

Débuter dans la création d'interfaces graphiques avec Qt 4



Traduction du tutoriel pour débutant de Nokia

par l'équipe Qt de Developpez.com ([la rubrique Qt](#)) ([Blog](#))

Date de publication : 18/01/2009

Dernière mise à jour : 26/01/2011

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, bien qu'elles soient souvent utilisées, ne sont pas couvertes par ce tutoriel d'introduction à Qt.

Pour toute question ou remarque, retrouvez ce tutoriel sur le forum.

I - Remerciements.....	4
II - Présentation.....	4
III - Hello World.....	5
III-A - Analyse du code ligne par ligne.....	5
III-B - Les variables d'environnement.....	6
III-C - Compiler l'application.....	7
III-D - Lancer l'application.....	7
III-E - Exercices.....	7
IV - Terminer proprement l'application.....	7
IV-A - Analyse du code ligne par ligne.....	8
IV-B - Lancer l'application.....	8
IV-C - Exercices.....	8
V - Les valeurs usuelles.....	9
V-A - Analyse du code ligne par ligne.....	9
V-B - Attention à l'ordre chronologique des événements.....	10
V-C - Lancer l'application.....	10
V-D - Exercices.....	11
VI - Soyons widgets.....	11
VI-A - Analyse du code ligne par ligne.....	12
VI-B - Lancer l'application.....	13
VI-C - Exercices.....	13
VII - Les widgets.....	13
VII-A - Analyse du code ligne par ligne.....	14
VII-B - Lancer l'application.....	15
VII-C - Exercices.....	15
VIII - Construire des widgets en vrac !.....	15
VIII-A - Analyse du code ligne par ligne.....	17
VIII-B - Lancer l'application.....	18
VIII-C - Exercices.....	18
IX - Une chose mène à une autre.....	19
IX-A - Analyse du code ligne par ligne.....	19
IX-A-1 - t7/lcdrange.h.....	19
IX-A-2 - t7/lcdrange.cpp.....	20
IX-B - Compiler l'application.....	21
IX-C - Exécuter l'application.....	21
IX-D - Exercices.....	21
X - Préparer le champ de bataille.....	21
X-A - Analyse du code ligne par ligne.....	22
X-A-1 - t8/lcdrange.h.....	22
X-A-2 - t8/lcdrange.cpp.....	22
X-A-3 - t8/cannonfield.h.....	23
X-A-4 - t8/cannonfield.cpp.....	23
X-A-5 - t8/main.cpp.....	24
X-B - Lancer l'application.....	27
X-C - Exercices.....	27
XI - Avec un canon, on peut tout faire.....	27
XI-A - Analyse du code ligne par ligne.....	28
XI-A-1 - t9/cannonfield.cpp.....	28
XI-B - Exécuter l'application.....	29
XI-C - Exercices.....	30
XII - Doux comme un agneau.....	30
XII-A - Analyse du code ligne par ligne.....	31
XII-A-1 - t10/connonfield.h.....	31
XII-A-2 - t10/cannonField.cpp.....	31
XII-A-3 - t10/main.cpp.....	32
XII-B - Exécuter l'application.....	33
XII-C - Exercices.....	33
XIII - Tire-lui dessus !.....	33

XIII-A - Analyse du code ligne par ligne.....	34
XIII-A-1 - t11/cannonfield.h.....	34
XIII-A-2 - t11/cannonfield.cpp.....	35
XIII-A-3 - t11/main.cpp.....	37
XIII-B - Exécuter l'application.....	37
XIII-C - Exercices.....	37
XIV - Accrochons des briques en l'air.....	37
XIV-A - Analyse du code ligne par ligne.....	38
XIV-A-1 - t12/lcdrange.h.....	38
XIV-A-2 - t12/lcdrange.cpp.....	39
XIV-A-3 - t12/cannonfield.h.....	40
XIV-A-4 - t12/cannonfield.cpp.....	40
XIV-A-5 - t12/main.cpp.....	42
XIV-B - Exécuter l'application.....	43
XIV-C - Exercices.....	43
XV - Game Over.....	43
XV-A - Analyse du code ligne par ligne.....	44
XV-A-1 - t13/lcdrange.cpp.....	44
XV-A-2 - t13/cannonfield.h.....	45
XV-A-3 - t13/cannonfield.cpp.....	45
XV-A-4 - t13/gameboard.h.....	46
XV-A-5 - t13/gameboard.cpp.....	47
XV-A-6 - t13/main.cpp.....	49
XV-B - Exécuter l'application.....	49
XV-C - Exercices.....	49
XVI - Face au mur.....	49
XVI-A - Analyse du code ligne par ligne.....	50
XVI-A-1 - t14/cannonfield.h.....	50
XVI-A-2 - t14/cannonfield.cpp.....	51
XVI-A-3 - t14/gameboard.cpp.....	53
XVI-B - Exécuter l'application.....	53
XVI-C - Exercices.....	53

I - 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.

Ce tutoriel est une traduction d'un des tutoriels écrits par **Nokia Corporation and/or its subsidiary(-ies)**, inclus dans la documentation de Qt 4.4.x en anglais. Les éventuels problèmes résultant d'une mauvaise traduction ne sont pas imputables à Nokia. Vous pouvez trouver le tutoriel original [ici](#).

Nous remercions la société Nokia pour sa confiance tout au long de ce travail de traduction.

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

Un merci tout particulier à **dourouc05** et **difredo** pour leur participation et relecture tout au long de la traduction, ainsi qu'à **Papy_77** pour le second passage et à **Jacques Thery** pour ses petites corrections ponctuelles !

Et, surtout, **toutes nos félicitations et un grand bravo** aux acteurs principaux de cette traduction. Par chapitre :

- Présentation : **John42**
- Hello World : **haraelendil**
- Terminer proprement l'application : **Spout**
- Les valeurs usuelles : **Spout**
- Soyons widgets : **Spout**
- Les widgets : **amoweb**
- Construire des widgets en vrac : **buggen25**
- Une chose mène à une autre : **buggen25**
- Préparer le champ de bataille : **buggen25**
- Avec un canon, on peut tout faire : **Spout**
- Doux comme un agneau : **buggen25**
- Tire-lui dessus : **amoweb**
- Accrochons des briques en l'air : **Spout**
- Game Over : **dourouc05**
- Face au mur : **signix**

II - 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, bien que 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, où 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 `exemples/tutorials/tutorial` de Qt 4.4. Pour les autres versions, vous pouvez télécharger les sources sous la forme d'une archive [Source RAR](#) ou [Source TAR.BZ2](#).

Chapitres :

- 1 Hello World ;
- 2 Terminer proprement l'application ;
- 3 Valeurs usuelles ;
- 4 Soyons widgets ;

- 5 Les widgets
- 6 Construire des widgets en vrac !
- 7 Une chose mène à une autre ;
- 8 Préparer le champ de bataille ;
- 9 Avec un canon, on peut tout faire ;
- 10 Doux comme un agneau ;
- 11 Tire-lui dessus !
- 12 Accrochons des briques en l'air ;
- 13 Game Over ;
- 14 Face au mur.

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

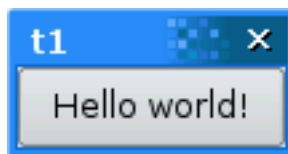
III - Hello World

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();
}
```

III-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 toute application qui possède une IHM avec Qt. QApplication 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 composant de l'interface graphique qui peut gérer une entrée de l'utilisateur et dessiner. Le programmeur peut changer tout le *look and feel* du widget ou seulement une partie de celui-ci (la couleur, par exemple), ainsi que son contenu. Un QPushButton peut afficher du texte ou un QIcon.

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

La fonction main() est le point d'entrée du programme. Le plus souvent, l'utilisation de cette fonction ne nécessite que quelques initialisations avant de passer la main à la bibliothèque Qt, qui indique ensuite au programme, par le biais d'événements, les actions demandées par l'utilisateur. 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 toute utilisation d'une quelconque fonctionnalité d'IHM de Qt.

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

Après la création de la QApplication, voilà la première ligne de code liée à l'IHM : un bouton est créé. Le bouton affiche le texte "Hello world!". Comme, dans le constructeur de QPushButton, on n'a pas spécifié un second argument pour préciser la fenêtre parente du bouton, ce 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 événements système et les renvoie aux widgets appropriés. Vous devriez maintenant compiler et lancer le programme.

III-B - Les variables d'environnement

Si vous utilisez la ligne de commande pour développer des applications Qt, vous devez vous assurer que les bibliothèques 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 vous-même la variable *PATH*.

III-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, en modifiant et en recompilant vous-même ces exemples, vous en apprendrez beaucoup sur la façon d'utiliser Qt. Dans le cas où vous auriez copié le(s) fichier(s) *.cpp* et *.h* d'un exemple dans un autre répertoire pour y apporter vos modifications, la prochaine étape sera de créer un *makefile* pour Qt dans ce répertoire. La création d'un *makefile* pour Qt utilise la commande *qmake*, un outil de compilation fourni avec Qt. Pour créer un *makefile*, lancez les deux commandes suivantes dans le répertoire contenant vos sources modifiées : **qmake -project** puis **qmake**. La première commande ordonne à 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. Vous n'avez plus qu'à lancer *make* (*nmake* si vous êtes sous Visual Studio) pour compiler le programme et vous pourrez ensuite lancer votre première application Qt !

III-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!".

III-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*).

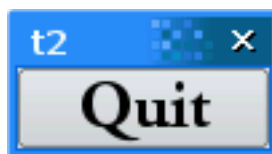
IV - Terminer proprement l'application

[Source](#) **t2.rar**

Fichiers

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

Cet exemple constitue 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();  
}
```

IV-A - Analyse du code ligne par ligne

```
#include <QFont>
```

Comme ce programme utilise la classe QFont, l'en-tête QFont doit 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 pu également 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 indirectement). 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`. Par conséquent, si on clique sur le bouton, l'application se terminera.

