les autres versions, vous pouvez télécharger les sources ici :

Source **tutorial.rar**

Chapitres :

Ce petit jeu ne ressemble pas beaucoup aux applications IHM modernes. Il utilise quelques techniques IHM, mais après avoir bien travaillé avec lui, nous vous recommandons de jeter un oeil à l'exemple de l'**Application**, lequel présente une petite application IHM, avec des menus, des barres d'outils, une barre de statut, et bien d'autres encore.

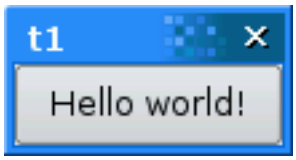
I - Hello Word

Source t1.rar

Fichiers

- tutorials/tutorial/t1/main.cpp
- tutorials/tutorial/t1/t1.pro

Ce premier programme est un simple exemple de Hello World. Il contient juste le minimum pour créer une application Qt. Voici un aperçu du programme:



Et voilà le code source complet de l'application:

```
#include <QApplication>
#include <QPushButton>

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

    QPushButton hello("Hello world!");

    hello.show();
    return app.exec();
}
```

I-A - Analyse du code ligne par ligne

```
#include <Qapplication>
```

Cette ligne inclut la définition de la classe QApplication. Il doit y avoir une instance de QApplication dans chaque application qui possède une IHM avec Qt. QApplication qui gère différentes ressources au niveau de l'application, comme la police de caractère par défaut ou le curseur.

```
#include <QPushButton>
```

Cette ligne inclut la définition de la classe QPushButton. Pour chaque classe faisant partie de l'API publique de Qt, il y a un fichier d'en-tête de même nom contenant sa définition. QPushButton est un bouton d'interface graphique que l'utilisateur peut presser et relâcher. Il gère lui même son propre *look and feel*, comme n'importe quel QWidget. Un widget est un objet de l'interface graphique qui peut gérer une entrée de l'utilisateur et dessiner. Le programmeur peut soit changer tout le *look and feel* du widget comme seulement un partie de celui-ci (la couleur, par exemple), ainsi que son contenu. Un QPushButton peut afficher, soit du texte, soit un QIcon.

```
int main(int argc, char *argv[])
{
```

La fonction *main()* est le point d'entrée du programme. Presque tout le temps en utilisant Qt, cette fonction n'a besoin de faire que quelques initialisations avant de passer la main à la bibliothèque Qt , qui informe ensuite le programme

des actions utilisateur via des évènements. Le paramètre *argc* est le nombre d'arguments de la ligne de commande et *argv* est le tableau contenant les arguments de la ligne de commande. C'est une propriété standard du C++.

```
QApplication app(argc, argv);
```

app est l'instance de QApplication de ce programme. Elle est créée ici. Les arguments *argc* et *argv* sont passés au constructeur de QApplication pour qu'il puisse traiter certains arguments de ligne de commande standard (comme -display sous X11). Tous les arguments de ligne de commande reconnus par Qt sont enlevés de *argv*, et *argc* est décrémenté en conséquence. Voir la documentation de *QApplication::arguments()* pour plus de détails. L'instance de QApplication doit-être créée avant l'utilisation de n'importe quelle fonctionnalité d'IHM de Qt.

```
QPushButton hello("Hello world!");
```

Après la création de la QApplication, voilà la première ligne liée à l'IHM : un bouton est créé. Le bouton affiche le texte "Hello world!". Comme on ne spécifie pas de fenêtre parente (comme deuxième argument du constructeur de QPushButton), le bouton sera lui-même une fenêtre, avec son propre cadre et sa barre de titre. La taille du bouton est déterminée par défaut. On pourrait appeler *QWidget::move()* pour assigner un emplacement spécifique au widget, mais nous laissons ici le système choisir une position.

```
hello.show();
```

Un widget n'est jamais visible quand on le crée. Il faut appeler *QWidget::show()* pour le rendre visible.

```
    return app.exec();
}
```

C'est ici que le *main()* passe le contrôle à Qt. *QCoreApplication::exec()* finira en même temps que l'application. QCoreApplication est une classe de base de QApplication. Elle implémente les fonctionnalités non liées à l'IHM et peut être utilisée pour le développement d'applications sans IHM. Dans *QCoreApplication::exec()*, Qt reçoit et traite les évènements système et les renvoie aux widgets appropriés. Vous devriez maintenant compiler et lancer le programme.

I-B - Les variables d'environnement

Si vous développez des applications Qt à la ligne de commande, vous devez vous assurer que les librairies et les exécutables de Qt sont accessibles à votre environnement en ajoutant le chemin du répertoire *bin* de Qt à votre variable *PATH*. Cette opération est décrite dans les instructions d'installation de votre plate-forme. Sous Windows, ceci est automatique si vous utilisez la fenêtre de commande depuis le menu Démarrer > (Tous les) Programmes > Qt. Si vous utilisez la fenêtre de commande depuis Démarrer > Exécuter > *command* ou *cmd*, vous devrez paramétrer la variable *PATH* vous même.

I-C - Compiler l'application

Les exemples de ce tutoriel sont situés dans le répertoire *examples/tutorials/tutorial* de Qt. Si vous avez installé Qt à partir d'un exécutable, les exemples ont également été installés. Si vous avez compilé Qt vous même, les exemples ont été compilés en même temps. Dans les deux cas, il y a beaucoup à apprendre sur comment utiliser Qt en modifiant et en recompilant vous même ces exemples. En assumant que vous ayez copié le(s) fichier(s) *.cpp* et *.h* d'un exemple dans un autre dossier pour y apporter vos modifications, la prochaine étape est de créer un *makefile* pour Qt dans ce répertoire. Pour créer un *makefile* pour Qt, il faut utiliser la commande *qmake*, un outil de compilation fourni avec Qt. Lancer les deux commandes suivantes dans le répertoire contenant vos sources modifiées pour créer un *makefile* : **qmake -project** puis **qmake** La première commande ordonne a qmake de créer un fichier de projet (*.pro*). La seconde commande ordonne à *qmake* d'utiliser le fichier *.pro* pour créer un *makefile* adapté à la plateforme et au compilateur.

Maintenant vous n'avez qu'à lancer *make* (*nmake* si vous êtes sous Visual Studio) pour compiler le programme, et vous pouvez ensuite lancer votre première application Qt!

I-D - Lancer l'application

Quand vous lancerez cet exemple, vous verrez une petite fenêtre affichant un seul bouton. Sur le bouton, vous pourrez lire le fameux: "Hello world!".

I-E - Exercices

Essayez de dimensionner la fenêtre. Si vous êtes sous X11, essayez de lancer le programme avec l'option -geometry (par exemple, -geometry 100x200+10+20).

II - Terminer proprement l'application

Source t2.rar

Fichiers

- tutorials/tutorial/t2/main.cpp
- tutorials/tutorial/t2/t2.pro

Cet exemple est une extension de la fenêtre créée dans le chapitre 1. Nous allons maintenant continuer afin de terminer proprement l'application sur requête de l'utilisateur.

Nous allons également utiliser une police plus attrayante que celle par défaut.

```
 #include <QApplication>
#include <QFont>
#include <QPushButton>

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

    QPushButton quit("Quit");
    quit.resize(75, 30);
    quit.setFont(QFont("Times", 18, QFont::Bold));

    QObject::connect(&quit, SIGNAL(clicked()), &app, SLOT(quit()));

    quit.show();
    return app.exec();
}
```

II-A - Analyse du code ligne par ligne

```
 #include <QFont>
```

Comme ce programme utilise la classe QFont, **<QFont>**a besoin d'être inclus.

```
 QPushButton quit("Quit");
```

Cette fois, le bouton affiche **Quit** et c'est exactement ce que le programme va faire quand l'utilisateur cliquera dessus.

```
 quit.resize(75, 30);
```

Comme le texte est un peu plus court que "Hello World!", nous avons choisi une autre taille pour le bouton. Nous aurions également pu utiliser QFontMetrics pour configurer la taille ou laisser QPushButton en choisir une raisonnable par défaut.

```
 quit.setFont(QFont("Times", 18, QFont::Bold));
```

Ici nous choisissons une nouvelle police pour le bouton de la famille Times, de taille 18 et avec le style gras.

```
QObject::connect(&quit, SIGNAL(clicked()), &app, SLOT(quit()));
```

QObject::connect() est peut-être la fonction la plus indispensable de Qt. Notez que la méthode *connect()* est statique dans QObject. Ne la confondez pas avec la méthode *connect()* de la librairie de sockets Berkeley.

Cet appel à *connect()* établit une connexion à sens unique entre deux objets Qt (objets qui héritent de QObject, directement ou indirectemment). Tous les objets Qt peuvent avoir des signaux (pour envoyer des messages) et des slots (pour en recevoir). Tous les widgets sont des objets Qt car ils héritent de QWidget, qui hérite elle-même de QObject.

Ici, le signal *clicked()* de *quit* est connecté au slot *quit()* de *app*, donc, quand on clique sur le bouton, l'application se termine.

La documentation des signaux et des slots décrit ce sujet en détail.

II-B - Lancer l'application

Quand vous lancez ce programme, vous voyez un bouton dans une fenêtre encore plus petite que dans le chapitre 1.

Consultez le chapitre 1 pour savoir comment créer un makefile et compiler l'application.

II-C - Exercices

Essayez de redimensionner la fenêtre. Cliquez sur le bouton pour fermer l'application.

Y-a-t'il d'autres signaux dans QPushButton que l'on puisse connecter pour quitter? [indice: QPushButton hérite la plupart de ses fonctionnalités de QAbstractButton]

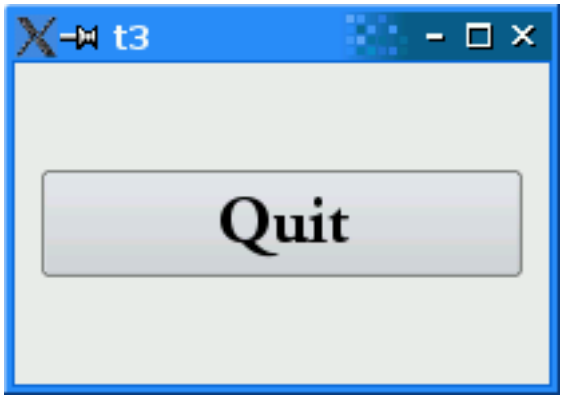
III - Les valeurs familières

Source t3.rar

Fichiers

- tutorials/tutorial/t3/main.cpp
- tutorials/tutorial/t3/t3.pro

Cet exemple montre comment créer des widgets parents et enfants.



Nous allons faire simple et utiliser juste un parent avec un seul enfant.

```
#include <QApplication>
#include <QFont>
#include <QPushButton>
#include <QWidget>

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

    QWidget window;
    window.resize(200, 120);

    QPushButton quit("Quit", &window);
    quit.setFont(QFont("Times", 18, QFont::Bold));
    quit.setGeometry(10, 40, 180, 40);
    QObject::connect(&quit, SIGNAL(clicked()), &app, SLOT(quit()));

    window.show();
    return app.exec();
}
```

III-A - Analyse du code ligne par ligne

```
#include <QWidget>
```

Nous avons une inclusion de QWidget pour utiliser la classe de base des widgets.

```
QWidget window;
```

Ici nous créons simplement un objet widget, pur et simple. La classe QWidget est la classe de base de tous les objets d'interface utilisateur (IU, IHM ; UI et GUI, respectivement, en anglais). Le widget est le noyau de l'interface utilisateur. Il reçoit les évènements de la souris, du clavier ou tout autre issu du système de fenêtrage et dessine une représentation de lui-même à l'écran. Un widget est fixé par son parent et le widget devant lui.

Un widget qui n'est pas intégré dans un widget parent, comme ce widget particulier, est appelé *window*. Habituellement, les windows ont leur propre frame et barre des tâches, fournies par le système de fenêtrage. Un widget sans widget parent est toujours comme un window indépendant. Sa position initiale sur l'écran est contrôlée par le système de fenêtrage.

```
window.resize(200, 120);
```

On fixe la largeur de window à 200 pixels et sa hauteur à 120 pixels.

```
QPushButton quit("Quit", &window);
```

Un enfant est né. Ce QPushButton est créé avec un widget parent (window). Un widget enfant est toujours affiché dans la zone de son parent. Quand il est affiché, il est fixé par les limites de son parent. Par défaut, il est ancré dans le coin supérieur gauche de son parent, à la position (0,0).

```
quit.setGeometry(10, 40, 180, 40);
```

La fonction *QWidget::setGeometry()* prend quatre arguments : les deux premiers sont les coordonnées x et y du coin supérieur gauche du bouton. Les coordonnées sont relatives au widget parent. Les deux derniers arguments sont la largeur et la hauteur du bouton. Le résultat est un bouton qui va de la position (10,40) à la position (190,80).

```
window.show();
```

Quand un widget parent est affiché, la méthode *show* de chacun de ses enfant va être appelé (sauf pour ceux qui ont été explicitement cachés par *QWidget::hide()*).

III-B - Pour être conscient de l'ordre des choses

Les lecteurs de ce tutoriel qui ont déjà étudié QObject vont se souvenir que lorsque le destructeur d'un QObject est appelé, si le QObject a un enfant, son destructeur appelle automatiquement le destructeur de chaque enfant. Il peut donc arriver que le destructeur de QPushButton quit soit appelé deux fois à la fin du main().D'abord quand son parent, window, sort du scope et que le destructeur supprime quit parce que c'est un enfant, puis une seconde fois quand quit lui-même sort du scope. Mais il n'y a pas besoin de s'inquiéter de cela; ce code est correct. Toutefois, *il y a* un cas où l'on doit être conscient de l'ordre de destruction des objets de la pile. Pour plus d'explications, voir cette note sur l'ordre de construction/destruction des QObjects : http://doc.trolltech.com/4.4/objecttrees.html#note-on-the-order-of-construction-destruction-of-qobjects

III-C - Lancer l'application

Le bouton ne remplit plus la fenêtre entière. A la place, il reste à la position (10,40) dans la fenêtre avec une taille (180,40), grâce à l'appel à *QWidget::setGeometry()*.

III-D - Exercices

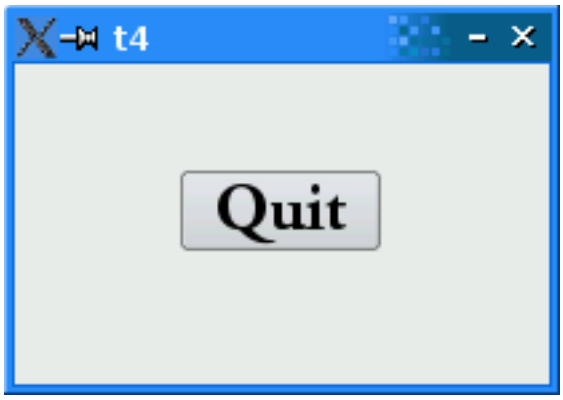
Essayez de redimensionner la fenêtre. Comment se comporte le bouton? Que se passe-t-il pour la hauteur du bouton si vous lancez le programme avec une police plus grosse? Que se passe-t-il si vous essayez de faire une fenêtre toute petite?