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

IV-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.

IV-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.

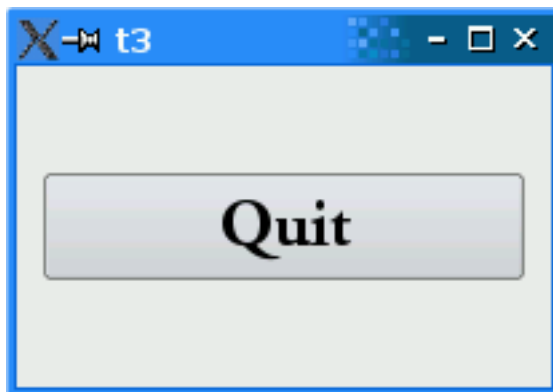
V - Les valeurs usuelles

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 seulement 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();
}
```

V-A - Analyse du code ligne par ligne

```
#include <QWidget>
```

Nous réalisons une inclusion de QWidget qui est la classe parente de tous les widgets que nous allons pouvoir utiliser.

```
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 (en français : IU ou IHM ; en anglais : UI ou GUI, respectivement). Un widget est le plus petit composant de l'interface utilisateur. Il reçoit les événements de la souris, du clavier ou de tout autre élément issu du système de fenêtrage et dessine une représentation de lui-même à l'écran. Il reçoit les événements de la souris, du clavier ou de tout autre élément issu du système de fenêtrage. Un widget est rattaché à son widget parent et aux widgets qui sont ses descendants.

Un widget qui n'est pas intégré dans un widget parent, comme ce widget particulier, est appelé *window*. Habituellement, les windows ont leur propre cadre ("*frame*") et barre des tâches, fournis par le système de fenêtrage. Un widget sans widget parent est toujours 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);
```

On crée un `QPushButton`, widget enfant dont le parent est window. Un widget enfant est toujours affiché à l'intérieur de la zone occupée par son parent et donc dans le cadre des 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()` nécessite quatre arguments : les deux premiers sont les coordonnées x et y du coin supérieur gauche du bouton. Ces coordonnées sont relatives au widget parent. Les deux derniers arguments sont la largeur et la hauteur du bouton. Ceci produit 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 enfants va être appelée (sauf pour ceux qui ont été explicitement cachés par `QWidget::hide()`).

V-B - Attention à l'ordre chronologique des événements

Les lecteurs de ce tutoriel qui ont déjà étudié `QObject` doivent 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 champ et que le destructeur supprime quit parce que c'est un enfant, puis une seconde fois quand quit lui-même sort du champ. Mais il n'y a pas lieu de s'en soucier ; 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 [la note sur l'ordre de construction/destruction des QObject](#).

V-C - Lancer l'application

Le bouton, au lieu de remplir toute la fenêtre, est à présent situé dans la fenêtre à la position (10,40), grâce à l'appel à `QWidget::setGeometry()`.

V-D - Exercices

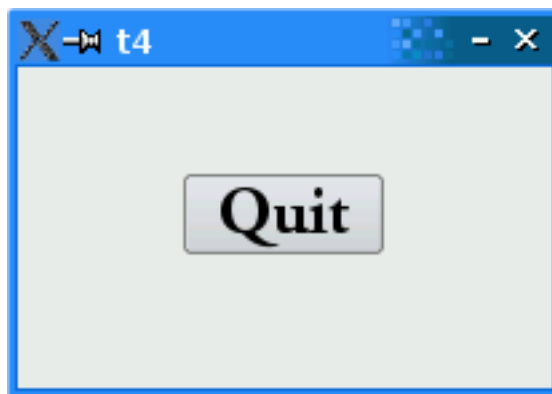
Essayez de redimensionner la fenêtre. Comment se comporte le bouton ? Que devient pour la hauteur du bouton si vous lancez le programme avec une police plus grosse ? Que se passe-t-il si vous essayez de définir une fenêtre toute petite ?

VI - Soyons widgets

Source [t4.rar](#)

Fichiers

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



Cet exemple montre comment on peut créer un widget personnalisé et en contrôler les tailles minimale et maximale

```
#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();
}
```

VI-A - Analyse du code ligne par ligne

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

Ici nous créons une nouvelle classe. Comme cette classe hérite de `QWidget`, cette nouvelle classe est un widget et peut être, soit une "fenêtre de premier plan", soit 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 pour la plupart des widgets, ce constructeur ne fait que transmettre le parent au constructeur de `QWidget`.

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

Comme ce widget ne sait pas comment gérer le redimensionnement, nous fixons sa taille. Dans le prochain chapitre, nous verrons comment un widget peut répondre à un événement de redimensionnement déclenché par 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 nouveau widget, enfant de ce widget (le parent de ce nouveau widget est `this`, c'est-à-dire l'instance de `MyWidget`).

L'appel à la fonction `tr()` pour le texte "Quit" désigne ce texte comme "destiné à être traduit", rendant ainsi possible son remplacement ultérieur à l'exécution à partir du contenu d'un fichier de traduction. C'est une bonne habitude, au cas où vous décideriez plus tard de traduire votre application dans d'autres langues, que d'utiliser `tr()` pour tous les textes visibles par l'utilisateur.

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

L'appel à `QWidget::setGeometry()` définit 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 déclarée dans le fichier d'en-tête `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 nousinstancions notre nouvel enfant, nous l'affichons et nous lançons l'application.

VI-B - Lancer l'application

Ce programme a un comportement similaire au précédent. La différence réside dans la manière dont nous l'avons implémenté. Aussi, il réagit un peu différemment. Essayez juste de le redimensionner pour voir.

VI-C - Exercices

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

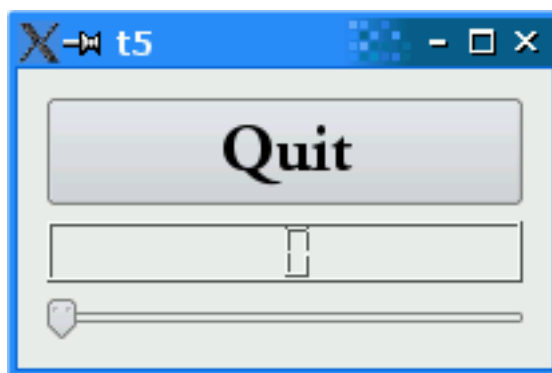
Essayez d'ajouter plus de boutons ou mettez d'autres widgets que QPushButton.

VII - Les widgets

[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 des signaux et des 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();
}
```

VII-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, c'est-à-dire un widget qui affiche un nombre sous la forme d'un écran LCD. Cette instance est prévue pour afficher deux chiffres. Donner à QLCDNumber::segmentStyle la valeur QLCDNumber::Filled rend le LCD plus lisible.

Anecdote historique : QLCDNumber est le premier widget écrit pour Qt, à une époque où 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 choisir une valeur entière comprise dans un certain intervalle ("range", en anglais). Ici nous en créons un dans l'intervalle [0, 99], avec 0 pour valeur initiale.

```
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. Chaque fois que le curseur ("`slider`") change de valeur, il envoie cette nouvelle valeur en émettant le signal `valueChanged()`. Comme le slot `display()` de `lcd` est connecté à `valueChanged()` de `slider`, il est appelé 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 donc 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 besoin de spécifier les coordonnées de chaque widget comme nous le faisons 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()`.

Cela fait du layout un enfant de `MyWidget`. Par conséquent, nous n'aurons pas à nous soucier de la suppression du layout, le lien de parenté assurant sa suppression en même temps que celle de `MyWidget`. En outre l'appel de la méthode `QWidget::setLayout()` assure automatiquement que les widgets sont considérés comme des enfants de `MyWidget`. Nous n'avons donc pas besoin de spécifier les liens de parenté.

Avec Qt, un widget est soit enfant d'un autre widget (comme *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 pointeur sur `QWidget` comme parent et `QLayout` n'hérite pas de `QWidget`.

VII-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 composants gardent à peu près la même taille.

VII-C - Exercices

Essayez d'ajouter plus de chiffres sur l'écran LCD ou de changer la base numérale (binaire, octale ou hexadécimale au lieu de décimale) du nombre affiché sur l'écran LCD. Vous pouvez même ajouter quatre boutons pour permettre à l'utilisateur de changer à volonté cette base.

Essayez d'ajouter plus de chiffres sur l'écran LCD ou essayez, avec `QLCDNumber::setMode()`, de changer la base numérale (binaire, octale ou hexadécimale

Vous pouvez aussi changer l'intervalle du `slider`.

Peut-être vaudrait-il mieux utiliser `QSpinBox` plutôt que `QSlider` ?

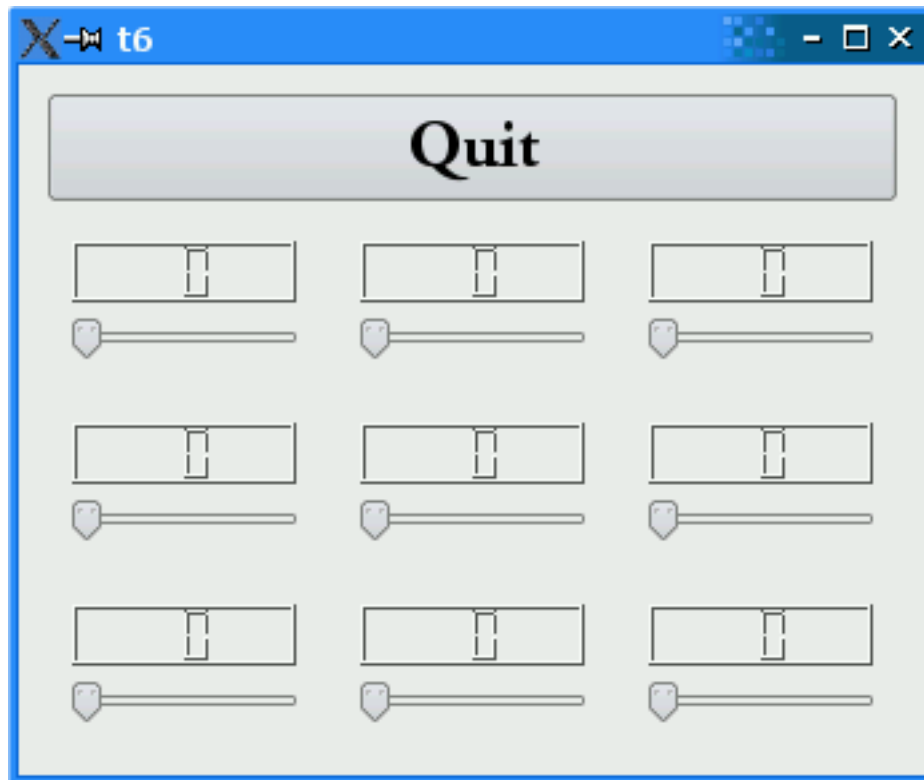
Essayez de quitter l'application quand le nombre affiché par l'écran LCD est en dépassement de capacité ("*overflow*").

VIII - 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();
}
```