IV - Soyons Widgets

Source t4.rar

Fichiers

- tutorials/tutorial/t4/main.cpp
- tutorials/tutorial/t4/t4.pro



Cet exemple montre comment créer son propre widget, et décrit comment en contrôler les tailles minimum et maximum.

```
#include <QApplication>
#include <QFont>
#include <QPushButton>
#include <QWidget>

class MyWidget : public QWidget
{
public:
    MyWidget(QWidget *parent = 0);
};

MyWidget::MyWidget(QWidget *parent)
    : QWidget(parent)
{
    setFixedSize(200, 120);

    QPushButton *quit = new QPushButton(tr("Quit"), this);
    quit->setGeometry(62, 40, 75, 30);
    quit->setFont(QFont("Times", 18, QFont::Bold));

    connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
}

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    MyWidget widget;
    widget.show();
    return app.exec();
}
```

IV-A - Analyse du code ligne par ligne

```
class MyWidget : public QWidget
{
public:
    MyWidget(QWidget *parent = 0);
```

```
};
```

Ici nous créons une nouvelle classe. Comme cette class hérite de QWidget, la nouvelle classe est un widget et peut être une fenêtre de premier plan ou un widget enfant (comme le QPushButton du chapitre précédent).

Cette classe possède un seul membre, un constructeur (en plus des membres qu'elle hérite de QWidget). Le constructeur est un constructeur d'un widget standard de Qt ; vous devez toujours inclure un constructeur similaire quand vous créez des widgets.

L'argument est son widget parent. Pour créer une fenêtre de premier plan vous devez spécifier un pointeur NULL comme parent. Comme vous pouvez le voir, ce widget est une fenêtre de premier plan par défaut.

```
MyWidget::MyWidget(QWidget *parent)
```

L'implémentation du constructeur commence ici. Comme la plupart des widgets, il ne fait que passer le parent au constructeur de QWidget.

```
  : QWidget(parent)
{
    setFixedSize(200, 120);
```

Comme ce widget ne sait pas comment gérer le redimmensionnement, nous fixons sa taille. Dans le prochain chapitre, nous verrons comment un widget peut répondre à un évènement de redimmensionnement issu de l'utilisateur.

```
    QPushButton *quit = new QPushButton(tr("Quit"), this);
    quit->setGeometry(62, 40, 75, 30);
    quit->setFont(QFont("Times", 18, QFont::Bold));
```

Ici nous créons et configurons un widget dans de ce widget (le parent du nouveau widget est *this*, donc l'instance de MyWidget).

L'appel à la fonction *tr()* pour le texte "Quit" le marque pour la traduction, rendant alors possible son remplacement à l'exécution à partir du contenu d'un fichier de traduction. C'est une bonne habitude d'utiliser *tr()* pour tous les textes visibles par l'utilisateur, dans le cas où vous décidez plus tard de traduire votre application dans d'autres langues.

Notez que quit est une variable locale dans le constructeur. MyWidget ne conserve aucune trace d'elle : Qt le fait, et le supprimera automatiquement quand l'objet MyWidget sera supprimé. C'est pourquoi MyWidget n'a pas besoin de destructeur (d'un autre côté, il n'y a aucun problème à supprimer un enfant quand vous le décidez : l'enfant informera automatiquement Qt de sa mort imminente).

L'appel à *QWidget::setGeometry()* configure à la fois la position du widget à l'écran et sa taille. Cela revient à appeler *QWidget::move()* suivi de *QWidget::resize()*.

```
    connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
}
```

Le pointeur *qApp* est une variable globale déclaré dans le fichier d'entête <*QApplication*>. Il pointe vers l'instance unique *QApplication* de l'application.

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    MyWidget widget;
    widget.show();
    return app.exec();
}
```

Ici nous instancions notre nouvel enfant, nous l'affichons et nous lançons l'application.

IV-B - Lancer l'application

Ce programme a un comportement similaire au précédent. La différence réside dans la manière dont nous l'avons implémenté. Il agit toutefois légèrement différemment. Essayez juste de le redimmensionner pour voir.

IV-C - Exercices

Essayez de créer un autre objet *MyWidget* dans la fonction *main()*. Que se passe-t-il?

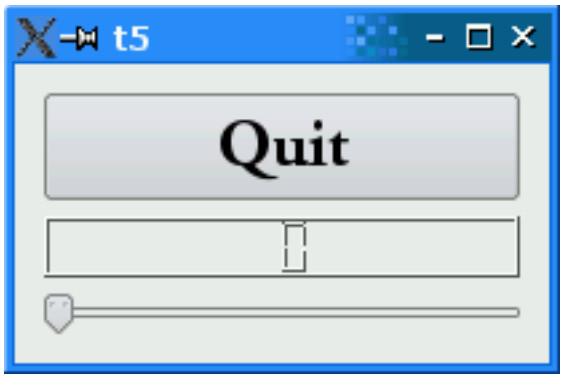
Essayez d'ajouter plus de boutons ou mettez d'autres widgets que QPushButton.

V - Les composants

Source t5.rar

Fichiers

- tutorials/tutorial/t5/main.cpp
- tutorials/tutorial/t5/t5.pro



Cet exemple montre comment créer et connecter des widgets en utilisant les signaux et les slots et gérer le redimensionnement.

```
#include <QApplication>
#include <QFont>
#include <QLCDNumber>
#include <QPushButton>
#include <QSlider>
#include <QVBoxLayout>
#include <QWidget>

class MyWidget : public QWidget
{
public:
    MyWidget(QWidget *parent = 0);
};

MyWidget::MyWidget(QWidget *parent)
    : QWidget(parent)
{
    QPushButton *quit = new QPushButton(tr("Quit"));
    quit->setFont(QFont("Times", 18, QFont::Bold));

    QLCDNumber *lcd = new QLCDNumber(2);
    lcd->setSegmentStyle(QLCDNumber::Filled);

    QSlider *slider = new QSlider(Qt::Horizontal);
    slider->setRange(0, 99);
    slider->setValue(0);

    connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
    connect(slider, SIGNAL(valueChanged(int)),
            lcd, SLOT(display(int)));

    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(quit);
    layout->addWidget(lcd);
    layout->addWidget(slider);
    setLayout(layout);
}

int main(int argc, char *argv[])
```

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

V-A - Analyse du code ligne par ligne

```
class MyWidget : public QWidget
{
public:
    MyWidget(QWidget *parent = 0);
};

MyWidget::MyWidget(QWidget *parent)
    : QWidget(parent)
{
    QPushButton *quit = new QPushButton(tr("Quit"));
    quit->setFont(QFont("Times", 18, QFont::Bold));

    QLCDNumber *lcd = new QLCDNumber(2);
    lcd->setSegmentStyle(QLCDNumber::Filled);
```

lcd est un *QLCDNumber*, un widget qui affiche un nombre comme un écran LCD. Cette instance est prévue pour afficher deux chiffres. Mettre la propriété *QLCDNumber::segmentStyle* à *QLCDNumber::Filled* rend le LCD plus lisible.

Anecdote : *QLCDNumber* est le premier widget écrit pour Qt, à l'époque *QPainter* ne supportait qu'une méthode de dessin : *drawLine()*. La première version de l'exemple Tetrix, qui utilisait *QLCDNumber* pour afficher les scores, a été écrite à ce moment.

```
    QSlider *slider = new QSlider(Qt::Horizontal);
    slider->setRange(0, 99);
    slider->setValue(0);
```

L'utilisateur utilise le widget *QSlider* pour changer une valeur entière comprise dans la gamme (*range*, en anglais). Ici nous en créons un, sa valeur minimum est 0 et son maximum 99. Sa valeur initiale est 0.

```
    connect(slider, SIGNAL(valueChanged(int)),
    lcd, SLOT(display(int)));
```

Ici nous connectons le signal *valueChanged()* du widget *slider* au slot *display()* de l'écran LCD. À chaque fois que la valeur du *slider*, change il envoie la nouvelle valeur en émettant le signal *valueChanged()*. Comme le slot *display()* de *lcd* est connecté à *valueChanged()* de *slider*, il est appelé à chaque fois que l'utilisateur change la valeur.

Les widgets ne se connaissent pas l'un l'autre. C'est une caractéristique essentielle de la programmation objet. Les slots sont des fonctions membres normales du C++ et suivent les règles normales d'accès du C++.

```
    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(quit);
    layout->addWidget(lcd);
    layout->addWidget(slider);
    setLayout(layout);
```

MyWidget utilise *QVBoxLayout* pour gérer la taille et la position (*geometry*) de ses widgets enfants. Nous n'avons donc pas besoins de spécifier les coordonnées de chaque widget comme nous le faisions au chapitre 4. De plus

utiliser un layout assure le redimensionnement des widgets quand la fenêtre change de taille. Nous ajoutons les widgets *quit*, *lcd* et *slider* au layout, en utilisant *QBoxLayout::addWidget()*.

La fonction *QWidget::setLayout()* installe le layout sur la fenêtre *MyWidget*. Cela rend les layout enfant de *MyWidget*. De cette manière, nous n'aurons pas à nous inquiéter de la suppression du layout ; le lien de parenté assure sa suppression en même temps que *MyWidget*. L'appel de la méthode *QWidget::setLayout()* rend automatiquement les widgets dans le layout enfant de *MyWidget*. Nous n'avons donc pas besoin de spécifier les liens de parentés.

Avec Qt, un widget est soit enfant d'un autre widget (ex. *this*), soit orphelin. Un widget peut être ajouté à un layout, dans ce cas le layout devient responsable des dimensions (geometry) du widget, mais le layout ne peut pas être parent. En effet, le constructeur de QWidget prend un héritier de QWidget comme parent et QLayout n'hérite pas de QWidget.

V-B - Lancer l'application

L'écran LCD montre bien la position du *slider*, et le widget se redimensionne bien. Notez que l'écran LCD change de taille quand la fenêtre est redimensionnée, mais les autres gardent à peu près la même.

V-C - Exercices

Essayez d'ajouter plus de chiffres sur l'écran LCD ou essayer de changer la base (*QLCDNumber::setMode()*). Vous pouvez même ajouter quatre boutons pour changer la base.

Vous pouvez aussi changer la gamme du *slider*.

Peut-être qu'il aurait été mieux d'utiliser *QSpinBox* plutôt que le *slider* ?

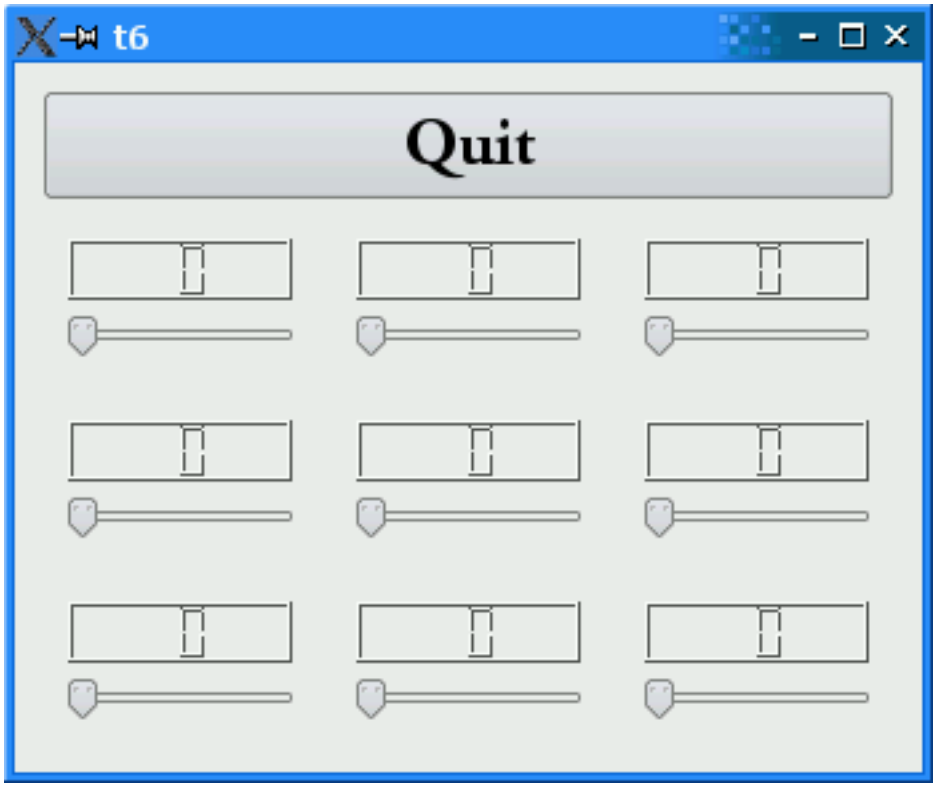
Essayer de faire quitter l'application quand l'écran LCD dépasse sa capacité.

VI - Construire des widgets en vrac!

Source t6.rar

Fichiers

- tutorials/tutorial/t6/main.cpp




```
#include <QApplication>
 #include <QFont>
 #include <QGridLayout>
 #include <QLCDNumber>
 #include <QPushButton>
 #include <QSlider>
 #include <QVBoxLayout>
 #include <QWidget>

 class LCDRange : public QWidget
 {
 public:
     LCDRange(QWidget *parent = 0);
 };

 LCDRange::LCDRange(QWidget *parent)
     : QWidget(parent)
 {
     QLCDNumber *lcd = new QLCDNumber(2);
     lcd->setSegmentStyle(QLCDNumber::Filled);

     QSlider *slider = new QSlider(Qt::Horizontal);
     slider->setRange(0, 99);
     slider->setValue(0);
     connect(slider, SIGNAL(valueChanged(int)),
             lcd, SLOT(display(int)));

     QVBoxLayout *layout = new QVBoxLayout;
     layout->addWidget(lcd);
```

```
    layout->addWidget(slider);
    setLayout(layout);
}

class MyWidget : public QWidget
{
public:
    MyWidget(QWidget *parent = 0);
};

MyWidget::MyWidget(QWidget *parent)
    : QWidget(parent)
{
    QPushButton *quit = new QPushButton(tr("Quit"));
    quit->setFont(QFont("Times", 18, QFont::Bold));
    connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));

    QGridLayout *grid = new QGridLayout;
    for (int row = 0; row < 3; ++row) {
        for (int column = 0; column < 3; ++column) {
            LCDRange *lcdRange = new LCDRange;
            grid->addWidget(lcdRange, row, column);
        }
    }

    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(quit);
    layout->addLayout(grid);
    setLayout(layout);
}

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    MyWidget widget;
    widget.show();
    return app.exec();
}
```

VI-A - Analyse du code ligne par ligne

```
class LCDRange : public QWidget
{
public:
    LCDRange(QWidget *parent = 0);
};
```

Le widget LCDRange est un widget sans la moindre méthode : il possède juste un constructeur. Ce genre de widget n'est pas très utile, mais on rajoutera des méthodes plus tard.

```
LCDRange::LCDRange(QWidget *parent)
    : QWidget(parent)
{
    QLCDNumber *lcd = new QLCDNumber(2);
    lcd->setSegmentStyle(QLCDNumber::Filled);

    QSlider *slider = new QSlider(Qt::Horizontal);
    slider->setRange(0, 99);
    slider->setValue(0);
    connect(slider, SIGNAL(valueChanged(int)),
            lcd, SLOT(display(int)));

    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(lcd);
    layout->addWidget(slider);
    setLayout(layout);
}
```

Ce code est prélevé du constructeur MyWidget du Chapitre 5. La seule différence réside dans le faite que le bouton Quit est délaissé et que la classe a été renommée.

```
class MyWidget : public QWidget
 {
 public:
     MyWidget(QWidget *parent = 0);
};
```

MyWidget aussi ne contient aucune méthode, elle ne contient qu'un constructeur.

```
MyWidget::MyWidget(QWidget *parent)
     : QWidget(parent)
 {
     QPushButton *quit = new QPushButton(tr("Quit"));
     quit->setFont(QFont("Times", 18, QFont::Bold));
     connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
```

Le bouton qui avait l'habitude d'être dans ce qu'on appelle maintenant LCDRange a été séparé de manière à avoir un seul bouton Quit et plusieurs objets LCDRange.

```
    QGridLayout *grid = new QGridLayout;
```

On crée un QWidget avec un QGridLayout qui va contenir trois colonnes.

QGridLayout arrange automatiquement ses widgets en lignes et colonnes, vous pouvez spécifier les numéros des lignes et des colonnes quand vous ajoutez des widgets à l'objet QGridLayout, qui va les arranger sur la grille.

```
      for (int row = 0; row < 3; ++row) {
         for (int column = 0; column < 3; ++column) {
             LCDRange *lcdRange = new LCDRange;
             grid->addWidget(lcdRange, row, column);
         }
     }
```

On crée neuf objets LCDRange, qui sont tous des widgets enfants de l'objet grid, et qu'on arrange sur une grille de trois lignes et de trois colonnes.

```
     QVBoxLayout *layout = new QVBoxLayout;
     layout->addWidget(quit);
     layout->addLayout(grid);
     setLayout(layout);
```

Au final, on rajoute le bouton Quit et la grille de disposition grid contenant les objets LCDRange à l'outil de disposition principal. La fonction QWidget::addLayout() est similaire à la fonction QWidget::addWidget(), et permet d'ajouter un objet de disposition enfant à l'outil de disposition principal.

C'est tout.

VI-B - Lancer l'application

Ce programme montre la facilité d'utilisation de plusieurs widgets à la fois. Chacun se comporte comme un slider et un nombre digital comme dans le chapitre précédent. Une fois encore, la différence réside dans l'implémentation.

VI-C - Exercices

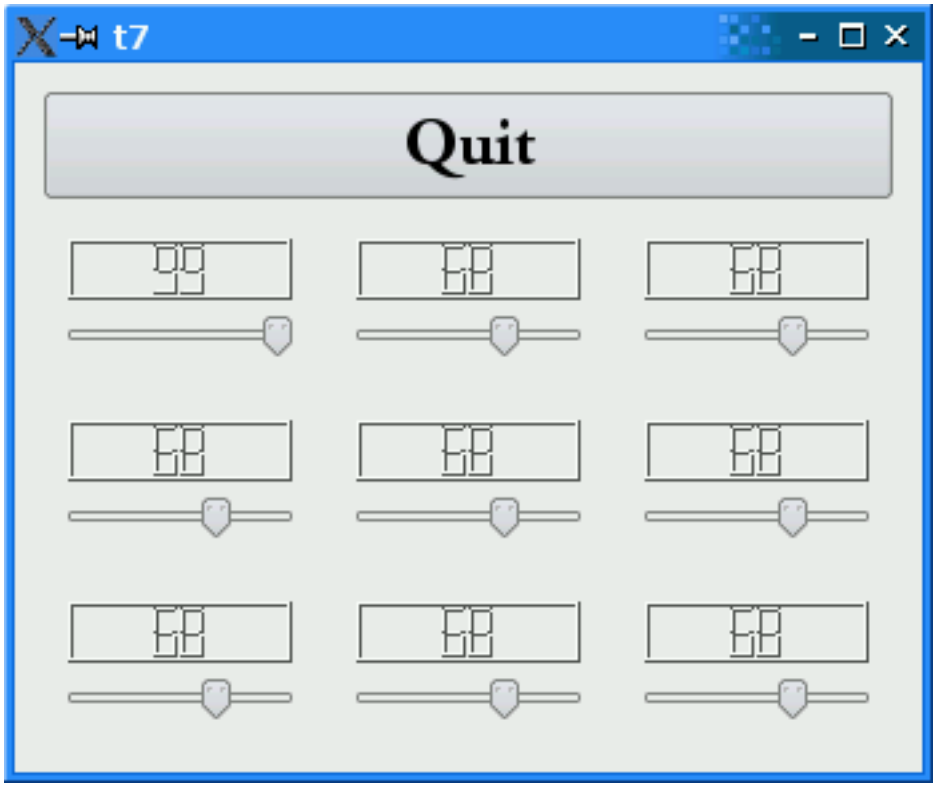
Initialiser chaque slider avec valeur différente et/ou aléatoire au lancement.

VII - Une chose nous mène vers autre chose

Source t7.rar

Fichiers

- tutorial/t7/lcdrange.cpp
- tutorial/t7/lcdrange.h
- tutorial/t7/main.cpp



Cet exemple montre comment créer des widgets personnalisés avec des signaux et des slots, et comment les connecter d'une manière plus complexe. Pour la première fois, la source est divisée en plusieurs fichiers que nous avons placé dans le dossier **exemples/tutorial/t7**.

VII-A - Analyse du code ligne par ligne

- t7/lcdrange.h

```
#ifndef LCDRANGE_H
#define LCDRANGE_H
```

Ceci, combiné avec **#endif** à la fin du fichier, est une construction standard du langage C++ qui empêche les erreurs dans le cas où le fichier entête venait à être inclus plusieurs fois. Si vous n'utilisez pas encore cette notation, c'est une bonne habitude à prendre pour développer.

```
#include <QWidget>
```

<QWidget> est inclus du moment que notre LCDRange hérite de **QWidget**. On doit toujours inclure l'entête de la classe parente - on a un peu triché sur les tutoriaux précédents, en incluant le fichier <QWidget> d'une manière indirecte via d'autres fichiers entête.

```
class QSlider;
```

Ceci est une autre astuce classique, qui est moins utilisée que la précédente. On n'a pas besoin de **QSlider** dans l'interface de la classe, on l'utilisera dans l'implémentation. On utilise uniquement la déclaration de la classe **QSlider** dans le fichier entête, et on utilise le fichier entête de **QSlider** dans le fichier .cpp.

Ceci accélère la compilation de grands projets, car la compilateur passe la plupart de son temps à parcourir les fichiers entête, mais ce n'est pas le cas du code actuel. Cette seule astuce augmente la vitesse de compilation d'un facteur de deux ou plus.

```
class LCDRange : public QWidget
 {
     Q_OBJECT

 public:
     LCDRange(QWidget *parent = 0);
```

Noter Q_OBJECT. Cette macro doit être déclaré dans toutes les classes contenant des signaux et des slots. Si vous êtes curieux, elle définie toutes les fonctions implémentées dans le fichier **meta-object.**

```
int value() const;

 public slots:
     void setValue(int value);

 signals:
     void valueChanged(int newValue);
```

Ces trois membres créent une interface entre ce widget et d'autres composants du programme. Jusqu'à présent, LCDRange ne contenait aucun API.

value() est une méthode publique pour accéder a la valeur de LCDRange, setValue() est notre premier slot, et valueChanged() est notre premier signal custom.

Les slots doivent être implémenté de la manière habituelle (un slot est aussi une fonction membre C++). Les signaux sont automatiquement implémentés dans le fichier **meta-object**. Les signaux obéissent aux règles C++ des fonctions membres protégé (i.e., ils ne peuvent être émis que par la classe où ils sont déclarés et les classes dérivées de cette dernière).

Le signal valueChanged() est utilisé quand la valeur de LCDRange change.

- t7/lcdrange.cpp

```
LCDRange *previousRange = 0;

     for (int row = 0; row < 3; ++row) {
         for (int column = 0; column < 3; ++column) {
             LCDRange *lcdRange = new LCDRange;
             grid->addWidget(lcdRange, row, column);
             if (previousRange)
                 connect(lcdRange, SIGNAL(valueChanged(int)),
                         previousRange, SLOT(setValue(int)));
             previousRange = lcdRange;
         }
     }
```

Tout le main.cpp a été copié du chapitre précédent, à l'exception du constructeur de MyWidget. Quand on crée les neuf objets LCDRange, on les connecte avec le mécanisme des signaux et des slots. Chacun possède son propre signal valueChanged() connecté au slot setValue() du précédent. Le faite que LCDRange émette le signal valueChanged() au changement de sa valeur, engendre un enchaînement de signaux et de slots.

VII-B - Compiler l'application

Créer un fichier make pour une application multi fichiers ne diffère pas de la création d'un fichier make pour une application à fichier unique. Si vous avez sauvegardé vos fichiers dans un seul répertoire, tout ce qui vous reste à faire est :

```
qmake -project
qmake
```

La première commande donne l'ordre à qmake de créer un fichier .pro. La seconde lui ordonne de créer un fichier make (spécifique à la plateforme) basé sur le fichier projet. Vous devriez maintenant être capable d'exécuter make (ou nmake si vous utilisez Visual Studio) pour construire l'application.

VII-C - Exécuter l'application

Au lancement, le programme a la même apparence que dans le chapitre précédent. Essayez de manipuler le slider de coin inférieur droit.

VII-D - Exercices

Utiliser le slider du coin inférieur droit pour régler tous les LCD sur 50. Ensuite régler les six premiers sur 30 en cliquant sur le slider de la ligne en dessous. Maintenant utiliser le slider à gauche du dernier slider manipulé pour remettre les cinq premiers LCD à 50.

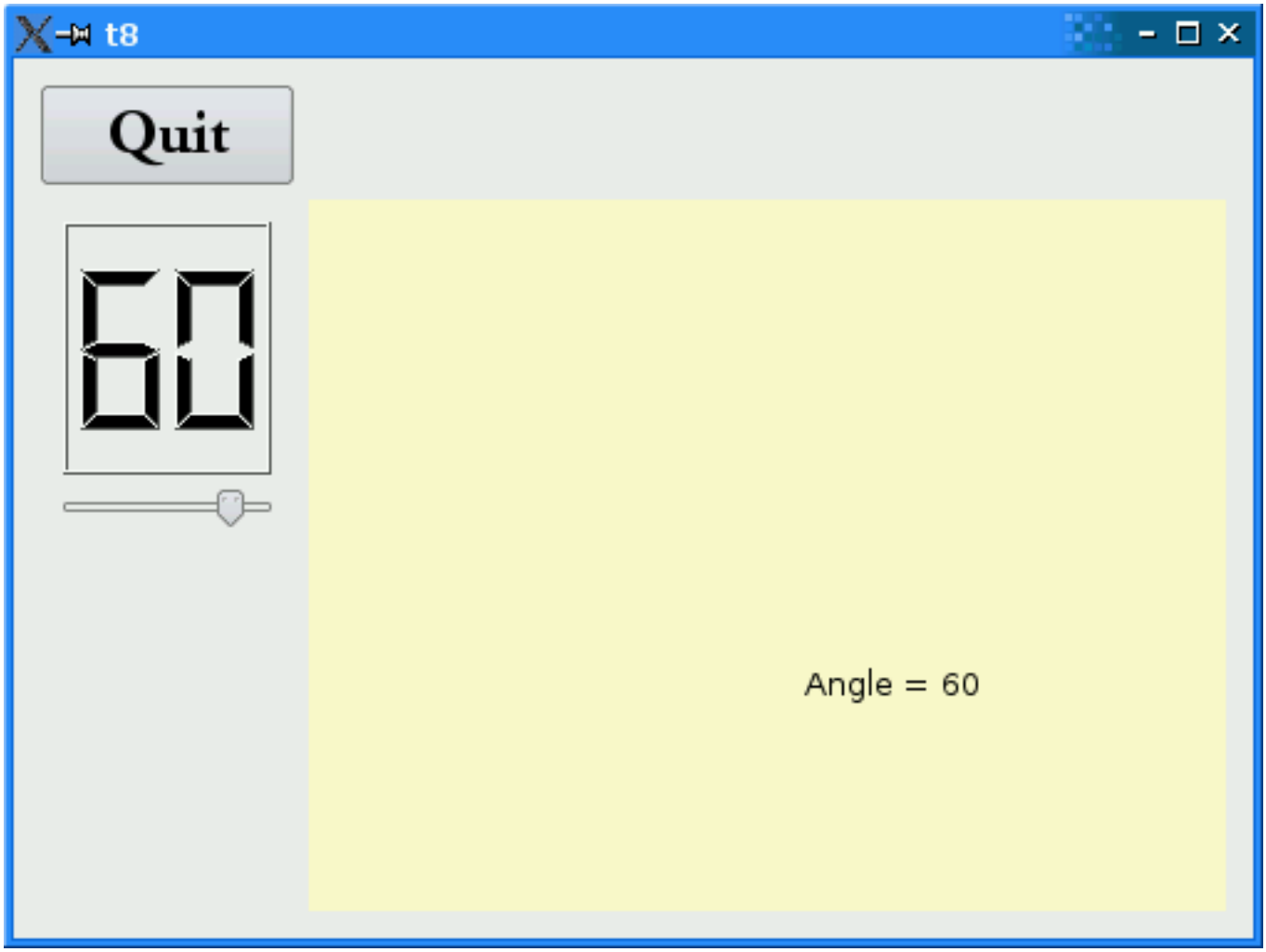
Maintenant cliquez sur la gauche du bouton du slider inférieur droit. Que se passe-t-il ? Pour quelle raison ce comportement est normal ?

VIII - Préparer le champ de bataille

Source t8.rar

Fichiers

- tutorial/t8/cannonfield.cpp
- tutorial/t8/cannonfield.h
- tutorial/t8/lcdrange.cpp
- tutorial/t8/lcdrange.h
- tutorial/t8/main.cpp



Dans cet exemple, on introduit le premier widget personnalisé qui se dessine par lui-même. On rajoutera aussi quelques interactions avec le clavier (en deux lignes de code).

VIII-A - Analyse du code ligne par ligne

- t8/lcdrange.h

Ce fichier est similaire au lcdrange.h dans le Chapitre 7. On a rajouté un seul slot setRange().

```
void setRange(int minValue, int maxValue);
```

On a maintenant la possibilité de régler l'intervalle **LCDRange**. Jusqu'à présent on fixait ce dernier de 0 à 99.

- t8/lcdrange.cpp

Il y a du changement dans le constructeur (on en discutera plus tard).

```
void LCDRange::setRange(int minValue, int maxValue)
 {
     if (minValue < 0 || maxValue > 99 || minValue > maxValue) {
         qWarning("LCDRange::setRange(%d, %d)\n"
                  "\tRange must be 0..99\n"
                  "\tand minValue must not be greater than maxValue",
                  minValue, maxValue);
         return;
     }
     slider->setRange(minValue, maxValue);
 }
```

Le slot setRange() fixe l'intervalle de slider dans **LCDRange**.Puisqu'on a réglé **QLCDNumber** pour afficher deux chiffres, on voudrait limiter les valeurs de **minValue** et **maxValue** pour empêcher le débordement sur **QLCDNumber** (on aurait pu autoriser des valeurs allant jusqu'à -9, mais on a choisi de ne pas le faire). Si les arguments sont illégaux, on utilise la fonction **qWarning()** pour afficher un avertissement à l'utilisateur et retourner immédiatement. **qWarning()** est une fonction semblable à printf qui, par défaut, envoie sa sortie vers **stderr**. Si vous voulez, vous pouvez utiliser votre propre gestionnaire d'erreur en utilisant la fonction **qInstallMsgHandler()**.

- t8/cannonfield.h

CannonField est un nouveau widget personnalisé qui s'affiche par lui-même.

```
class CannonField : public QWidget
 {
     Q_OBJECT

 public:
     CannonField(QWidget *parent = 0);
```

CannonField hérite de **QWidget**. On utilise le même idiome que pour **LCDRange**.

```
int angle() const { return currentAngle; }

 public slots:
     void setAngle(int angle);

 signals:
     void angleChanged(int newAngle);
```

Pour le moment, **CannonField** ne contient que la valeur d'un angle pour laquelle on fournit une interface avec le même idiome que pour la valeur de **LCDRange**.

```
protected:
     void paintEvent(QPaintEvent *event);
```

Ceci est le second gestionnaire d'événement qui fait partie d'une multitude de gestionnaires qu'on peut rencontrer dans **QWidget**. Cette fonction virtuelle est appelée quand un widget à besoin de se redessiner (c'est-à-dire peindre la surface du widget).

- t8/cannonfield.cpp

```
CannonField::CannonField(QWidget *parent)
     : QWidget(parent)
 {
```

Une fois encore, on utilise le même idiome que pour **LCDRange** dans le chapitre précédent.