VIII-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 emprunté au constructeur MyWidget du chapitre VII. La seule différence réside dans le fait que le bouton Quit a été enlevé et que la classe a été renommée.

```
class MyWidget : public QWidget
{
public:
```

```
MyWidget(QWidget *parent = 0);  
};
```

MyWidget, lui aussi, ne contient aucune méthode : il 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 était jusqu'à maintenant inclus dans ce qui s'appelle à présent LCDRange n'en fait plus partie ; désormais on a un bouton Quit unique 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 en colonnes : vous pouvez spécifier les numéros des lignes et des colonnes quand vous ajoutez des widgets à l'objet QGridLayout qui va se charger de les disposer 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);
```

Pour terminer, on ajoute à l'outil de disposition principal 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.

VIII-B - Lancer l'application

Ce programme montre la facilité d'utilisation simultanée de plusieurs widgets. Comme dans le chapitre précédent, chacun se comporte comme l'association d'un curseur et d'un affichage numérique LCD. Là encore, la différence réside dans l'implémentation.

VIII-C - Exercices

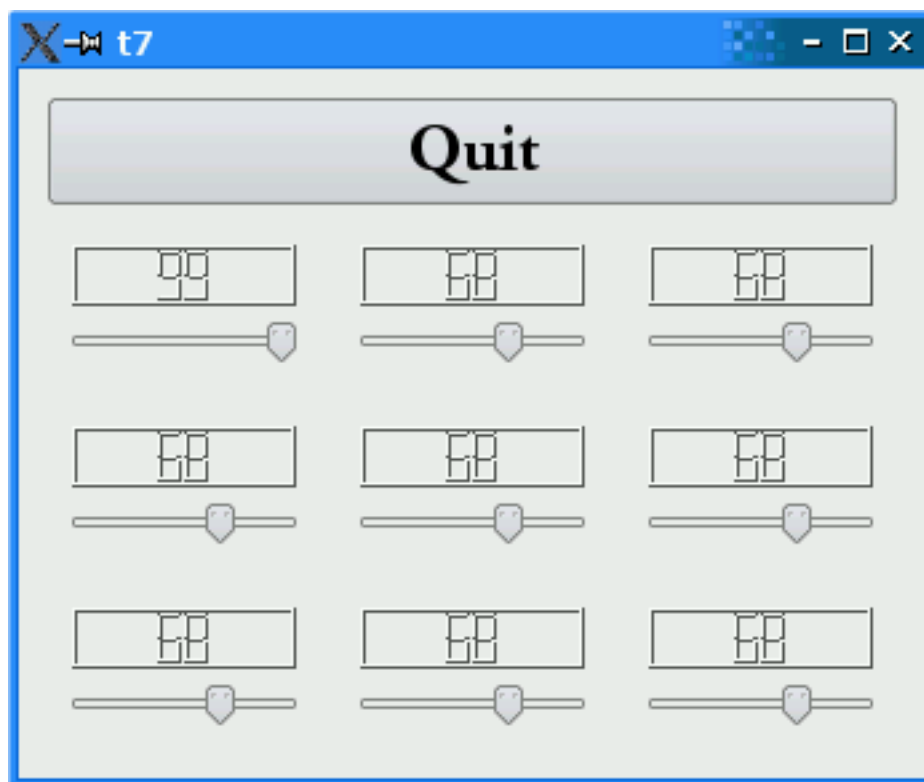
Au lancement de l'application, initialisez chaque curseur avec une valeur différente et/ou aléatoire.

IX - Une chose mène à une autre

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, le code source est divisé en plusieurs fichiers que nous avons placés dans le répertoire **exemples/tutorial/t7**.

IX-A - Analyse du code ligne par ligne

IX-A-1 - t7/lcdrange.h

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

Ces 2 lignes de code, combinées avec le `#endif` placé en fin de fichier, correspondent à une construction standard du langage C++ qui évite toute erreur dans le cas où le fichier en-tête viendrait à être inclus plusieurs fois. Si vous ne pratiquez pas encore cette syntaxe, prenez cette bonne habitude en tant que développeur.

```
#include <QWidget>
```

`QWidget` se trouve inclus du fait que notre `LCDRange` hérite de `QWidget`. On doit toujours inclure l'en-tête de la classe parente ; dans les tutoriaux précédents, on a un peu "triché" en incluant le fichier `QWidget` d'une manière indirecte, via d'autres fichiers en-tête.

```
class QSlider;
```

Ceci est une autre astuce classique, moins utilisée cependant 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 en-tête, et on utilise le fichier en-tête de QSlider dans le fichier .cpp.

Ceci accélère la compilation des grands projets, car le compilateur passe la plus grande partie de son temps à parcourir les fichiers en-tête ; ce n'est pas le cas avec le code actuel. Cette seule astuce augmente la vitesse de compilation d'au moins un facteur deux.

```
class LCDRange : public QWidget
{
    Q_OBJECT

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

Noter Q_OBJECT. Cette macro doit être déclarée dans toutes les classes contenant des signaux et des slots. Si vous êtes curieux, sachez qu'elle définit 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 permettant d'accéder à la valeur de LCDRange ; setValue() est notre premier slot et valueChanged() est notre premier signal personnalisé.

Les slots doivent être implémentés comme d'habitude (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 par les classes dérivées de cette dernière).

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

IX-A-2 - 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 fait que LCDRange émette le signal valueChanged() au changement de sa valeur engendre un enchaînement de signaux et de slots.

IX-B - Compiler l'application

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

```
qmake -project  
qmake
```

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

IX-C - Exécuter l'application

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

IX-D - Exercices

Utilisez le curseur du coin inférieur droit pour régler tous les LCD sur 50. Ensuite, réglez les six premiers sur 30 en cliquant sur le curseur de la ligne inférieure. Maintenant, utilisez le curseur situé à gauche du dernier curseur manipulé pour remettre les cinq premiers LCD à 50.

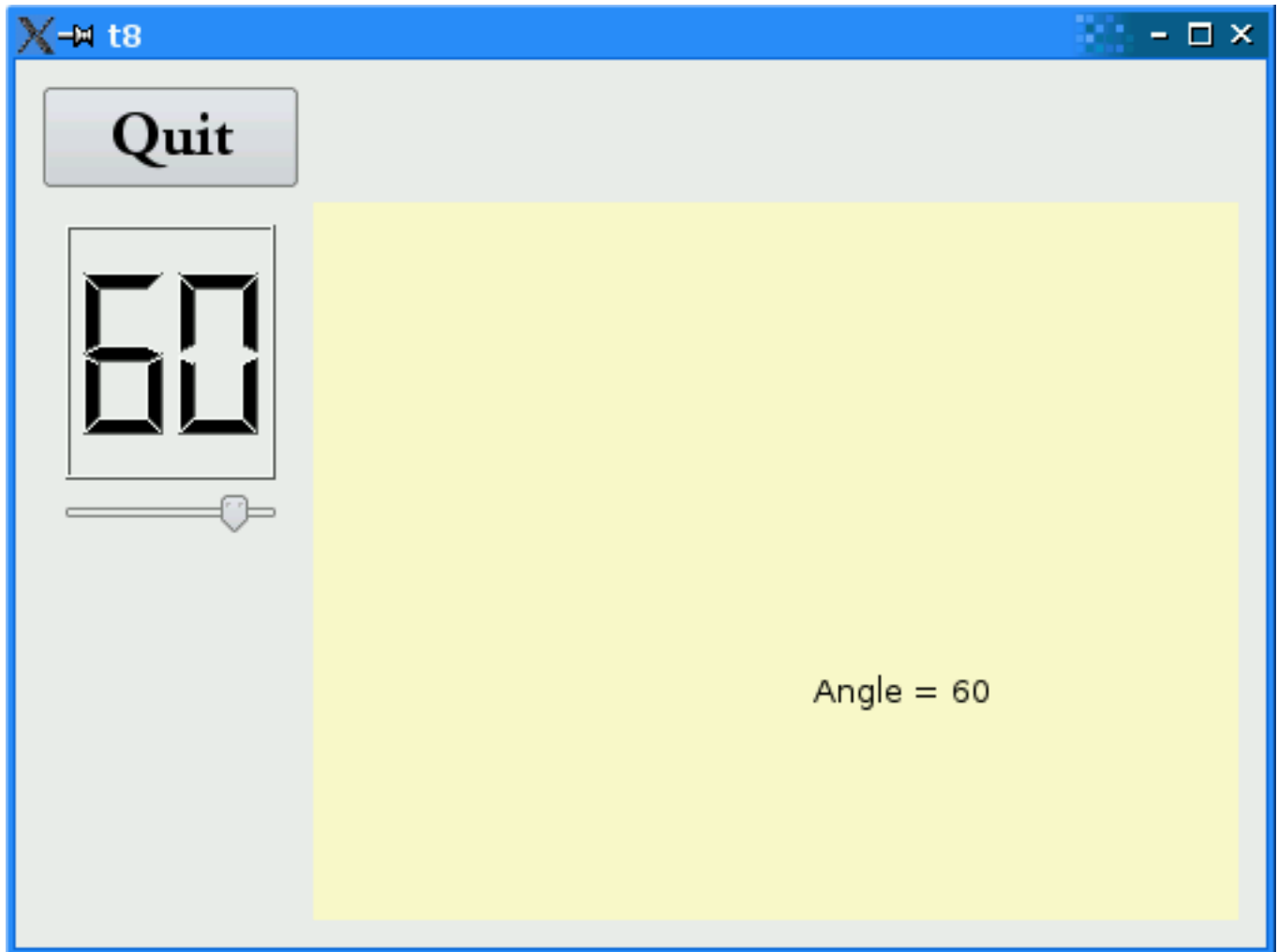
Maintenant cliquez sur la gauche du bouton du slider inférieur droit. Que se passe-t-il ? Pour quelle raison ce comportement est normal ?