```
   currentAngle = 45;
     setPalette(QPalette(QColor(250, 250, 200)));
     setAutoFillBackground(true);
 }
```

Le constructeur initialise l'angle à 45 degrés et crée une palette personnalisée pour ce widget.

Cette palette utilise la couleur indiquée comme arrière plan et choisie selon le besoin (pour ce widget, seules les couleurs d'arrière plan et de texte ont été utilisées). Ensuite, on appelle **setAutoFillBackground(true)** pour ordonner à Qt de remplir automatiquement l'arrière plan.

QColor est spécifié comme un triplet **RVB** (rouge-vert-bleu), où chaque valeur est comprise entre 0 (sombre) et 255 (intense). On aurait pu utiliser une couleur prédéfinie comme **Qt::yellow** au lieu de spécifier une valeur **RVB**.

```
void CannonField::setAngle(int angle)
 {
     if (angle < 5)
         angle = 5;
     if (angle > 70)
         angle = 70;
     if (currentAngle == angle)
         return;
     currentAngle = angle;
     update();
     emit angleChanged(currentAngle);
 }
```

Cette fonction définit la valeur de l'angle. On a choisi un intervalle compris entre 5 et 70 et on ajuste l'amplitude de l'angle spécifié en conséquence. On a choisi de ne pas afficher d'avertissement dans le cas où l'angle est en dehors de l'intervalle.

Si l'angle actuel est égal au précèdent, on retourne immédiatement. Il est important d'émettre le signal **angleChanged()** dans le cas ou l'angle change vraiment.

Ensuite, on définit le nouvel angle et on redessine le widget. La fonction **QWidget::update()** efface le widget (généralement, en le remplissant avec la couleur de l'arrière plan) et envoie un événement de dessin au widget. Le résultat est un appel à la fonction de l'événement de dessin de widget.

Au final, on émet le signal **angleChanged()** pour signaler au monde extérieur que l'angle a changé. Le mot clé **emit** est une syntaxe spécifique à Qt et qui n'appartient pas à la syntaxe C++ normalisée. En fait, c'est une macro.

```
void CannonField::paintEvent(QPaintEvent * /* event */)
 {
     QPainter painter(this);
     painter.drawText(200, 200,
                      tr("Angle = ") + QString::number(currentAngle));
 }
```

Ceci est notre première tentative de création d'un gestionnaire d'événement de dessin. L'argument contient des informations concernant cet événement, comme, par exemple, la région du widget qui doit être rafraîchie. Pour le moment, soyons paresseux et redessinons tout le widget.

Notre code affiche la valeur de l'angle à une certaine position dans le widget. Pour accomplir ceci, on crée un **QPainter**, qui opère sur le widget **CannonField**, que l'on utilise pour peindre une représentation littérale de la valeur de **currentAngle**. Nous reviendrons plus tard sur **QPainter**, et nous verrons qu'on peut faire des choses merveilleuses avec lui.

- t8/main.cpp

```
#include "cannonfield.h"
```

On inclut la définition de notre nouvelle classe.

```
class MyWidget : public QWidget
 {
 public:
     MyWidget(QWidget *parent = 0);
 };
```

La classe MyWidget va inclure un seul **LCDRange** et un **CannonField**.

```
LCDRange *angle = new LCDRange;
```

Dans le constructeur, on crée et initialise un widget **LCDRange**.

```
angle->setRange(5, 70);
```

On initialise **LCDRange** pour qu'il accepte des angles compris entre 5 et 70.

```
CannonField *cannonField = new CannonField;
```

On crée notre widget **CannonField**.

```
connect(angle, SIGNAL(valueChanged(int)),
             cannonField, SLOT(setAngle(int)));
     connect(cannonField, SIGNAL(angleChanged(int)),
             angle, SLOT(setValue(int)));
```

Ici, on connecte le signal **valueChanged()** de **LCDRange** au slot **setAngle()** de **cannonField**. Ceci va permettre de mettre à jour l'angle de **cannonField** quand l'utilisateur manipule **LCDRange**. On rajoute aussi une connexion inversée, pour que le changement de l'angle de **cannonField** engendre réciproquement une mise à jour de **LCDRange**. Dans notre exemple, nous n'aurons jamais à modifier l'angle de **cannonField** directement. En rajoutant le dernier **connect()**, on s'assure que, dans le cas d'un changement futur, aucune modification ne perturbera la synchronisation entre les deux valeurs.

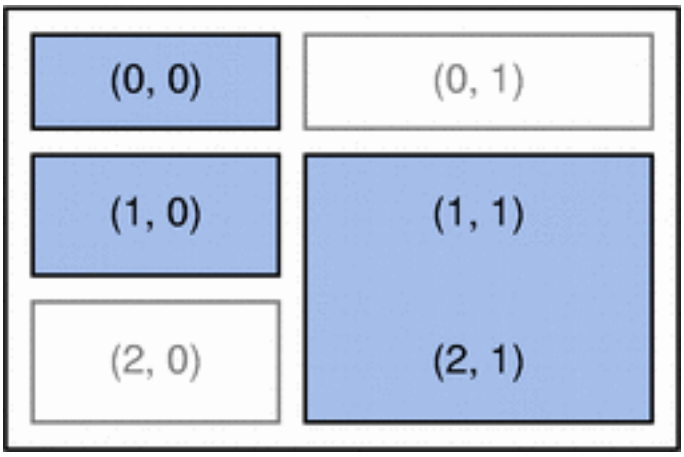
Ceci illustre la puissance de la programmation par composants avec une encapsulation propre.

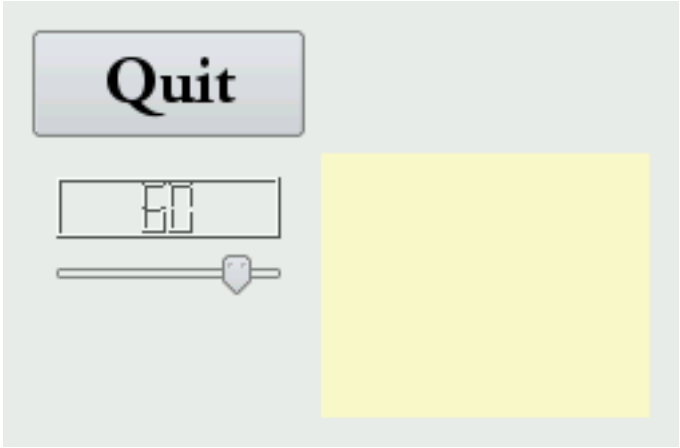
Noter qu'il est important de n'émettre le signal valueChanged() que dans le cas où l'angle change vraiment. Si les deux widget **LCDRange** et **cannonField** ont omis ce contrôle, le programme va entrer dans une boucle infinie au premier changement de l'une de ces deux valeurs.

```
QGridLayout *gridLayout = new QGridLayout;
```

Jusqu'à présent on a utilisé **QVBoxLayout** pour gérer la géométrie, mais, maintenant, on voudrait plus de contrôle sur la disposition, et on va basculer vers une disposition plus puissante avec la classe **QGridLayout**. **QGridLayout** n'est pas un widget, c'est une classe différente qui va permettre de gérer les enfants de n'importe quel widget.

On n'a pas besoin de spécifier les dimensions dans le constructeur de **QGridLayout**, vu qu'il va déterminer le nombre de lignes et de colonnes en se basant sur les cellules de la grille qu'on définit.





Le diagramme ci-dessus montre la disposition vers laquelle on veut aboutir, le coté gauche montre un schéma de la disposition, et le coté droit montre une capture d'écran actuelle du programme.

```
gridLayout->addWidget(quit, 0, 0);
```

On ajoute le bouton **Quit** à la cellule au coin supérieur gauche de la grille, c'est-à-dire la cellule avec les coordonnées (0,0).

```
gridLayout->addWidget(angle, 1, 0);
```

On rajoute l'angle **LCDRange** à la cellule (1,0).

```
gridLayout->addWidget(cannonField, 1, 1, 2, 1);
```

On laisse **cannonField** occuper les cellules (1,1) et (2,1).

```
gridLayout->setColumnStretch(1, 10);
```

On ordonne à **QGridLayout** de rendre la colonne de droite (2) extensible, avec un facteur de 10. Puisque la colonne de gauche ne l'est pas (son facteur d'extensibilité est 0, la valeur par défaut), **QGridLayout** va essayer de garder la taille des widgets à gauche inchangés, et va redimensionner le **CannonField** quand la taille de MyWidget change.

Dans cet exemple particulier, tout facteur d'extensibilité plus grand que 0 pour la colonne 1 aurait le même effet. Dans des dispositions géométriques plus complexes, vous pouvez utiliser les facteurs d'extensibilités pour spécifier qu'une colonne ou une ligne s'étend plus rapidement qu'une autre en assignant les facteurs adéquats.

```
angle->setValue(60);
```

On définie la valeur initiale de l'angle. Noter que ceci va déclencher la connexion entre **LCDRange** et **CannonField.**

```
angle->setFocus();
```

Notre dernière action est de donner le focus au widget **angle**, ainsi toute action sur le clavier va aller vers le widget **LCDRange** par défaut.

LCDRange ne contient aucun **keyPressEvent()**, il semble que ceci n'est pas vraiment intéressant. Cependant, son constructeur possède une nouvelle ligne.

```
setFocusProxy(slider);
```

Le **LCDRange** va définir le slider comme son " focus proxy ", ce qui signifie que, quand quelqu'un donne le contrôle du clavier au **LCDRange**, le slider doit le prendre en charge. **QSlider** possède une interface clavier décente, qui a permit avec une seule ligne de code d'en donner une à **LCDRange.**

VIII-B - Lancer l'application

Maintenant, le clavier peut être utilisé : les flèches, **Home**, **End**, **PageUp**, et **PageDown** font vraiment quelque chose.

Quand le slider est manipulé, Le **CannonField** affiche un nouvel angle. Au redimensionnement, **CannonField** prend le plus d'espace possible.

VIII-C - Exercices

Essayer de redimensionner la fenêtre. Que se passe-t-il quand la fenêtre devient trop étroite ou bien trop large ?

Quand vous donnez à la colonne de gauche un facteur d'extensibilité supérieur à zéro, que se passe-t-il quand vous redimensionner la fenêtre ?

Enlever l'appel à QWidget::setFocus(). Quel est le comportement que vous préférez ?

Essayez de remplacer " Quit " par " &Quit ", Quel changement sur le look du bouton ? Que se passe-t-il si on appuye sur Alt+Q pendant l'exécution du programme ?

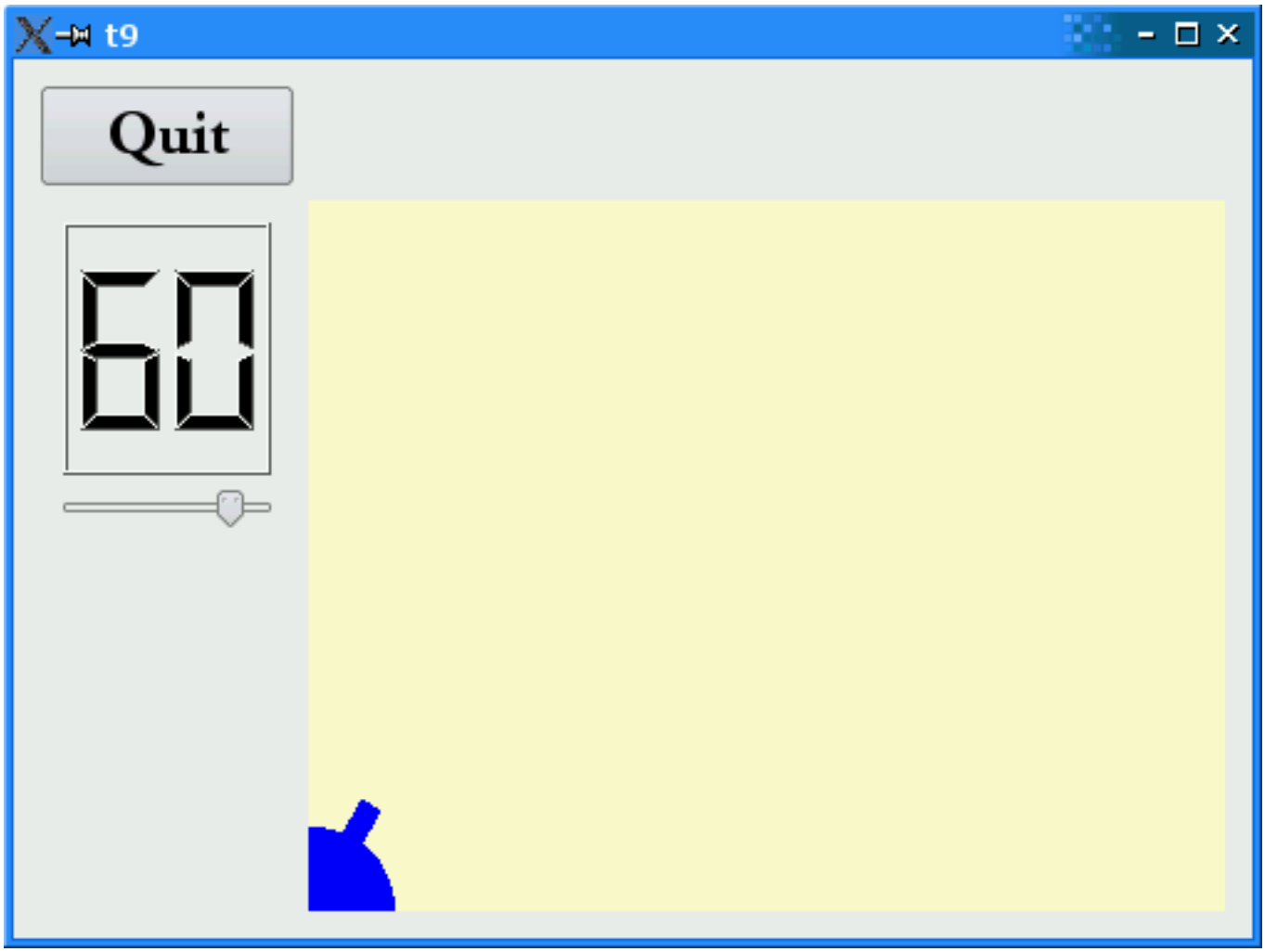
Centrer le texte sur **CannonField**.

IX - Avec un canon, on peut tout faire

Source t9.rar

Fichiers

- tutorials/tutorial/t9/cannonfield.cpp
- tutorials/tutorial/t9/cannonfield.h
- tutorials/tutorial/t9/lcdrange.cpp
- tutorials/tutorial/t9/lcdrange.h
- tutorials/tutorial/t9/main.cpp
- tutorials/tutorial/t9/t9.pro



Dans cet exemple, nous nous attaquons au graphique, en dessinant un joli petit canon. Il n'y a que *cannonfield.cpp* qui change par rapport au chapitre précédent.

IX-A - Analyse du code ligne par ligne

- t9/cannonfield.cpp

```
 void CannonField::paintEvent(QPaintEvent * /* event */)
{
    QPainter painter(this);
```

Nous allons maintenant commencer à utiliser sérieusement *QPainter*. Nous créons un peintre qui va travailler sur ce *widget*.

```
    painter.setPen(Qt::NoPen);
```

Les bords du dessin de *QPainter* sont faits par le crayon. Ici, nous le configurons à *Qt::NoPen*, ce qui veut dire qu'il n'y aura pas de bord spécial quand nous dessinerons quelque chose.

```
    painter.setBrush(Qt::blue);
```

Quand QPainter remplit un rectangle, un cercle, ou n'importe quoi d'autre, il remplit la forme avec sa brosse. Ici, nous le configurons pour qu'il utilise une brosse d'un bleu plein. Nous aurions également pu utiliser un motif. La brosse bleue va remplir tout l'intérieur des bords de ce que nous allons dessiner.

```
    painter.translate(0, rect().height());
```

Cette fonction traduit le système de coordonnées de *QPainter* (il le déplace d'un offset). Ici, nous plaçons le point (0,0) au coin inférieur gauche du widget. Les directions *X* et *Y* restent inchangées : toutes les coordonnées *y* dans le widget sont donc négatives. (voir *The Coordinate System* pour plus d'informations à propos du système de coordonnées de Qt).

```
    painter.drawPie(QRect(-35, -35, 70, 70), 0, 90 * 16);
```

La fonction *QPainter::drawPie()* dessine un camembert dans le rectangle spécifié à partir d'un angle de départ et de la longueur d'un arc. Les angles sont spécifiés en 16e de degrés. Le zéro est situé au 3-heures horaire. Le dessin est anti-horaire. Ici, nous dessinons un quart de cercle dans le coin inférieur gauche du widget. Le camembert est rempli avec du bleu, sans contour.

```
    painter.rotate(-currentAngle);
```

La fonction *QPainter::rotate()* fait tourner le système de coordonnées du *QPainter* autour de l'origine. L'argument de la rotation est un nombre à virgule flottante, en degrés (et pas en 16e de degrés comme précédemment) et dans le sens horaire. Ici nous tournons le système de coordonnées de currentAngle degrés dans le sens anti-horaire.

```
    painter.drawRect(QRect(30, -5, 20, 10));
```

La fonction *QPainter::drawRect()* dessine le rectangle spécifié. Ici, nous dessinons l'orifice du canon.

Il peut être souvent difficile d'imaginer le résultat quand le système de coordonnées a été transformé (translaté, tourné, réduit, décalé) comme ci-dessus.

Dans ce cas, le système de coordonnées est d'abord translaté, puis tourné. Si le rectangle QRect(30, -5, 20, 10) avait été dessiné dans le système de coordonnées translaté, il aurait ressmblé à cela:

Notez que le rectangle est fixé au bord du *widget cannonfield*. Puis, nous tournons le système de coordonnées, par exemple de 60 degrés, le rectangle sera tourné autour de (0,0), qui est le coin inférieur gauche car nous avons translaté le système de coordonnées. Le résultat ressemble à cela :

IX-B - Exécuter l'application

Quand la position du *slider* évolue, l'angle du canon dessiné change, en accord avec cette valeur.

Le 'Q' sur le bouton **Quit** est maintenant souligné, et **Alt+Q** permet de l'activer.

IX-C - Exercices

Configurez un crayon différent au lieu de Qt::NoPen. Utilisez un pinceau avec des motifs.

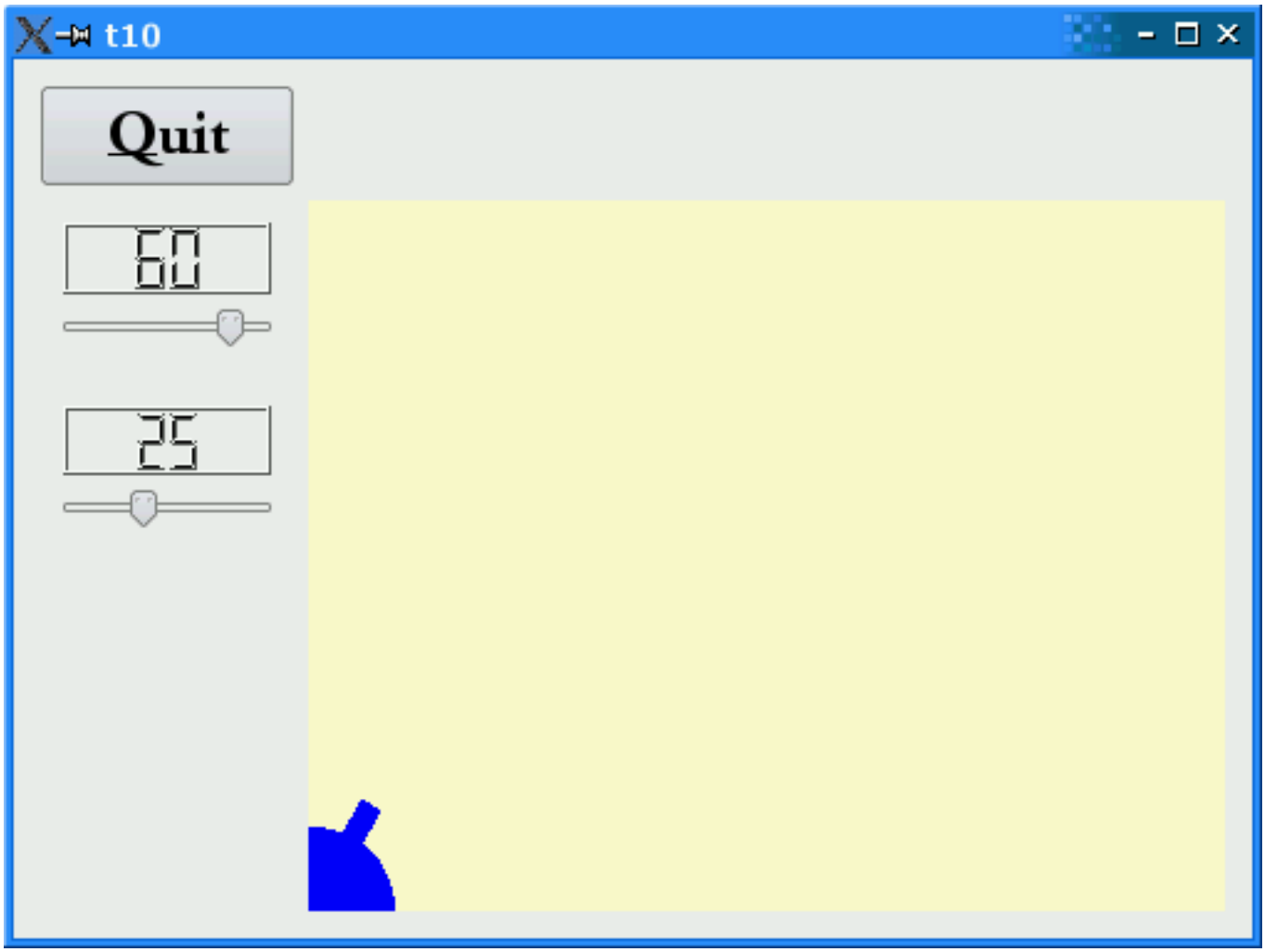
Essayez "Q&uit" ou "Qu&it" comme texte de bouton au lieu de "&Quit". Que se passe-t-il?

X - Doux comme un agneau

Source t10.rar

Fichiers

- tutorials/tutorial/t10/cannonfield.cpp
- tutorials/tutorial/t10/cannonfield.h
- tutorials/tutorial/t10/lcdrange.cpp
- tutorials/tutorial/t10/lcdrange.h
- tutorials/tutorial/t10/main.cpp
- tutorials/tutorial/t10/t10.pro



Dans cet exemple on rajoute le contrôle de la force.

X-A - Analyse du code ligne par ligne

- t10/connonfield.h

Le **cannonField** possède les attributs force et l'angle, définis précédemment.

```
    int angle() const { return currentAngle; }
    int force() const { return currentForce; }
```

```
public slots:
    void setAngle(int angle);
    void setForce(int force);
signals:
    void angleChanged(int newAngle);
    void forceChanged(int newForce);
```

L'interface de la force se définit de la même manière que l'angle.

```
private:
    QRect cannonRect() const;
```

La définition du cadre du canon se situe dans une fonction séparée.

```
    int currentAngle;
    int currentForce;
};
```

La force est stockée dans la variable **currentForce**.

- t10/cannonField.cpp

```
CannonField::CannonField(QWidget *parent)
    : QWidget(parent)
{
    currentAngle = 45;
    currentForce = 0;
    setPalette(QPalette(QColor(250, 250, 200)));
    setAutoFillBackground(true);
}
```

La force est initialisée à zéro.

```
void CannonField::setAngle(int angle)
{
    if (angle < 5)
        angle = 5;
    if (angle > 70)
        angle = 70;
    if (currentAngle == angle)
        return;
    currentAngle = angle;
    update(cannonRect());
    emit angleChanged(currentAngle);
}
```

On a fait quelques modifications dans la fonction **setAngle()**. Elle ne redessine que la portion du widget qui contient le canon.

```
void CannonField::setForce(int force)
{
    if (force < 0)
        force = 0;
    if (currentForce == force)
        return;
    currentForce = force;
    emit forceChanged(currentForce);
}
```

L'implémentation de **setForce** est similaire à celle **setAngle()**. La seule différence réside dans le fait que la force ne nécessite pas un affichage, donc on n'a pas besoin de redessiner le widget.

```
void CannonField::paintEvent(QPaintEvent * /* event */)
 {
     QPainter painter(this);
     painter.setPen(Qt::NoPen);
     painter.setBrush(Qt::blue);
     painter.translate(0, height());
     painter.drawPie(QRect(-35, -35, 70, 70), 0, 90 * 16);
     painter.rotate(-currentAngle);
     painter.drawRect(QRect(30, -5, 20, 10));
 }
```

On déssine comme dans le chapitre 9.

```
QRect CannonField::cannonRect() const
 {
     QRect result(0, 0, 50, 50);
     result.moveBottomLeft(rect().bottomLeft());
     return result;
 }
```

La fonction **cannonRect()** retourne le cadre contenant le canon dans les coordonnées relatives au widget. On crée d'abord un rectangle de dimensions 50 x 50, ensuite on le déplace jusqu'à ce que le coin inférieur gauche coïncide avec le coin inférieur gauche du widget.

La méthode **QWidget::rect()** retourne le cadre d'un **QWidget** dans ses propres coordonnées. Les coordonnées du coin supérieur gauche sont toujours (0,0).

- t10/main.cpp

```
MyWidget::MyWidget(QWidget *parent)
     : QWidget(parent)
 {
```

Le constructeur demeure globalement le même, on a juste ajouté quelques miettes.

```
   LCDRange *force = new LCDRange;
      force->setRange(10, 50);
```

On a ajouté un nouveau **LCDRange**, qui va être utilisé pour manipuler la force.

```
     connect(force, SIGNAL(valueChanged(int)),
             cannonField, SLOT(setForce(int)));
     connect(cannonField, SIGNAL(forceChanged(int)),
             force, SLOT(setValue(int)));
```

On connecte un widget **cannonField**, comme on a fait pour le widget **angle**.

```
     QVBoxLayout *leftLayout = new QVBoxLayout;
     leftLayout->addWidget(angle);
     leftLayout->addWidget(force);
     QGridLayout *gridLayout = new QGridLayout;
     gridLayout->addWidget(quit, 0, 0);
     gridLayout->addLayout(leftLayout, 1, 0);
     gridLayout->addWidget(cannonField, 1, 1, 2, 1);
     gridLayout->setColumnStretch(1, 10);
```

Dans le chapitre 9, on a mis un angle dans la cellule du coin inférieur gauche. Maintenant, on voudrait deux widgets dans cette cellule. Alors, on crée une boîte de disposition verticale, on met cette boîte dans la cellule et on ajoute **angle** et **range** dans cette boîte.

```
  force->setValue(25);
```

On initialise la force à 25.

X-B - Exécuter l'application

On possède maintenant le contrôle de la force.

X-C - Exercices

Rendre la taille du canon dépendante de la force.

Mettre le canon dans le coin inférieur droit.

Essayer d'améliorer l'interface, en ajoutant les boutons **+** et □ pour augmenter ou diminuer la force et le bouton **entrée** pour tirer. Si vous êtes dérangé par le comportement des boutons **Left** et **Right**, vous pouvez les changer, eux aussi.

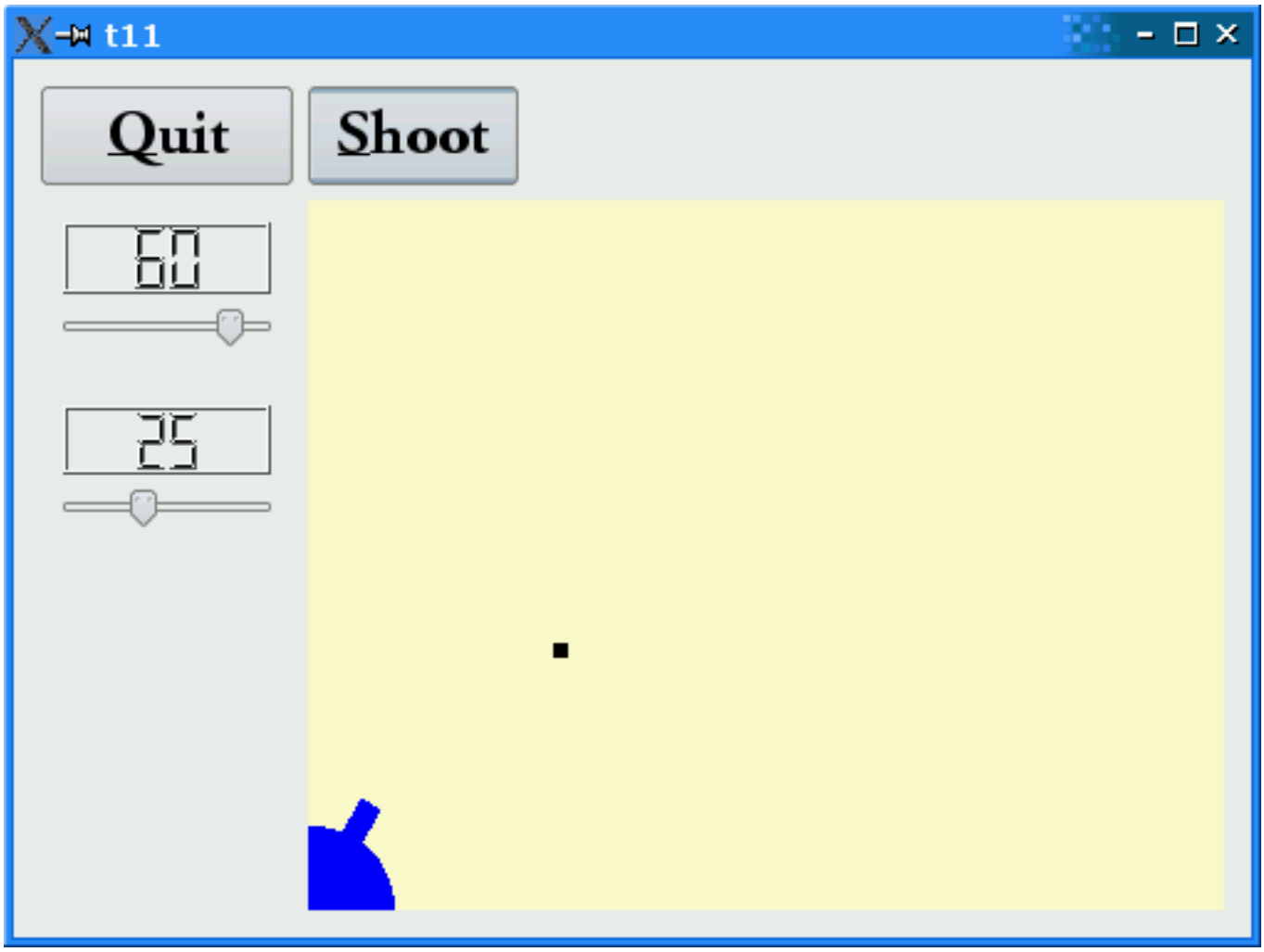
Remarque: Réimplémentez **QWidget::KeyPressEvent()**

XI - Tire-lui dessus!

Source t11.rar

Fichiers

- tutorials/tutorial/t11/cannonfield.cpp
- tutorials/tutorial/t11/cannonfield.h
- tutorials/tutorial/t11/lcdrange.cpp
- tutorials/tutorial/t11/lcdrange.h
- tutorials/tutorial/t11/main.cpp
- tutorials/tutorial/t11/t11.pro



Dans cet exemple, nous introduisons un *timer* pour créer un tir animé.

XI-A - Analyse du code ligne par ligne

- t11/cannonfield.h

```
CannonField
```

est maintenant capable de tirer.