X - 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 de lui-même. On va aussi ajouter quelques interactions avec le clavier (en deux lignes de code).

X-A - Analyse du code ligne par ligne

X-A-1 - t8/lcdrange.h

Ce fichier est très proche du fichier lcdrange.h du Chapitre IX. Nous n'avons ajouté qu'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.

X-A-2 - 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 du curseur dans **LCDRange**. Puisqu'on a paramétré **QLCDNumber** pour qu'il puisse 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 le souhaitez, vous pouvez aussi utiliser votre propre gestionnaire d'erreur en utilisant la fonction **qInstallMsgHandler()**.

X-A-3 - t8/cannonfield.h

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

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

CannonField hérite de **QWidget**. On utilise la même syntaxe 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 la même syntaxe que pour la valeur de **LCDRange**.

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

On trouve ici un second exemple de gestionnaire d'événement : il fait partie de la multitude de gestionnaires qu'on peut rencontrer dans **QWidget**. Cette fonction virtuelle est appelée quand un widget a besoin de se redessiner (c'est-à-dire de repeindre sa propre surface).

X-A-4 - t8/cannonfield.cpp

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

Ici encore, on utilise la même syntaxe que dans le **LCDRange** du 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 couleur d'arrière plan et choisie selon le besoin (pour ce widget, seules les couleurs d'arrière plan et de texte sont utilisées). Ensuite, on appelle **setAutoFillBackground(true)** pour demander à Qt de remplir automatiquement l'arrière plan.

QColor est spécifié comme un triplet **RVB** (rouge-vert-bleu), où chaque composante (R, V et B) est comprise entre 0 (luminosité minimale) et 255 (luminosité maximale). Au lieu de spécifier une valeur RVB, on aurait pu aussi utiliser une couleur prédéfinie (par exemple **Qt::yellow**).

```
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 serait en dehors de cet intervalle.

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

Pour terminer, on émet le signal **angleChanged()** pour signaler au monde extérieur que l'angle a changé. Le mot clé **emit** est un élément syntaxique propre à Qt qui n'appartient donc 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, la région du widget qui doit être rafraîchie. Pour le moment, soyons paresseux et redessignons tout le widget.

Notre code affiche la valeur de l'angle à une certaine position dans le widget. Pour réaliser cela, on crée un **QPainter**, qui opère sur le widget **CannonField** et que l'on utilise pour dessiner 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.

X-A-5 - 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.

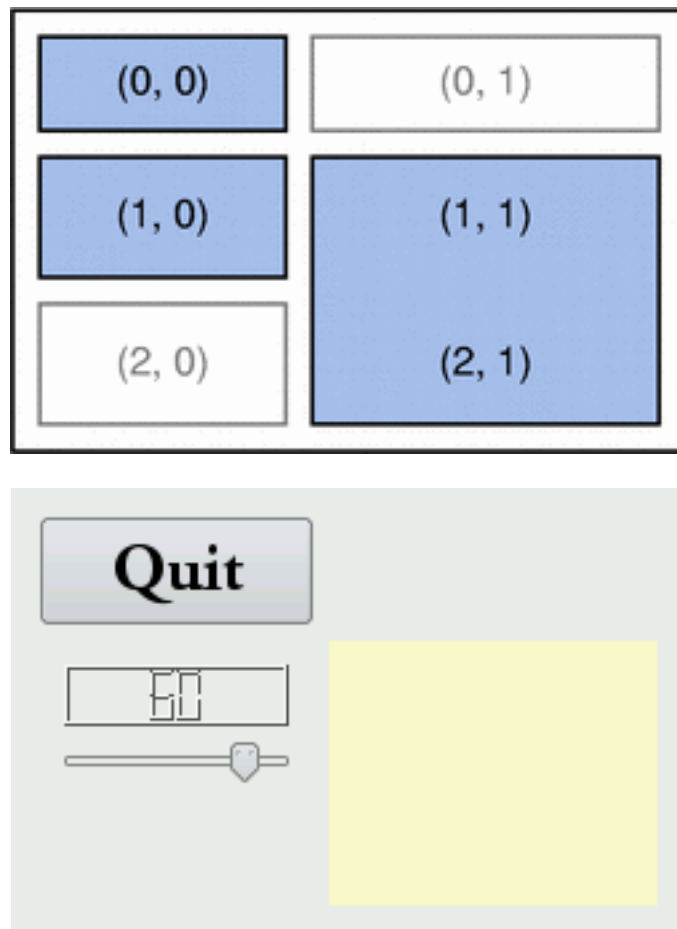
Ceci illustre la puissance de la programmation par composants quand elle met en oeuvre une encapsulation appropriée.

Noter qu'il est important de n'émettre le signal **valueChanged()** que dans le cas où l'angle change vraiment. Si les deux widgets **LCDRange** et **cannonField** avaient omis cette précaution, le programme entrerait dans une boucle infinie

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

Jusqu'à présent, on a utilisé **QVBoxLayout** pour gérer la géométrie mais on voudrait à présent un meilleur contrôle de la disposition des widgets : pour cela, on va passer à une disposition plus puissante grâce à 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**, car il va déterminer lui-même 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 à laquelle on veut aboutir : le côté gauche montre un schéma de la disposition et le côté droit montre une capture d'écran actuelle du programme.

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

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

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

On ajoute l'angle **LCDRange** dans 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, valeur par défaut), **QGridLayout** va essayer de garder inchangée la taille des widgets à gauche et va redimensionner le **CannonField** quand la taille de MyWidget changera.

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, en choisissant les facteurs d'extensibilité adéquats, utiliser ces facteurs pour spécifier qu'une colonne ou une ligne s'étend plus rapidement qu'une autre.

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

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

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

Notre dernière intervention consiste à donner le focus au widget **angle**, de sorte que toute action sur le clavier ira vers le widget **LCDRange** par défaut.

LCDRange ne contient aucun **keyPressEvent()** : il semblerait que ceci n'ait pas été jugé vraiment intéressant ! Cependant, son constructeur possède une nouvelle ligne.