```
    void shoot();
```

Appeler ce slot fera tirer le canon à condition qu'un projectile ne soit pas dans les airs.

```
private slots:
    void moveShot();
```

Ce slot privé est utilisé pour déplacer le projectile pendant qu'il est dans l'air, en utilisant *QTimer*.

```
private:
    void paintShot(QPainter &painter);
```

La fonction privée qui dessine le tir.

```
    QRect shotRect() const;
```

Cette fonction privée renvoie le rectangle qui contient le tir si on est dans le ciel ; autrement, elle retourne un rectangle indéfini.

```
    int timerCount;
    QTimer *autoShootTimer;
    float shootAngle;
    float shootForce;
};
```

Ces variables privées contiennent les informations qui décrivent le tir. Le *timerCount* calcule le temps passé depuis le dernier tir. *shootAngle* représente l'angle du canon, et *shootForce*, la force du canon.

- t11/cannonfield.cpp

```
#include <math.h>
```

Nous incluons *math.h* pour pouvoir utiliser les fonctions trigonométriques *sin()* et *cos()*. Une alternative pourrait être d'inclure *cmath*, plus moderne. Cependant, certaines plateformes UNIX ne supportent toujours pas cet en-tête.

```
CannonField::CannonField(QWidget *parent)
    : QWidget(parent)
{
    currentAngle = 45;
    currentForce = 0;
    timerCount = 0;
    autoShootTimer = new QTimer(this);
    connect(autoShootTimer, SIGNAL(timeout()), this, SLOT(moveShot()));
    shootAngle = 0;
    shootForce = 0;
    setPalette(QPalette(QColor(250, 250, 200)));
    setAutoFillBackground(true);
}
```

Nous initialisons nos nouvelles variables privées et nous connectons le signal *QTimer::timeout()* à *moveShot()*. Nous bougeons le tir à chaque fois que le *timer* du cannon dépasse le temps imparti.

```
void CannonField::shoot()
{
    if (autoShootTimer->isActive())
        return;
    timerCount = 0;
    shootAngle = currentAngle;
    shootForce = currentForce;
```

```
    autoShootTimer->start(5);
}
```

Cette fonction tire, mais pas dans le ciel. *timerCount* est remis à zéro, *shootAngle* et *shootForce* sont définis avec l'angle et la force. Pour finir, nous démarrons le *timer*.

```
void CannonField::moveShot()
{
    QRegion region = shotRect();
    ++timerCount;

    QRect shotR = shotRect();

    if (shotR.x() > width() || shotR.y() > height()) {
        autoShootTimer->stop();
    } else {
        region = region.unite(shotR);
    }
    update(region);
}
```

moveShot() est le slot qui déplace le tir, il est appelé toutes les 5 milisecondes pendant que *QTimer* tire. Ses tâches sont de calculer la nouvelle position, de mettre à jour l'écran avec le tir à sa nouvelle position, et, si nécessaire, d'arrêter le *timer*. En premier, nous créons une *QRegion* qui conserve l'ancien rectangle de tir. Une *QRegion* est capable de garder n'importe quelle région, nous les utilisont ici pour simplifier le dessin. *shotRect()* renvoie le rectangle du slot. Ce sera expliqué en détail un peu plus bas. Maintenant nous incrémentons le *timerCount* qui a pour effet de décaler la trajectoire de tir. Après, nous cherchons le nouveau rectangle de tir. Si le projectile est sorti du bord droit ou du bas du widget, nous stoppons le *timer*, où nous ajoutons le nouveau *shotRect()* à la *QRegion*. Pour finir, nous redessinons la *QRegion*. Cela va envoyer un simple événement "peindre" pour mettre à jour seulement les rectangles qui en ont besoin.

```
void CannonField::paintEvent(QPaintEvent * /* event */)
{
    QPainter painter(this);

    paintCannon(painter);
    if (autoShootTimer->isActive())
        paintShot(painter);
}
```

L'événement "peindre" a été simplifié depuis le précédent chapitre. La majeure partie de la logique a été déplacé dans les nouvelles fonctions *paintShot()* et *paintCannon()*.

```
void CannonField::paintShot(QPainter &painter)
{
    painter.setPen(Qt::NoPen);
    painter.setBrush(Qt::black);
    painter.drawRect(shotRect());
}
```

Cette méthode privée dessine le tir en peignant un rectangle noir. Nous laisson de côté l'implémentation de *paintCannon()*, semblable à *QWidget::paintEvent()*, réimplanté du dernier chapitre.

```
QRect CannonField::shotRect() const
{
    const double gravity = 4;

    double time = timerCount / 20.0;
    double velocity = shootForce;
    double radians = shootAngle * 3.14159265 / 180;

    double velx = velocity * cos(radians);
    double vely = velocity * sin(radians);
    double x0 = (barrelRect.right() + 5) * cos(radians);
```

```
    double y0 = (barrelRect.right() + 5) * sin(radians);
    double x = x0 + velx * time;
    double y = y0 + vely * time - 0.5 * gravity * time * time;

    QRect result(0, 0, 6, 6);
    result.moveCenter(QPoint(qRound(x), height() - 1 - qRound(y)));
    return result;
}
```

Cette fonction détermine le centre du projectile et renvoie le rectangle contenant le projectile. Il utilise la force initiale du canon, l'angle et ajoute *timerCount* au temps passé. La formule utilisée est la formule standard de Newton pour un mouvement sans frottement, sous l'effet de la gravité. Pour simplifier, nous avons négligé l'effet d'Einstein. Nous calculons le point central dans un système où les coordonnées *y* augmentent vers le haut. Après avoir calculé la position du point central, nous créons un *QRect* de taille 6x6 que nous centrons (à l'aide des points que nous avons calculé). En même temps nous convertissons le point en coordonées pour le widget (voir le système de coordonnées). La fonction *qRound()* est une fonction *inline* définie dans *QtGlobal*, et incluse dans tous les en-têtes de Qt. *qRound()* arrondit un *double* en *integer*.

- t11/main.cpp

```
class MyWidget : public QWidget
{
public:
    MyWidget(QWidget *parent = 0);
};
```

Ajout d'un bouton poussoir.

```
    QPushButton *shoot = new QPushButton(tr("&Shoot"));
    shoot->setFont(QFont("Times", 18, QFont::Bold));
```

Dans le constructeur, nous créons et configurons le bouton exactement comme nous l'avions fait avec le bouton *Quitter*.

```
    connect(shoot, SIGNAL(clicked()), cannonField, SLOT(shoot()));
```

Connexion du signal *clicked()* du bouton poussoir au slot *shoot()* de *CannonField*.

XI-B - Exécuter l'application

Le canon peut tirer, mais il n'a rien a détruire.

XI-C - Exercices

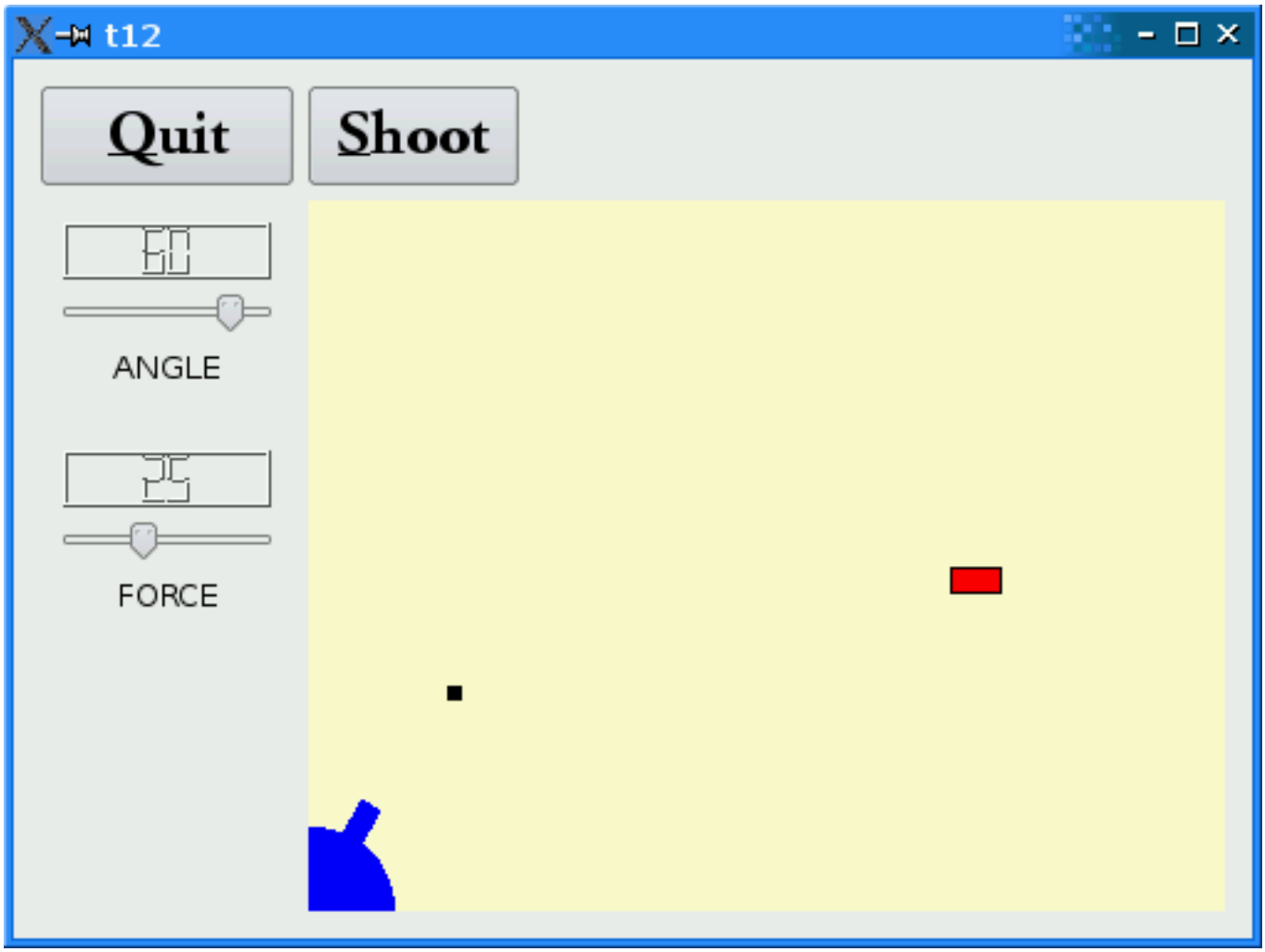
Représentez le tir par un cercle [Astuce : *QPainter::drawEllipse()* peut aider] Changez la couleur du canon quand il vise le ciel.

XII - Accrochons des briques en l'air

Source t12.rar

Fichiers

- tutorials/tutorial/t12/cannonfield.cpp
- tutorials/tutorial/t12/cannonfield.h
- tutorials/tutorial/t12/lcdrange.cpp
- tutorials/tutorial/t12/lcdrange.h
- tutorials/tutorial/t12/main.cpp
- tutorials/tutorial/t12/t12.pro



Dans cet exemple, nous étendons notre classe LCDRange pour ajouter un label. Nous fournissons également quelque chose pour tirer dessus.

XII-A - Analyse du code ligne par ligne

- t12/lcdrange.h

LCDRange possède maintenant un label.

```
class QLabel;
```

```
class QSlider;
```

Nous déclarons à l'avance *QLabel* et *QSlider* car nous voulons utiliser des pointeurs dessus dans la définition de la classe. Nous aurions pu également utiliser *#include*, mais cela aurait ralenti la compilation pour rien.

```
class LCDRange : public QWidget
{
    Q_OBJECT

public:
    LCDRange(QWidget *parent = 0);
    LCDRange(const QString &text, QWidget *parent = 0);
```

Nous avons ajouté un nouveau contructeur qui configure le label en complément du parent.

```
    QString text() const;
```

La fonction retourne le label.

```
    void setText(const QString &text);
```

Cette partie configure le label.

```
private:
    void init();
```

Comme nous avons maintenant deux constructeurs, nous avons choisi de placer l'initialisation commune dans la fonction privée *init()*.

```
    QLabel *label;
```

Nous avons également une nouvelle variable privée : un *QLabel*. *QLabel* est l'un des widgets standards de Qt et peut afficher un texte ou un *QPixmap*, avec ou sans fenêtre.

- t12/lcdrange.cpp

```
LCDRange::LCDRange(QWidget *parent)
    : QWidget(parent)
{
    init();
}
```

Ce constructeur appelle la fonction *init()*, qui contient le code d'initialisation commun.

```
LCDRange::LCDRange(const QString &text, QWidget *parent)
    : QWidget(parent)
{
    init();
    setText(text);
}
```

Ce constructeur appelle d'abord *init()* et configure ensuite le label.

```
void LCDRange::init()
{
    QLCDNumber *lcd = new QLCDNumber(2);
    lcd->setSegmentStyle(QLCDNumber::Filled);

    slider = new QSlider(Qt::Horizontal);
```

```
    slider->setRange(0, 99);
    slider->setValue(0);
    label = new QLabel;
    label->setAlignment(Qt::AlignHCenter | Qt::AlignTop);

    connect(slider, SIGNAL(valueChanged(int)),
            lcd, SLOT(display(int)));
    connect(slider, SIGNAL(valueChanged(int)),
            this, SIGNAL(valueChanged(int)));

    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(lcd);
    layout->addWidget(slider);
    layout->addWidget(label);
    setLayout(layout);

    setFocusProxy(slider);
}
```

La configuration du LCD et du *slider* est la même que dans le chapitre précédent. Ensuite, nous créons un *QLabel* et lui disons d'aligner le contenu au centre, horizontalement, et en haut, verticalement. L'appel à *QObject::connect()* a également été pris du chapitre précédent.

```
QString LCDRange::text() const
{
    return label->text();
}
```

Cette fonction renvoie le texte du label.

```
void LCDRange::setText(const QString &text)
{
    label->setText(text);
}
```

Cette fonction configure le texte du label.

- t12/cannonfield.h

Le *cannonfield* a maintenant deux nouveaux signaux : *hit()* et *missed()*. De plus, il contient une cible.

```
    void newTarget();
```

Ce slot crée une cible à une nouvelle position.

```
signals:
    void hit();
    void missed();
```

Le signal *hit()* est émis quand un tir touche la cible. Le signal *missed()* est émis quand le tir va au-delà des bords droit et bas du widget (c'est certain qu'il n'a pas et ne touchera pas la cible).

```
    void paintTarget(QPainter &painter);
```

Cette fonction privée dessine la cible.

```
    QRect targetRect() const;
```

Cette fonction privée retourne le rectangle lié à la cible.

```
    QPoint target;
```

Cette variable privée contient le centre de la cible.

- t12/cannonfield.cpp

```
#include <stdlib.h>
```

Nous incluons le fichier d'en-tête *<stdlib.h>* car nous avons besoin de la fonction *qrand()*.