```
setFocusProxy(slider);
```

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

X-B - Lancer l'application

Maintenant, le clavier peut être utilisé : les flèches **Home**, **End**, **PageUp** et **PageDown** sont vraiment opérationnelles.

Quand le curseur est manipulé, **CannonField** affiche un nouvel angle. Au redimensionnement, **CannonField** prend tout l'espace disponible.

X-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 redimensionnez la fenêtre ?

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

Essayez de remplacer « Quit » par « &Quit », quel changement cela produit-il sur le look du bouton ? Que se passe-t-il si on appuie sur **Alt+Q** pendant l'exécution du programme ?

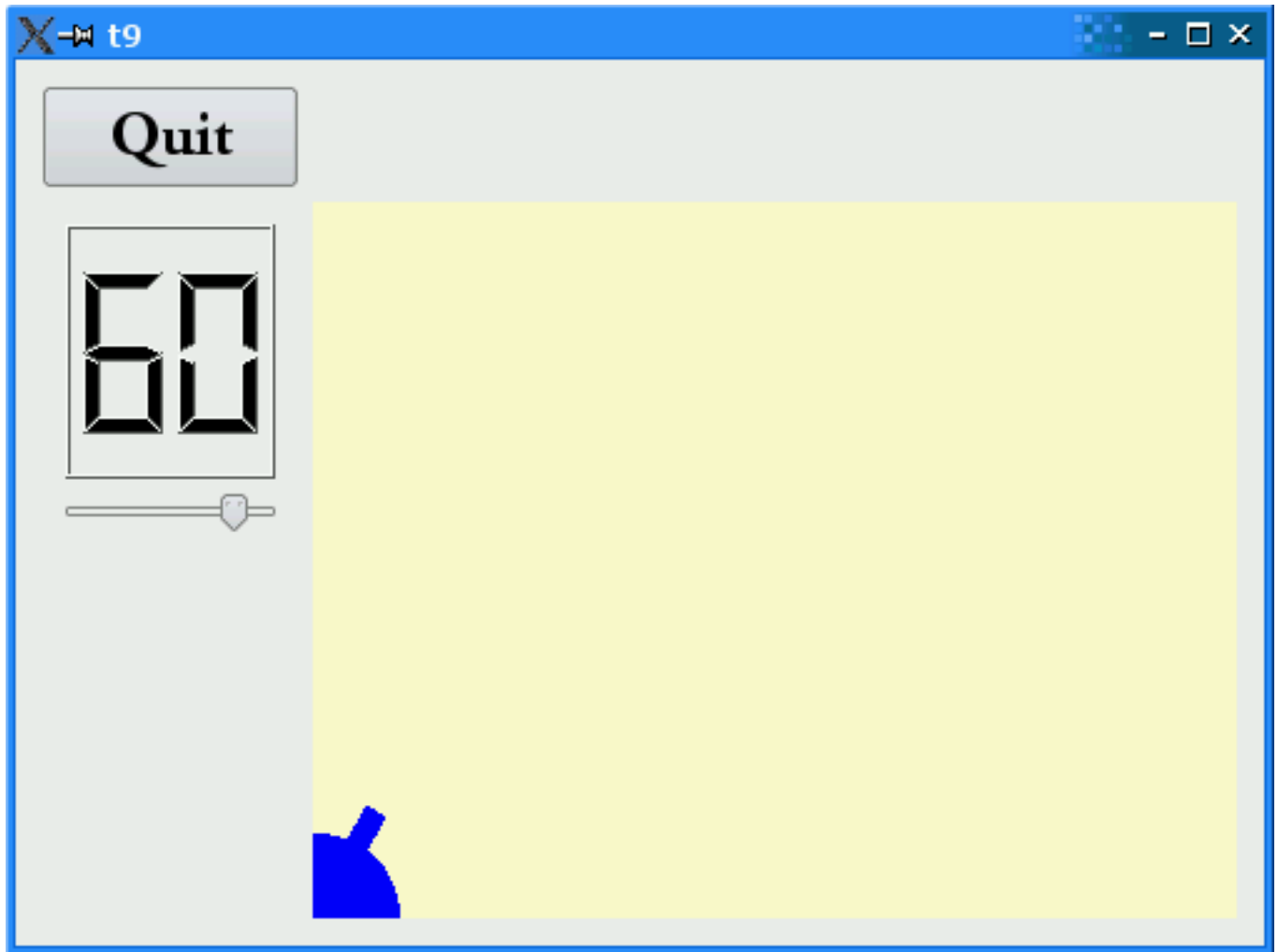
Centrez le texte sur **CannonField**.

XI - 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.

XI-A - Analyse du code ligne par ligne

XI-A-1 - t9/cannonfield.cpp

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

Nous allons maintenant commencer à utiliser sérieusement QPainter. Nous créons un outil de peintre ("painter") qui va permettre de travailler sur ce *widget*.

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

Les bords du dessin de QPainter sont toujours faits avec un 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 réalisant des aplats de couleur bleue. Nous aurions également pu utiliser un motif. La brosse bleue va remplir toute la surface limitée par les traits que nous allons dessiner.

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

Cette fonction traduit le système de coordonnées de `QPainter` (il le définit par une translation). 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 un rectangle (spécifié par quatre données) 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 dans la direction horaire correspondant à 3 heures. Le sens du dessin est anti-horaire (trigonométrique). 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, exprimé en degrés (et non 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 est souvent difficile de prévoir 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 ressemblé 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 :



XI-B - Exécuter l'application

Quand la position du curseur (*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.

XI-C - Exercices

Au lieu de spécifier Qt::NoPen, définissez un "vrai" crayon.

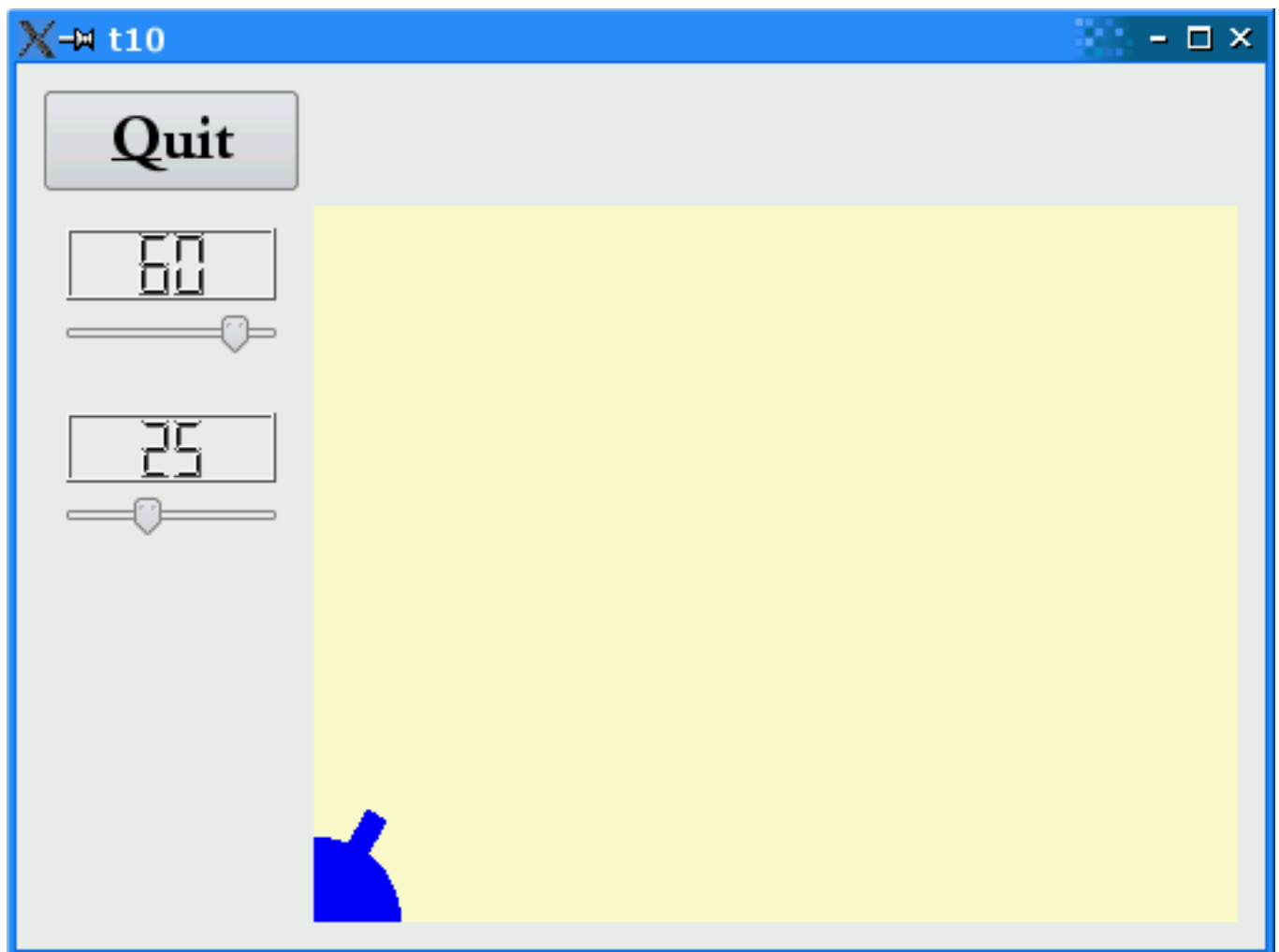
Essayez "Q&uit" ou "Qu&it" comme texte de bouton au lieu de "&Quit". Que se passe-t-il ?

XII - 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 ajoute le contrôle de la puissance du canon.

XII-A - Analyse du code ligne par ligne

XII-A-1 - t10/cannonfield.h

Le **CannonField** possède les attributs force (puissance de tir) et angle (angle de tir), 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 puissance se définit de la même manière que celle de l'angle

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

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

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

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

XII-A-2 - t10/cannonField.cpp

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

La puissance 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 de `setAngle()`. La seule différence réside dans le fait que la valeur de la puissance n'a pas à être représentée ; on n'a donc pas besoin, quand la puissance change, 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 dessine comme au chapitre XI.

```
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).

XII-A-3 - t10/main.cpp

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

Le constructeur demeure globalement le même : on a juste ajouté quelques détails.

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

On a ajouté un nouveau `LCDRange` qui va être utilisé pour indiquer la puissance.

```
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 l'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);
```

Au chapitre IX, on avait placé le contrôle de l'angle dans la cellule du coin inférieur gauche. À présent, on voudrait deux widgets dans cette cellule. On crée donc une boîte de disposition verticale, on met cette boîte dans la cellule et on ajoute **angle** et force dans cette boîte.

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

On initialise la puissance à 25.

XII-B - Exécuter l'application

On peut maintenant contrôler la valeur de la puissance.

XII-C - Exercices

Faire varier la taille du canon en fonction de la puissance.

Placer le canon dans le coin inférieur droit.

Essayez encore d'améliorer l'interface. Par exemple, faites en sorte que les touches **+** et **-** du clavier permettent d'augmenter ou de diminuer la puissance et que la touche **entrée** déclenche le tir. Si vous êtes troublé par le comportement des touches **Left** et **Right**, vous pouvez aussi le changer.

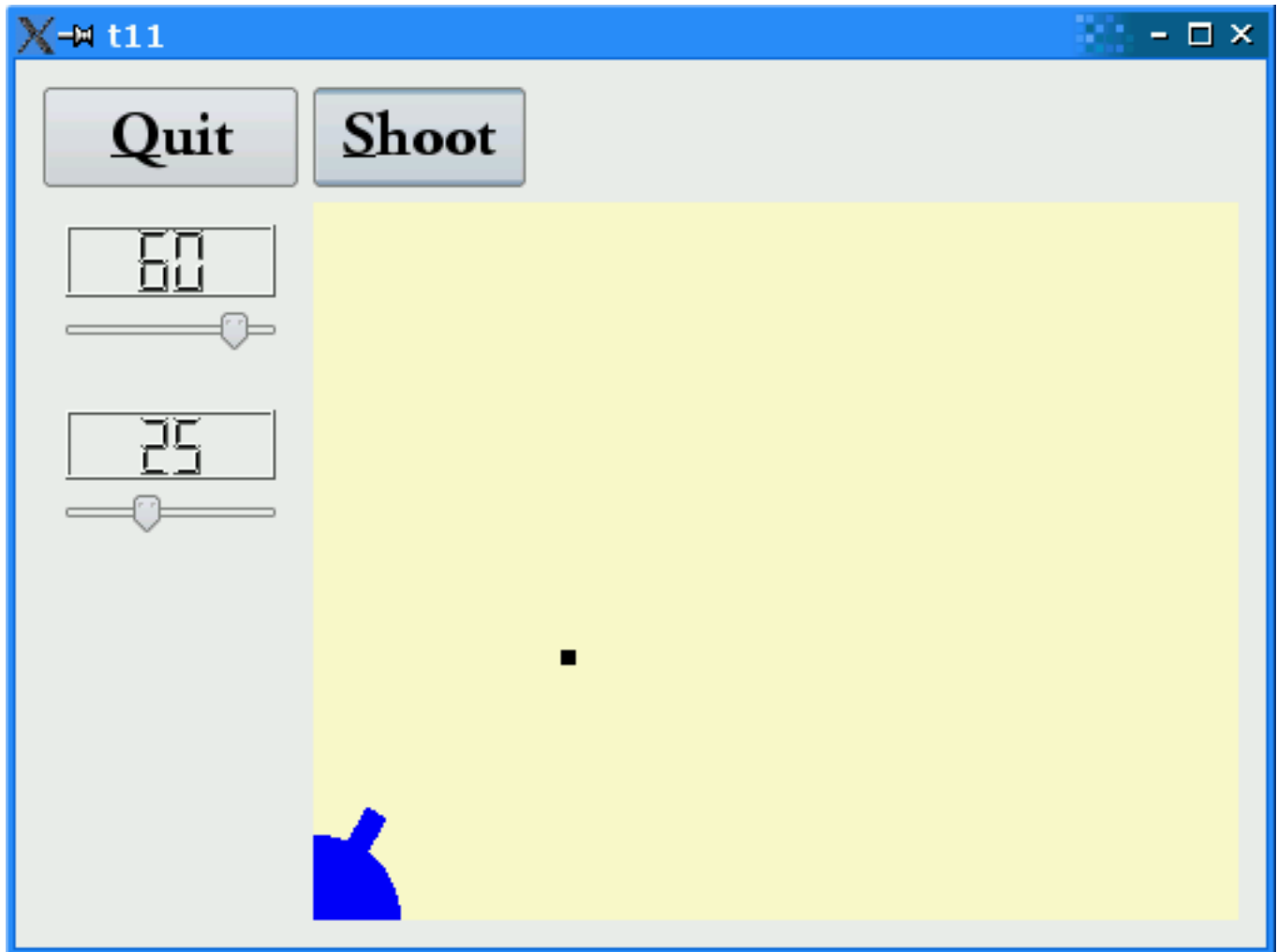
Remarque : réimplémentez **QWidget::KeyPressEvent()**.

XIII - 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.

XIII-A - Analyse du code ligne par ligne

XIII-A-1 - 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 déjà dans les airs.

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

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

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

Cette fonction privée dessine le tir.

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

Cette fonction privée renvoie, si un projectile est en l'air, le rectangle qui contient la trajectoire de tir ; sinon, 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 écoulé depuis le dernier tir. *shootAngle* représente l'angle du canon et *shootForce* la puissance du canon.

XIII-A-2 - 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 l'en-tête `cmath`, qui est plus moderne. Malheureusement certaines plateformes UNIX ne le supportent toujours pas correctement.

```
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 déplaçons le projectile chaque fois que le *timer* du canon dépasse le délai choisi (5 millisecondes ; voir ci-dessous).

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

Cette fonction déclenche le tir, sauf si un projectile est déjà en l'air. `timerCount` est remis à zéro, `shootAngle` et `shootForce` sont définis avec les valeurs d'angle et de puissance choisies. Et, enfin, 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 projectile ; il est appelé toutes les 5 millisecondes pendant que `QTimer` tire. Ses tâches sont de calculer la nouvelle position, de mettre à jour l'écran en dessinant le projectile à sa nouvelle position et, si nécessaire, d'arrêter le timer. Pour commencer, nous créons une `QRegion` qui conserve le rectangle de tir précédent. Une `QRegion` est capable de garder les caractéristiques de n'importe quelle région ; ici, nous en utilisons une pour simplifier le dessin. `shotRect()` renvoie le rectangle du slot. Ce sera expliqué en détail un peu plus bas. Ensuite, nous incrémentons le `timerCount` qui a pour effet de déplacer « *d'un cran* » le projectile sur sa trajectoire. Puis nous déterminons le nouveau rectangle de tir. Si le projectile a franchi le bord droit ou le bord inférieur du widget, nous arrêtons le timer ; sinon, nous ajoutons le nouveau `shotRect()` à la `QRegion`. Enfin, nous redessignons la `QRegion`, ce qui va envoyer un simple événement "paint" pour la mise à jour des seuls rectangles qui en ont besoin.

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

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

Par rapport au chapitre précédent, l'événement "paint" a été simplifié. La majeure partie de la logique a été déplacée 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 laissons 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 calcule les coordonnées du centre du projectile et renvoie le rectangle contenant le projectile. Il tient compte de la puissance initiale et de l'angle du canon 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 relativiste 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 6 x 6 que nous centrons (à l'aide des points que nous avons calculés). En même temps, nous convertissons le point en coordonnées

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()` convertit un nombre double (réel en double précision) dans l'entier integer le plus proche.

XIII-A-3 - t11/main.cpp

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

On ajoute simplement un bouton poussoir nommé *Shoot*.

```
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'avons fait avec le bouton *Quit*.

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

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

XIII-B - Exécuter l'application

Le canon peut tirer mais il n'a pas de cible.

XIII-C - Exercices

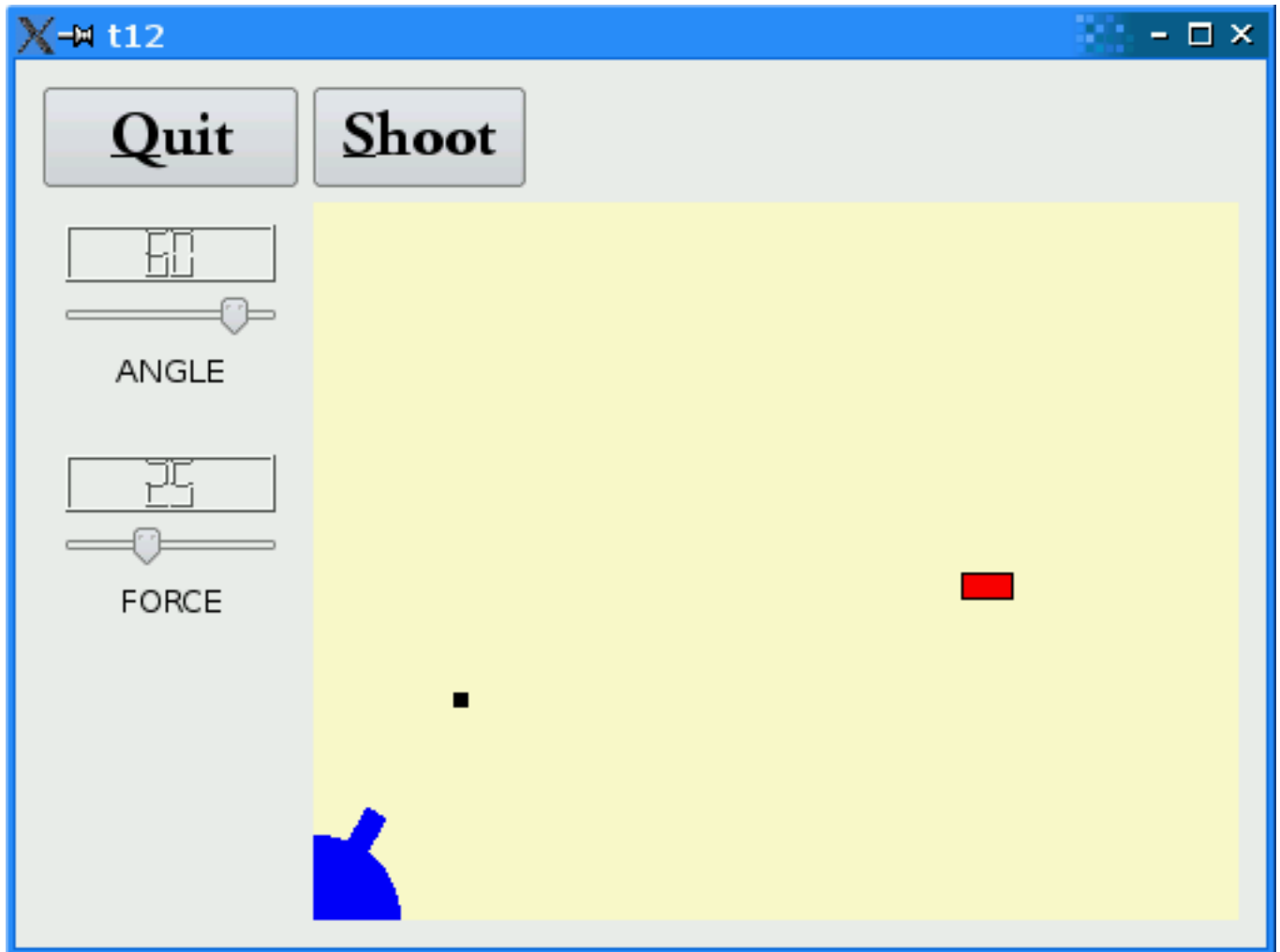
Représentez le tir par un cercle (suggestion : `QPainter::drawEllipse()` peut aider). Changez la couleur du canon quand il vise le ciel.