```
    newTarget();
```

Cette ligne a été ajoutée au constructeur. Il crée une position "aléatoire" pour la cible. En fait, la fonction *newTarget()* va essayer de peindre la cible. Comme nous sommes dans un constructeur, le *widget CannonField* est invisible. Qt garantit qu'aucun dommage n'est fait quand *QWidget::update()* est appelé sur un widget invisible.

```
void CannonField::newTarget()
{
    static bool firstTime = true;

    if (firstTime) {
        firstTime = false;
        QTime midnight(0, 0, 0);
        qsrand(midnight.secsTo(QTime::currentTime()));
    }
    target = QPoint(200 + qrand() % 190, 10 + qrand() % 255);
    update();
 }
```

Cette fonction privée crée un point central de cible à une position aléatoire.

Nous utilisons la fonction *qrand()* pour obtenir des entiers aléatoires. Cette fonction retourne normalement les mêmes séries de nombres à chaque fois que vous lancez un programme. Cela ferait apparaître la cible à la même position à chaque fois. Pour éviter cela, nous devons configurer un pas aléatoire au premier appel de la fonction. Ce pas doit également être aléatoire afin d'éviter des séries de nombres aléatoires identiques. La solution est d'utiliser le nombre de secondes qui sont passées depuis minuit comme valeur pseudo-aléatoire.

D'abord nous créons une varibale booléenne locale statique. Il est garanti qu'une variable statique comme celle-ci conserve sa valeur entre deux appels à la fonction.

La condition ne sera vraie qu'au premier appel de cette fonction car nous passons *firstTime* à *false* dans le bloc conditionnel.

Ensuite nous créons l'objet *QTime* "*midnight*", qui représente minuit. Puis, nous récupérons le nombre de secondes depuis minuit jusqu'à maintenant et l'utilisons comme un pas aléatoire. Voir la documentation de *QDate*, *QTime* et *QDateTime* pour plus d'informations.

Enfin, nous calculons le point central de la cible. Nous le conservons dans le rectangle (*x* : 200, *y* : 35, *width* : 190, *height* : 255, les valeurs possibles de *x* et *y* sont respectivement de 200 à 389 et de 35 à 289), dans un système de coordonnées où nous plaçons la valeur 0 de *y* au bord inférieur du widget et laissons ses valeurs incrémenter, alors que *x* est comme d'habitude, c'est-à-dire que la valeur 0 correspond au bord gauche et qu'elle s'incrémente vers la droite.

Par expérience, nous avons trouvé cela pour être toujours à la portée du tir.

```
void CannonField::moveShot()
{
    QRegion region = shotRect();
```

```
    ++timerCount;

    QRect shotR = shotRect();
```

Cette partie de l'évènement du *timer* n'a pas changé depuis le chapitre précédent.

```
    if (shotR.intersects(targetRect())) {
        autoShootTimer->stop();
        emit hit();
```

Cette condition vérifie si le rectangle du tir rencontre le rectangle de la cible. Si c'est le cas, le tir a touché la cible (aïe !). Nous stoppons le *timer* du tir, émettons le signal *hit()* pour indiquer au monde exterieur qu'une cible a été détruite et retournons.

Notez que l'on aurait pu créer une nouvelle cible sur la zone, mais comme *CannonField* est un composant nous laissons une telle décision à l'utilisateur du composant.

```
    } else if (shotR.x() > width() || shotR.y() > height()) {
        autoShootTimer->stop();
        emit missed();
```

Cette condition est la même que dans le chapitre précédent, excepté qu'elle émet maintenant le signal *missed()* pour indiquer au monde extérieur cet échec.

```
    } else {
        region = region.unite(shotR);
    }
    update(region);
}
```

Et le reste de la fonction reste inchangé.

CannonField::paintEvent() est comme avant, sauf que ceci a été ajouté :

```
    paintTarget(painter);
```

Cette ligne s'assure que la cible est aussi dessinée quand il le faut.

```
void CannonField::paintTarget(QPainter &painter)
{
    painter.setPen(Qt::black);
    painter.setBrush(Qt::red);
    painter.drawRect(targetRect());
}
```

Cette fonction privée dessine la cible, un rectangle rempli de rouge avec un bord noir.

```
QRect CannonField::targetRect() const
{
    QRect result(0, 0, 20, 10);
    result.moveCenter(QPoint(target.x(), height() - 1 - target.y()));
    return result;
}
```

Cette fonction privée retourne le rectangle de la cible. Souvenez-vous de *newTarget()* dans laquelle le point de la cible utilisait la coordonnée 0 de *y* en bas du widget. Nous calculons le point en coordonnées du widget avant d'appeler *QRect::moveCenter()*.

La raison pour laquelle nous avons choisi ce changement de coordonnées est pour fixer la distance entre la cible et le bas du widget. Souvenez-vous que le widget peut être redimmensionné par l'utilisateur ou le programme à tout moment.

- t12/main.cpp

Il n'y a pas de nouveaux membres dans la classe *MyWidget*, mais nous avons légèrement changé le constructeur pour configurer les labels de texte du nouveau LCDRange.

```
    LCDRange *angle = new LCDRange(tr("ANGLE"));
```

Nous mettons "ANGLE" dans le label texte de l'angle.

```
    LCDRange *force = new LCDRange(tr("FORCE"));
```

Nous mettons "FORCE" dans le label texte de la force.

XII-B - Exécuter l'application

Le widget LCDRange paraît un peu bizarre : quand nous redimmensionnons le widget, le gestionnaire de couches intégré dans *QVBoxLayout* donne trop de place au label et pas assez au reste, provoquant ainsi un changement de tailles des deux LCDRange widgets. Nous corrigerons cela dans le prochain chapitre.

XII-C - Exercices

Faisez un bouton de triche qui, lorsqu'il est cliqué, fait afficher au *CannonField* la trajectoire du tir pendant 5 secondes.

Si vous avez fait l'exercice "tir rond" du chapitre précédent, essayez de changer le *shotRect()* en *shotRegion()* qui retourne une *QRegion()* pour avoir une détection de collision très précise.

Faites une cible mouvante.

Assurez-vous que la cible est toujours créée entièrement à l'écran.

Assurez-vous que le widget ne peut pas être redimmensionné et que la cible ne soit pas visible. [Indice : *QWidget::setMinimumSize()* est votre ami.]

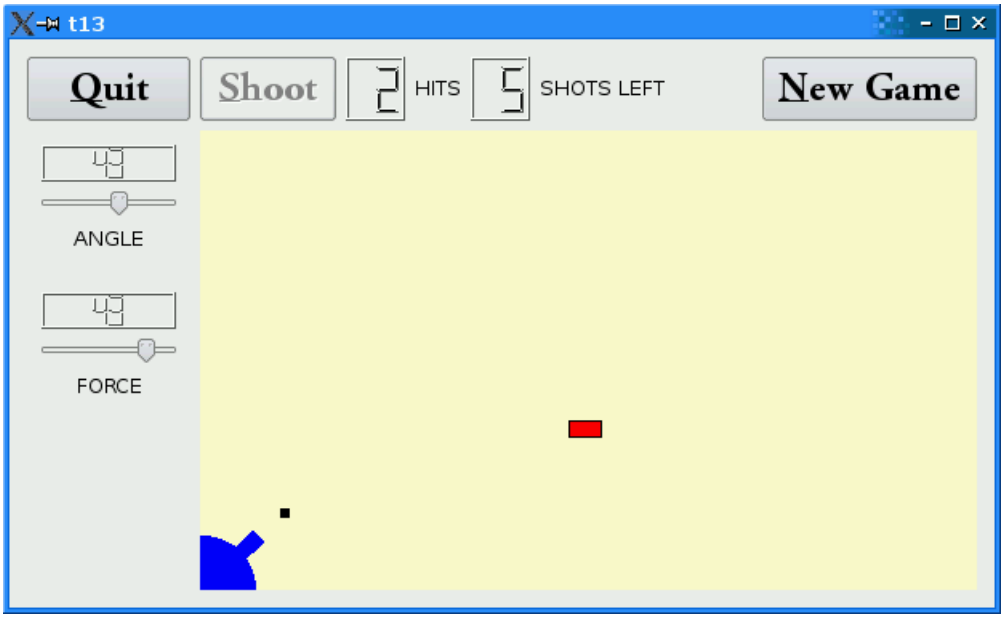
Pas facile, rendez possible d'avoir plusieurs tirs en même temps en l'air. [Indice : faites une classe Tir.]

XIII - Game Over

Source t13.rar

Fichiers

- tutorials/tutorial/t13/cannonfield.cpp
- tutorials/tutorial/t13/cannonfield.h
- tutorials/tutorial/t13/gameboard.cpp
- tutorials/tutorial/t13/gameboard.h
- tutorials/tutorial/t13/lcdrange.cpp
- tutorials/tutorial/t13/lcdrange.h
- tutorials/tutorial/t13/main.cpp
- tutorials/tutorial/t13/t13.pro



Dans cet exemple, nous commençons à nous approcher d'un jeu vraiment jouable. Nous donnons aussi à *MyWidget* un vrai nom (*GameBoard*), et quelques slots supplémentaires.

Nous plaçons la définition dans *gameboard.h*, et l'implémentation dans *gameboard.cpp*.

Le *CannonField* a désormais un état *game over*.

Les problèmes d'affichages de *LCDRange* sont réglés.

XIII-A - Analyse du code ligne par ligne

- t13/lcdrange.cpp

```
    label->setSizePolicy(QSizePolicy::Preferred, QSizePolicy::Fixed);
```

Nous déclarons la taille de police du *QLabel* à *(Preferred, Fixed)*. Le composant vertical s'assure que le *QLabel* ne bougera pas verticalement ; il va rester à sa taille optimale (sa *sizeHint()*). Ceci résoud un problème d'affichage, observé au cours du chapitre 12.

- t13/cannonfield.h

Le *CannonField* possède, en sus d'un état *game over*, quelques fonctions supplémentaires.

```
bool gameOver() const { return gameEnded; }
```

Cette fonction renvoie *true* si le jeu est perdu, *false* s'il continue.

```
void setGameOver();
void restartGame();
```

Voici les deux nouveaux slots : *setGameOver()* et *restartGame()*.

```
void canShoot(bool can);
```

Ce nouveau signal indique que le *CannonField* est dans un étant dans lequel le slot *shoot()* a du sens. Nous allons l'utiliser bientôt pour activer ou pas le boutton *Tirer*.

```
bool gameEnded;
```

Cette variable privée contient l'état du jeu ; *true* signifie que le jeu s'arrête, *false*, qu'il continue.

- t13/cannonfield.cpp

```
gameEnded = false;
```

Cette ligne a été ajoutée au constructeur. Initialement, le jeu n'est pas fini (heureusement pour le joueur :-)).

```
void CannonField::shoot()
{
    if (isShooting())
        return;
    timerCount = 0;
    shootAngle = currentAngle;
    shootForce = currentForce;
    autoShootTimer->start(5);
    emit canShoot(false);
}
```

Nous avons ajouté une nouvelle fonction *isShooting()*, pour que *shoot()* l'utilise, au lieu de le tester directement. Aussi, *shoot()* nous indique si le CannonFiels peut attaquer, ou non.

```
void CannonField::setGameOver()
{
    if (gameEnded)
        return;
    if (isShooting())
        autoShootTimer->stop();
    gameEnded = true;
    update();
}
```

Ce slot finit le jeu. Il doit être appelé en dehors du *CannonField*, car ce widget ne sait pas quand finir le jeu. Ceci est un principe de design très important dans la programmation par composants. Nous avons choisi de créer un composant aussi flexible que possible pour qu'il puisse se plier à différentes règles (par exemple, une version multi-joueur dans lequel le premier joueur à frapper dix fois a gagné, peut utiliser le *CannonField* inchangé.)

Si le jeu a déjà été terminé, nous retournons immédiatement. Si le jeu continue, nous arrêtons le tir, utilisons le *flag game over*, et repeignons l'intégralité de ce widget.

```
void CannonField::restartGame()
{
    if (isShooting())
        autoShootTimer->stop();
    gameEnded = false;
    update();
    emit canShoot(true);
}
```

Ce slot commence une nouvelle partie. Si un tir est dans l'air, nous l'arrêtons. Ensuite, nous remmettons la variable *gameEnded* à *false* et repeignons le widget.

moveShot(), aussi, émet le (nouveau) signal *canShoot(true)*, en même temps que *hit()* ou *miss()*.

Modifications dans *CannonField::paintEvent()* :

```
void CannonField::paintEvent(QPaintEvent * /* event */)
{
    QPainter painter(this);

    if (gameEnded) {
        painter.setPen(Qt::black);
        painter.setFont(QFont("Courier", 48, QFont::Bold));
        painter.drawText(rect(), Qt::AlignCenter, tr("Game Over"));
    }
```

L'évènement *paint* a été amélioré pour afficher le texte *Game Over*, si le jeu est fini (par exemple, si *gameEnded* est à *true*). Nous ne nous occupons pas de vérifier la mise à jour du rectangle car la vitesse n'est pas du tout critique quand le jeu est fini.

Pour dessiner le texte, nous utilisons d'abord un stylo noir ; cette couleur est utilisée quand le texte est dessiné. Ensuite, nous choisissons une police, de la famille Courier, de 48 points de haut et grasse. Finalement, nous dessinons le texte, au centre du widget du rectangle. Malheureusement, sur certains systèmes (spécialement les serveurs X avec des polices Unicode), cela peut prendre du temps, beaucoup de temps pour charger une police aussi grande. Parce que Qt met en cache les polices, vous ne remarquerez ceci que la première fois que la police est utilisée.

```
    paintCannon(painter);
    if (isShooting())
        paintShot(painter);
    if (!gameEnded)
        paintTarget(painter);
}
```

Nous dessinons le tir seulement quand on tire, et la cible, seulement quand on joue.

- t13/gameboard.h

Ce fichier est nouveau : il contient la définition de la classe *GameBoard*, qui était autrefois connue comme *MyWidget*.

```
class CannonField;

class GameBoard : public QWidget
{
    Q_OBJECT

public:
    GameBoard(QWidget *parent = 0);
```

```
protected slots:
    void fire();
    void hit();
    void missed();
    void newGame();

private:
    QLCDNumber *hits;
    QLCDNumber *shotsLeft;
    CannonField *cannonField;
};
```

Nous avons ajouté quatre slots, qui sont protégés, et utilisés en interne. Nous avons aussi ajouté deux *QLCDNumbers* (*hits* et *shotsLeft*) qui affichent l'état du jeu.

- t13/gameboard.cpp

Ce fichier est nouveau, lui aussi, et contient l'implémentation de la classe.

Nous avons fait quelques changements dans le constructeur.

```
cannonField = new CannonField;
```

cannonField est désormais une variable membres, nous changeons donc le constructeur pour le prendre en compte.

```
connect(cannonField, SIGNAL(hit()),
        this, SLOT(hit()));
connect(cannonField, SIGNAL(missed()),
        this, SLOT(missed()));
```

Cette fois, nous voulons que quelque chose soit fait quand le tir a touché ou loupé la cible. Ainsi, nous connectons les signaux *hit()* et *missed* du *CannonField* aux deux slots protégés homonymes de la classe.

```
connect(shoot, SIGNAL(clicked()),
        this, SLOT(fire()));
```

Auparavant, nous avons connecté le signal *clicked()* du bouton directement au slot *shoot()* du *CannonField*. Cette fois, nous voulons garder une trace du nomlbre de tirs, nous le connectons donc au slot protégé dans la classe.

Notez la simplicité avec laquelle nous changeons le comportement d'un programme quand nous travaillons avec des composants.

```
connect(cannonField, SIGNAL(canShoot(bool)),
        shoot, SLOT(setEnabled(bool)));
```

Nous utilisons aussi le signal *canShoot()* pour activer ou pas le boutton de tir.

```
QPushButton *restart = new QPushButton(tr("&New Game"));
restart->setFont(QFont("Times", 18, QFont::Bold));
connect(restart, SIGNAL(clicked()), this, SLOT(newGame()));
```

Nous créons, initialisons, et connectons le boutton *Nouveau jeu*, comme nous avons fait avec les autres bouttons. Un simple clic sur ce boutton activera le slot *newGame()* dans ce widget.

```
hits = new QLCDNumber(2);
hits->setSegmentStyle(QLCDNumber::Filled);

shotsLeft = new QLCDNumber(2);
```

```
shotsLeft->setSegmentStyle(QLCDNumber::Filled);

QLabel *hitsLabel = new QLabel(tr("HITS"));
QLabel *shotsLeftLabel = new QLabel(tr("SHOTS LEFT"));
```

Nous créons quatre nouveaux widgets. Notez que nous ne nous occupons pas de garder les pointeurs vers les QLabel dans la classe car nous ne voulons plus rien faire avec ceux-là. Qt les supprimera quand le widget *GameBoard* sera détruit, et les classes gérant l'affichage vont les redimensionner de manière appropriée.

```
QHBoxLayout *topLayout = new QHBoxLayout;
topLayout->addWidget(shoot);
topLayout->addWidget(hits);
topLayout->addWidget(hitsLabel);
topLayout->addWidget(shotsLeft);
topLayout->addWidget(shotsLeftLabel);
topLayout->addStretch(1);
topLayout->addWidget(restart);
```

La cellule en haut, à droite du QGridLayout, commence à être peuplée. Nous mettons une étendue juste à gauche du bouton *Nouveau jeu* pour être sûr que ce boutton apparaisse toujours sur le côté droit de la fenêtre.

```
newGame();
```

Nous en avons fini avec la construction du *GameBoard*, nous commençons donc à l'utiliser avec *newGame()*. Même si *newGame()* est un slot, il peut être utilisé comme une fonction ordinaire.

```
void GameBoard::fire()
{
    if (cannonField->gameOver() || cannonField->isShooting())
        return;
    shotsLeft->display(shotsLeft->intValue() - 1);
    cannonField->shoot();
}
```

Cette fonction lance un tir. Si le jeu est fini, ou qu'il y a déjà un tir en l'air, nous nous arrêtons immédiatement. Nous décrémentons le nombre de tirs restants et disons au cannon de tirer.

```
void GameBoard::hit()
{
    hits->display(hits->intValue() + 1);
    if (shotsLeft->intValue() == 0)
        cannonField->setGameOver();
    else
        cannonField->newTarget();
}
```

Ce slot est activé quand un tir a touché la cible. Nous incrémentons le nombre de touches. S'il n'y a plus de tirs restants, le jeu est fini. Sinon, nous demandons au *CannonField* de générer une nouvelle cible.

```
void GameBoard::missed()
{
    if (shotsLeft->intValue() == 0)
        cannonField->setGameOver();
}
```

Ce slot est activé quand un tir a raté sa cible. S'il n'y a plus de tirs restants, le jeu est fini.

```
void GameBoard::newGame()
{
    shotsLeft->display(15);
    hits->display(0);
    cannonField->restartGame();
    cannonField->newTarget();
```

```
}
```

Ce slot est activé quand l'utilisateur clique sur le boutton *Nouveau jeu*. Il est aussi appelé par le constructeur. Premièrement, il autorise 15 tirs. Notez que ceci est le seul emplacement dans tout le programme où nous indiquons un nombre de tirs. Changez-le pour qu'il reflète vos *desiderata*. Ensuite, nous remettons à 0 le nombre de touches, relançons le jeu et générons une nouvelle cible.

- t13/main.cpp

Ce fichier est passé par un régime : *MyWidget* est parti, il ne reste plus que la fonction *main()*, inchangée, si ce n'est pour le nom.

XIII-B - Exécuter l'application

Le cannon peut tirer sur une cible ; une nouvelle cible est automatiquement créée quand la précédente est touchée.

Tirs restants et touches sont affichés et le programme en garde une trace. Le jeu prend fin, et un bouton pour commencer une nouvelle partie apparaît.

XIII-C - Exercices

Ajouter un vent pseudo-aléatoire et le montrer à l'utilisateur.

Afficher quelques effets quand la cible est touchée.

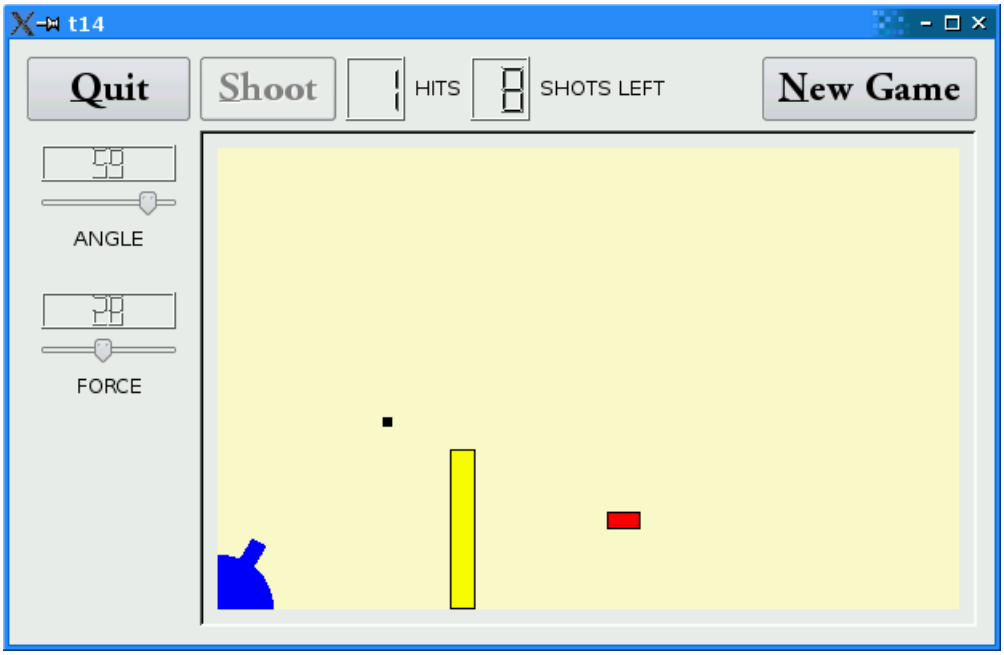
Implémenter les cibles mutliples.

XIV - Face au Mur

Source t14.rar

Fichiers

- tutorials/tutorial/t14/cannonfield.cpp
- tutorials/tutorial/t14/cannonfield.h
- tutorials/tutorial/t14/gameboard.cpp
- tutorials/tutorial/t14/gameboard.h
- tutorials/tutorial/t14/lcdrange.cpp
- tutorials/tutorial/t14/lcdrange.h
- tutorials/tutorial/t14/main.cpp
- tutorials/tutorial/t14/t14.pro



C'est l'exemple final : un jeu complet. Nous ajoutons les accélérateurs de clavier et ajoutons les événements de la souris à *CannonField*. Nous mettons un cadre autour de *CannonField* et placons un obstacle (un mur) pour rendre le jeu plus difficile.

XIV-A - Analyse du code ligne par ligne

- t14/cannonfield.h

Le *CannonField* peut maintenant recevoir les événements de la souris pour pointer l'extrémité du canon en cliquant et en la déplaçant. Le *CannonField* a également un obstacle.

```
protected:
    void paintEvent(QPaintEvent *event);
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void mouseReleaseEvent(QMouseEvent *event);
```

En plus des capteurs d'événements habituels, *CannonField* implémente trois capteurs d'événements pour la souris. Leurs noms parlent d'eux-mêmes.

```
    void paintBarrier(QPainter &painter);
```

Cette fonction privée dessine le mur.

```
    QRect barrierRect() const;
```

Cette fonction privée retourne le rectangle entourant l'obstacle.

```
    bool barrelHit(const QPoint &pos) const;
```

Cette fonction privée vérifie si un point est à l'intérieur de l'extrémité du canon.

```
    bool barrelPressed;
```

Cette variable privée est vraie si l'utilisateur a cliqué avec la souris sur l'extrémité du canon et ne l'a pas relâchée.

- t14/cannonfield.cpp

```
        barrelPressed = false;
```

Cette ligne a été ajoutée au constructeur. À l'initialisation, la souris ne clique pas sur le canon.

```
    } else if (shotR.x() > width() || shotR.y() > height()
               || shotR.intersects(barrierRect())) {
```

Maintenant que nous avons un obstacle, il y a trois moyen d'échouer. Nous allons tester la troisième en plus des précédentes.

```
void CannonField::mousePressEvent(QMouseEvent *event)
{
    if (event->button() != Qt::LeftButton)
        return;
    if (barrelHit(event->pos()))
        barrelPressed = true;
}
```

C'est un capteur d'événement Qt. Il est appelé quand l'utilisateur appuie sur un bouton de la souris alors que le curseur est sur le widget. Si l'événement n'est pas généré par le bouton gauche de la souris, nous retournons immédiatement. Sinon, nous vérifions que la position du curseur est à l'intérieur de l'extrémité du canon. Si c'est le cas, nous mettons *barrelPressed* à *true*. Notez que la fonction *QMouseEvent::pos()* retourne un point dans le système de coordonnées du widget.

```
void CannonField::mouseMoveEvent(QMouseEvent *event)
{
    if (!barrelPressed)
        return;
    QPoint pos = event->pos();
    if (pos.x() <= 0)
        pos.setX(1);
    if (pos.y() >= height())
        pos.setY(height() - 1);
    double rad = atan(((double)rect().bottom() - pos.y()) / pos.x());
    setAngle(qRound(rad * 180 / 3.14159265));
}
```

C'est un autre capteur d'événements Qt. Il est appelé quand l'utilisateur a appuyé sa souris dans le widget et qu'il la déplace. (Vous pouvez utiliser Qt pour envoyer des événements de mouvement de souris même quand aucun bouton n'est pressé. Regardez *QWidget::setMouseTracking()*.)

Ce capteur replace l'extrémité du canon par rapport à la position du curseur.

Premièrement, si l'embout n'est pas cliqué, nous retournons. Ensuite, nous regardons la position du curseur. Si le curseur est à gauche ou en dessous du widget, nous ajustons le position pour être dans le widget.

Ensuite nous calculons l'angle entre le bord bas du widget et la ligne imaginaire entre le coin en haut à gauche du widget et la position du widget. Finalement nous positionnons l'angle du canon à la nouvelle valeur valeur converti en degrés.

Souvenez vous que setAngle() redessine le canon.

```
void CannonField::mouseReleaseEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton)
        barrelPressed = false;
}
```

Le capteur d'événement Qt est appelé quand l'utilisateur relâche le bouton de la souris après l'avoir pressé dans le widget. Si le bouton gauche est relâché nous pouvons être sur que l'extrémité n'est plus cliquée. L'événement de dessin a droit à une ligne de plus :

```
    paintBarrier(painter);
```

paintBarrier() fait la même chose que *paintShot()*, *paintTarget()*, et *paintCannon()*

```
void CannonField::paintBarrier(QPainter &painter)
{
    painter.setPen(Qt::black);
    painter.setBrush(Qt::yellow);
    painter.drawRect(barrierRect());
}
```

Cette fonction privée dessine l'obstacle comme un rectangle rempli de jaune avec un contour noir.

```
QRect CannonField::barrierRect() const
{
    return QRect(145, height() - 100, 15, 99);
}
```

Cette fonction privée retourne le rectangle représentant l'obstacle. Nous fixons le bord bas de l'obstacle en bas du widget.

```
bool CannonField::barrelHit(const QPoint &pos) const
{
    QMatrix matrix;
    matrix.translate(0, height());
    matrix.rotate(-currentAngle);
    matrix = matrix.inverted();
    return barrelRect.contains(matrix.map(pos));
}
```

Cette fonction retourne *true* si le point est dans l'embout ; elle retourne *false* sinon. Nous utilisons ici la classe *QMatrix*. Cette classe définit un système de coordonnées. Elle permet les même transformations que *QPainter*. Nous appliquons les mêmes transformations que pour dessiner l'extrémité dans la fonction *paintCannon()*. D'abord nous translatons le système de coordonnées puis nous le tournons. Maintenant nous devons vérifier si le point *pos* (dans les coordonnées du widget) se situe dans l'extrémité. Pour cela, nous inversons la matrice de transformation. La matrice inversée inverse les transformations que nous avons utilisées pour dessiner l'embout. Nous localisons le point *pos* en utilisant la matrice inversée et retournons *true* s'il est dans le rectangle d'origine de l'embout.

- t14/gameboard.cpp

```
    QFrame *cannonBox = new QFrame;
    cannonBox->setFrameStyle(QFrame::WinPanel | QFrame::Sunken);
```

Nous créons et activons une *QFrame*, et lui assignons un style de cadre. Le résultat est un cadre 3D autour de *CannonField*.

```
    (void) new QShortcut(Qt::Key_Enter, this, SLOT(fire()));
    (void) new QShortcut(Qt::Key_Return, this, SLOT(fire()));
    (void) new QShortcut(Qt::CTRL + Qt::Key_Q, this, SLOT(close()));
```

Ici, nous créons et activons trois objets *QShortcut*. Ces objets interceptent les événements du clavier à un widget et appellent des slots si certaines touches sont pressées. Notez que un objet *QShortcut* est un enfant d'un widget et sera détruit quand le widget sera détruit. *QShortcut* lui même n'est pas un widget et n'a pas d'effet visible sur son parent.

Nous définissons 3 raccourcis. Nous voulons que le slot *fire()* soit appelé quand l'utilisateur presse *Entrée* ou *Retour*. Nous voulons aussi que l'application se ferme quand les touches *Ctrl* et *Q* sont simultanément pressée. Au lieu de se connecter à *QCoreApplication::quit()*, nous nous connectons à *QWidget::Close* cette fois. Maintenant que *GameBoard* est le widget principal de l'application, cela a le même effet que *quit()*.

Qt::CTRL, *Qt::Key_Enter*, *Qt::Key_Return* et *Qt::Key_Q* sont toutes des constantes déclarées dans l'espace de nomage de Qt.

```
    QVBoxLayout *cannonLayout = new QVBoxLayout;
    cannonLayout->addWidget(cannonField);
    cannonBox->setLayout(cannonLayout);

    QGridLayout *gridLayout = new QGridLayout;
    gridLayout->addWidget(quit, 0, 0);
    gridLayout->addLayout(topLayout, 0, 1);
    gridLayout->addLayout(leftLayout, 1, 0);
    gridLayout->addWidget(cannonBox, 1, 1, 2, 1);
    gridLayout->setColumnStretch(1, 10);
    setLayout(gridLayout);
```

Nous donnons à cannonBox son propre *QVBoxLayout*, et nous ajoutons *canonField* à cette disposition. *cannonField* devient implicitement l'enfant de *canonBox*. Puisqu'il n'y a rien d'autre dans la boite, l'effet sera que *QVBoxLayout* mettra un cadre autour de *CannonField*. Nous mettons *cannonBox*, pas *CannonField*, dans l'organisation de la grille.

XIV-B - Exécuter l'application

Le canon tire maintenant quand nous appuyons sur Entrée. Vous pouvez aussi positionner l'angle du canon avec la souris. L'obstacle rend le jeu un peu plus difficile. Nous avons maintenant un joli cadre autour de CannonField.

XIV-C - Exercices

Écrire un jeu de "space invaders". Cette exercice à initialement été fait par Igor Rafienko. Vous pouvez **télécharger son jeu**. Le nouvel exercice : Écrire un jeu de casse brique. Objectif final : Allez plus loin et créez des chefs-d'oeuvre de programmation.