XIV - 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 une cible sur laquelle on pourra tirer.

XIV-A - Analyse du code ligne par ligne

XIV-A-1 - `t12/lcdrange.h`

`LCDRange` possède maintenant un label.

```
class QLabel;
class QSlider;
```

Nous déclarons à l'avance `QLabel` et `QSlider` car nous voulons dans la définition de la classe utiliser des pointeurs sur cette classe. Nous pourrions également utiliser `#include`, mais cela ralentirait inutilement la compilation.

```
class LCDRange : public QWidget
{
    Q_OBJECT

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

Nous avons ajouté un nouveau constructeur qui permet de fournir un label à cette classe 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.

XIV-A-2 - 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 demandons que son contenu soit aligné : au centre dans le sens horizontal, et en haut dans le sens vertical. L'appel à *QObject::connect()* a également été emprunté au 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.

XIV-A-3 - 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 projectile va au-delà des bords droit et bas du widget (il est alors certain qu'il n'a pas atteint et qu'il n'atteindra 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.

XIV-A-4 - t12/cannonfield.cpp

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

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

```
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 en principe toujours les mêmes séries de nombres à chaque fois que vous lancez un programme ; utilisée normalement, elle ferait donc toujours apparaître la cible à la même position. Pour l'éviter, nous devons transmettre un germe initial aléatoire au premier appel de la fonction. Ce germe doit lui aussi être aléatoire afin d'éviter des séries de nombres aléatoires identiques. La solution adoptée ici est d'utiliser comme germe pseudo-aléatoire le nombre de secondes qui se sont écoulées depuis minuit.

D'abord nous créons une variable booléenne locale statique. Nous avons l'assurance 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 germe aléatoire. Pour plus de détails, voir la documentation de *QDate*, *QTime* et *QDateTime*.

Enfin, nous calculons le point central de la cible. Nous le conservons dans le rectangle (*x* : 200, *y* : 35, *width* : 190, *height* : 255, les plages de valeur possibles pour *x* et *y* sont respectivement de 200 à 389 et de 35 à 289), dans un système de coordonnées où *y* prend la valeur 0 sur le bord inférieur du widget et augmente vers le haut, tandis que *x* prend comme d'habitude la valeur 0 sur le bord gauche du widget et augmente vers la droite.

Par expérience, nous avons constaté que cela permet toujours d'avoir une cible à portée de tir.

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

    QRect shotR = shotRect();
```

Cette partie de l'événement du *timer* n'a pas changé par rapport au chapitre précédent.

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

Ce test conditionnel 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 alors le *timer* du tir, nous émettons le signal *hit()* pour indiquer au monde extérieur que la cible a été détruite et nous sortons finalement de la fonction.

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 assure que la cible est également 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. A propos de *newTarget()*, souvenez-vous que la valeur 0 de l'ordonnée y du centre de la cible correspond au bord inférieur du widget. Nous calculons le point en coordonnées du widget avant d'appeler *QRect::moveCenter()*.

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

XIV-A-5 - 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.

XIV-B - Exécuter l'application

Le widget LCDRange paraît un peu bizarre : quand nous redimensionnons 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.

XIV-C - Exercices

Fabriquez un "bouton pour tricheur" qui, lorsqu'il est cliqué, fait en sorte que *CannonField* affiche 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()* permettant une détection de collision très précise.

Faites une cible mobile.

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

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

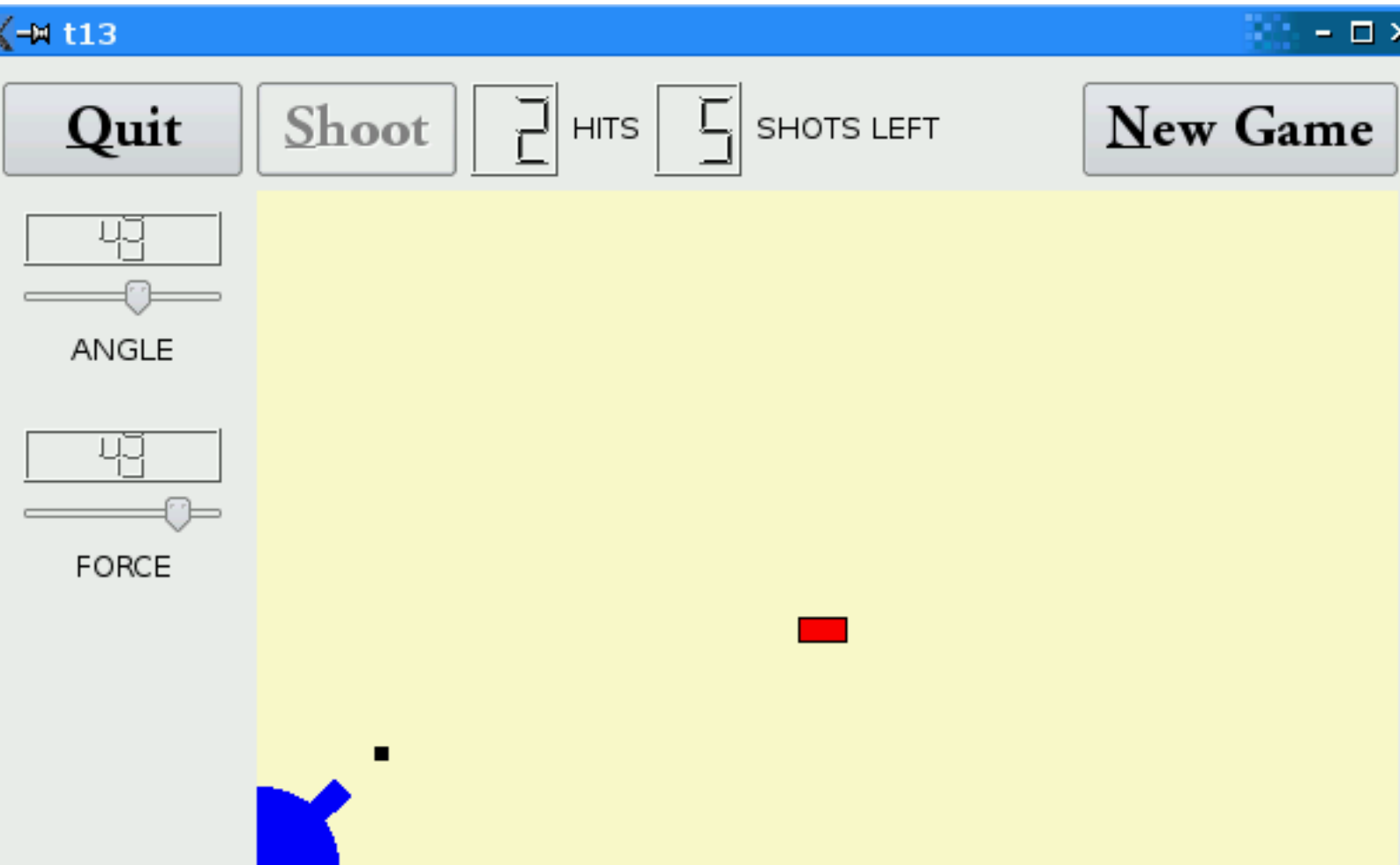
Exercice pas facile : rendez possible des tirs simultanés [indice : faites une classe Tir].

XV - 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



Avec cet exemple, nous commençons à nous approcher d'un jeu vraiment jouable. Nous donnons aussi à MyWidget un vrai nom (GameBoard) et nous ajoutons 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.

XV-A - Analyse du code ligne par ligne

XV-A-1 - 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 résout le problème d'affichage rencontré au cours du chapitre XIV.

XV-A-2 - t13/cannonfield.h

Le *CannonField* possède, outre un état *gameOver*, 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 état dans lequel le slot *shoot()* a du sens. Nous allons l'utiliser bientôt pour activer ou non le bouton *Tirer*.

```
bool gameEnded;
```

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

XV-A-3 - 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. Ainsi, *shoot()* nous indique si le *CannonField* 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 laquelle le joueur gagnant est celui qui le premier touche dix cibles, peut utiliser le *CannonField* inchangé).

Si le jeu est déjà terminé, nous retournons immédiatement. Si le jeu continue, nous arrêtons le tir, utilisons le *flag game over* et redessinons 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 projectile est en l'air, nous l'arrêtons. Ensuite, nous remettons la variable `gameEnded` à `false` et repeignons le widget.

`restartGame()` émet en outre 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), le chargement d'une police aussi grande peut prendre beaucoup de temps. Mais comme Qt met en cache les polices, vous ne remarquerez ce problème que la première fois que la police est utilisée.

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

Nous ne dessinons le projectile que quand on tire ; la cible, seulement quand on joue.

XV-A-4 - t13/gameboard.h

Ce fichier est nouveau : il contient la définition de la classe `GameBoard`, qui correspond au `MyWidget` des paragraphes précédents.

```
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.

XV-A-5 - 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 membre, 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 se produise quand le tir atteint ou manque 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, voulant garder la trace du nombre de tirs, nous le connectons 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 non le bouton 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 bouton New Game, comme nous l'avons fait pour les autres boutons. Un simple clic sur ce bouton 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 que les classes qui gèrent l'affichage les redimensionneront 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 encombrée. Nous ajoutons un espace immédiatement à gauche du bouton *New Game* pour être sûr que ce bouton 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 déclenche un tir. Si le jeu est fini ou s'il y a déjà un projectile en l'air, nous nous arrêtons immédiatement. Nous décrémentons le nombre de tirs restants et demandons au canon 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 tirs réussis. 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 manqué 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 bouton *New Game*. Il est aussi appelé par le constructeur. Initialement, il autorise 15 tirs. Notez que c'est là le seul emplacement du programme où nous indiquons le nombre de tirs. Changez-le si vous le souhaitez. Ensuite, nous remettons à 0 le nombre de tirs réussis, relançons le jeu et générons une nouvelle cible.

XV-A-6 - t13/main.cpp

Ce fichier a subi un "régime amaigrissant" ! MyWidget est parti, il ne reste plus que la fonction *main()*, inchangée, si ce n'est pour le nom.

XV-B - Exécuter l'application

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

Tirs restants et tirs réussis sont affichés et le programme en garde une trace. Le jeu prend fin, un bouton apparaît, permettant de commencer une nouvelle partie.

XV-C - Exercices

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

Afficher quelques effets quand la cible est touchée.

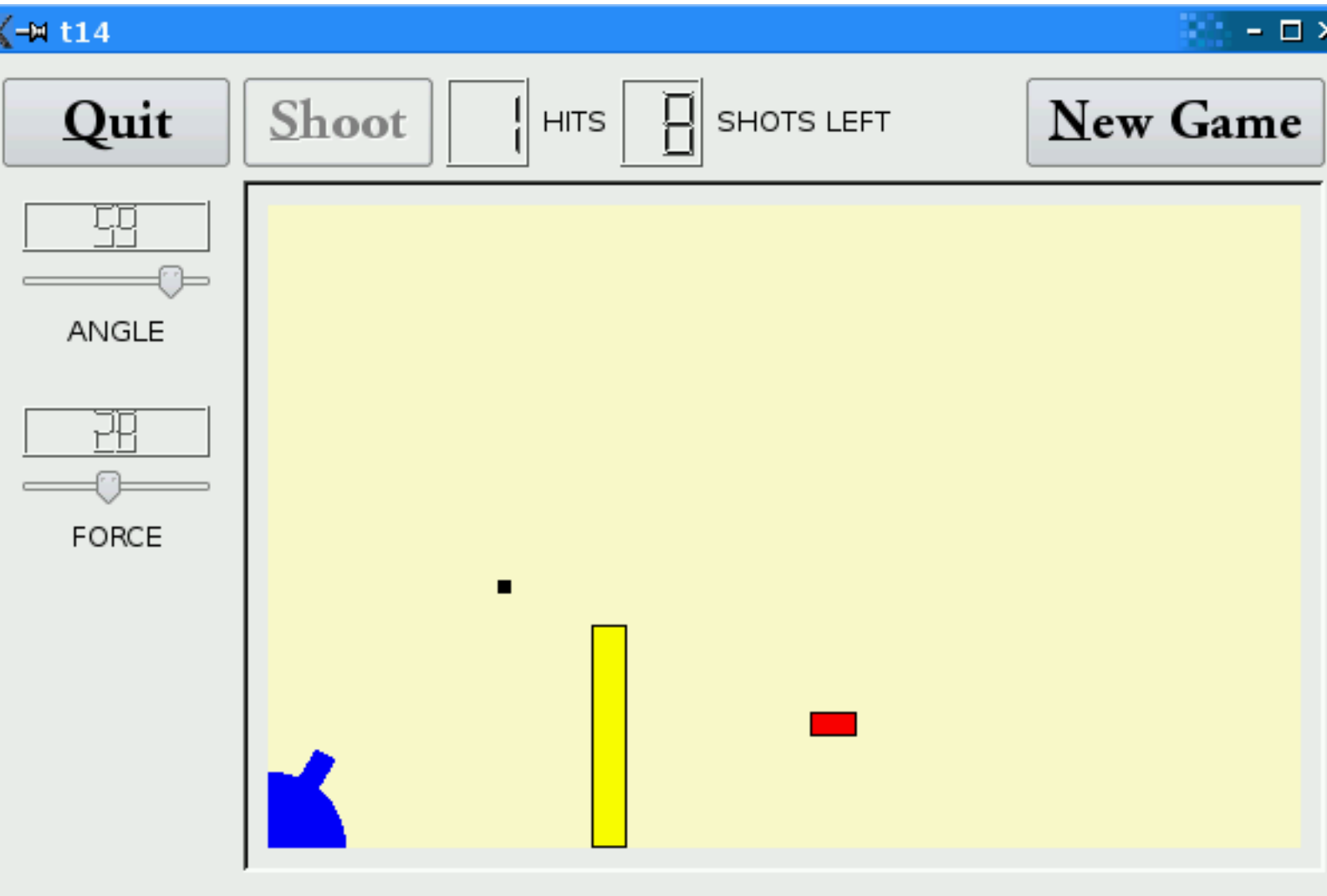
Implémenter des cibles multiples.

XVI - 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 : le jeu est complet ! Nous ajoutons à CannonField des accélérateurs de clavier ainsi que des événements de la souris. Nous dessinons un cadre autour de CannonField et plaçons un obstacle (un mur) pour rendre le jeu plus difficile.

XVI-A - Analyse du code ligne par ligne

XVI-A-1 - t14/cannonfield.h

Le CannonField peut maintenant recevoir les événements de la souris permettant de pointer l'extrémité du canon sur lui et en le déplaçant avec la souris. Le CannonField comporte 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 pixel est à l'intérieur du fût du canon.

```
bool barrelPressed;
```

Cette variable privée est vraie si l'utilisateur a cliqué avec la souris sur le fût du canon.

XVI-A-2 - 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 une troisième façon de manquer le tir. Nous allons la tester.

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

C'est un gestionnaire 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 si le curseur de la souris est bien situé à l'intérieur du fût 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 gestionnaire d'événement Qt. Il est appelé quand l'utilisateur maintient appuyé le bouton de la souris alors que son curseur est situé à l'intérieur du widget et qu'il déplace la souris (vous pouvez utiliser Qt pour envoyer des événements de mouvement de souris même quand aucun bouton n'est pressé ; regardez QWidget::setMouseTracking()).

Ce gestionnaire replace le fût du canon selon la position du curseur.

Tout d'abord, si l'on ne clique pas sur le fût du canon, nous retournons. Ensuite, nous examinons la position du curseur : si le curseur est à gauche ou en-dessous du widget, nous ajustons sa position pour qu'il soit dans le widget.

Puis nous calculons l'angle formé par le bord inférieur du widget et la ligne imaginaire reliant l'angle supérieur gauche du widget et la position du curseur. Enfin, nous réglons l'angle de tir à la nouvelle valeur exprimée en degrés.

Souvenez-vous que `setAngle()` redessine le canon.

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

Le gestionnaire d'événement Qt est appelé quand l'utilisateur relâche le bouton de la souris après avoir cliqué à l'intérieur du widget. Si le bouton gauche est relâché, nous pouvons être sûrs que le fût du canon n'est plus concerné par le clic. 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 plaçons le bord inférieur 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 à l'intérieur du fût du canon ; sinon elle retourne false. Nous utilisons ici la classe `QMatrix`. Cette classe définit un système de coordonnées. Elle permet les mêmes transformations que `QPainter`. Nous appliquons les mêmes transformations que pour dessiner l'extrémité dans la fonction `paintCannon()`. Tout d'abord nous appliquons au système de coordonnées une translation suivie d'une rotation. Nous devons alors vérifier si le point `pos` (dans les coordonnées du widget) se situe dans le fût du canon.. Pour cela, nous inversons la matrice de transformation. Cette matrice inversée correspond aux opérations inverses de celles que nous avons utilisées pour dessiner le fût du canon. Nous localisons le point `pos` en utilisant la matrice inversée et retournons true s'il est dans le rectangle d'origine de l'embout.

XVI-A-3 - 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 envoyés à un widget et appellent des slots si certaines touches sont pressées. Notez qu'un objet QShortcut est l'enfant d'un widget et sera donc détruit quand ce widget sera détruit. QShortcut lui-même n'est pas un widget et n'a pas d'effet visible sur son parent.

Nous définissons trois raccourcis-clavier. 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 enfoncées en même temps. Au lieu de nous connecter à QApplication::quit(), nous nous connectons cette fois à QWidget::Close, ce qui a le même effet que quit() puisque GameBoard est maintenant le widget principal de l'application.

Qt::CTRL, Qt::Key_Enter, Qt::Key_Return et Qt::Key_Q sont toutes des constantes déclarées dans l'espace de nommage 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 cannonField à cette mise en page. cannonField devient implicitement l'enfant de cannonBox. Puisqu'il n'y a rien d'autre dans la boîte, QVBoxLayout placera donc un cadre autour de CannonField. Nous mettons cannonBox, non CannonField, dans la mise en page de la grille.

XVI-B - Exécuter l'application

Maintenant le canon tire quand nous appuyons sur Entrée. Nous pouvons aussi régler l'angle de tir avec la souris. L'obstacle rend le jeu un peu plus difficile. Nous avons maintenant un joli cadre autour de CannonField.

XVI-C - Exercices

Écrire un jeu de "space invaders". Cet exercice a initialement été proposé par Igor Rafienko. Vous pouvez [télécharger son jeu](#). Nouvel exercice : écrire un jeu de casse briques. Objectif final : aller plus loin et créer des chefs-d'oeuvre de programmation.