Construction d'Interfaces Graphiques

Alexis NEDELEC

Centre Européen de Réalité Virtuelle
Ecole Nationale d'Ingénieurs de Brest

enib ©2012

Introduction

Présentation

- toolkit Graphique C++
- à la base de l'environnement KDE
- développé par TrollTech, puis Nokia
  - `http://qt.nokia.com`
- licences LGPL et commerciale
- multiplateformes : OS classiques et mobiles
- devise: "write once, compile anywhere"

Introduction

Toolkit graphique ...

- framework pour applications graphiques 2D/3D
- programmation objet, événementielle
- mécanismes de signaux et de slots (`moc`)
- binding `C`, `python`, `C#` `...`
- outils de développement (`Qt Creator`, `Qt Designer`)

... mais pas seulement

- Open GL multiplateforme
- internationalisation (`Unicode`, `QString`)
- gestion de fichiers, connexion SGBD
- communication inter-processus, réseau
- W3C : XML, SAX, DOM

Introduction

Arbre d'Héritage

```
QObject
   |-- QTimer
   |-- QWidget
   |      |--QDialog
  ...     |--QFrame
          |    |--QLabel
          |    |--QSpinbox
         ...   |
              ...
          |--Q...
          |
         ...
```

Introduction

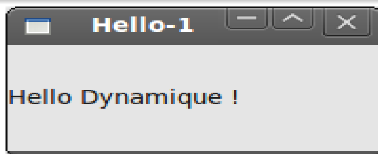
Principaux modules Qt

- `QtCore` : classes de base pour tous les modules
- `QtGUI` : composants graphiques 2D
- `QtOpenGL` : pour faire de la 3D
- `QtSql` : connexion, manipulation SGBD
- `QtSvg` : gestion de contenu SVG (Scalable Vector Graphics)
- `QtWebKit` : gestion de contenu WEB
- `QtXML,QtXMLPatterns` : W3C (XML,XQuery,XPath ...)
- `Phonon` : pour applications multimédia

Introduction

Hello World

```
#include <QApplication>
#include <QLabel>
int main(int argc, char **argv)
{
 QApplication* app = new QApplication(argc, argv);
 QLabel* hello = new QLabel("Hello Dynamique!");
 hello->show( );
 return app->exec( );
}
```



Introduction

Hello World : les inclusions

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

Hello World : un conteneur

```
int main(int argc, char **argv)
{
 QApplication app(argc, argv);
 QWidget box;
 box.resize(200, 120);
```

Introduction

Hello World : les composants graphiques

```
 QLabel hello("Hello Statique !", &box);
 hello.resize(300, 50);
 hello.move(10, 30);
 hello.setFont( QFont("Times", 18, QFont::Bold) );
 box.show();
 return app.exec();
}
```

Introduction

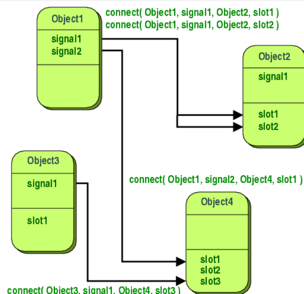
Hello World : l'interaction

```
int main(int argc, char **argv)
{
 ...
 QPushButton quitBtn("Quit", &box);
 ...
 QObject::connect(&quitBtn, SIGNAL(clicked( )),
                  &app, SLOT(quit( )) );
 box.show();
 return app.exec();
}
```

Signaux et Slots

Principe

- changement d'état d'un objet : émission de signal
- réception de signal par un objet : déclenchement d'un slot
- un slot est un comportement (une méthode) à activer
- programmation par composants, modèle multi-agents



Signaux et Slots

Caractéristisques

- modulaire, flexible
  - un signal, plusieurs slots et réciproquement
  - l'émetteur n'a pas à connaître le récepteur et réciproquement
  - l'émetteur ne sait pas si le signal est reçu (broadcast)
- transmission de données
  - typage fort : les types de données doivent être les mêmes
  - un slot peut avoir moins de paramètres
- remarques
  - différent des mécanismes de *callbacks, listeners*
  - aspect central de la programmation Qt
  - `SLOT, SIGNAL` sont des macros : précompilation (`moc`)

Signaux et Slots

Déclaration : Q_OBJECT,slots,signals

```
class SigSlot : public QObject {
Q_OBJECT
 public:
  SigSlot(): _val(0) {}
  int  value() const {return _val;}
 public slots:
  void setValue(int);
 signals:
  void valueChanged(int);
 private:
  int  _val;
};
```

Signaux et Slots

Définition : méthodes correspondant aux slots uniquement

```
#include "sigslot.h"

void SigSlot::setValue(int v) {
 if (v != _val) {
   _val = v;
   emit valueChanged(v);
 }
}
```

Emission de signal : `emit`

- `valueChanged(v)` : avec la nouvelle valeur `v`
- `v != _val` : si cette dernière a changé

Signaux et Slots

Connexion : QObject::connect()

```
#include <QDebug>
#include "sigslot.h"

int main(int argc, char* argv[]) {
 SigSlot a, b;
 QObject::connect(&a, SIGNAL(valueChanged(int)),
                  &b, SLOT(setValue(int)));
 b.setValue( 10 );
 qDebug() << a.value(); // 0 or 10 ?
 a.setValue( 100 );
 qDebug() << b.value(); // 10 or 100 ?
}
```

Signaux et Slots

Connexion composants graphiques

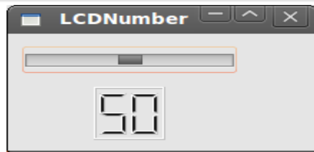
```
#include <QApplication>
#include <QSlider>
#include <QLCDNumber>

int main(int argc, char **argv) {
 QApplication app(argc, argv);
 QWidget mw;
 mw.setGeometry(10, 10, 200, 100);
 QSlider numSlider(Qt::Horizontal, &mw);
 numSlider.setGeometry(10, 10, 150, 30);
 numSlider.setValue(50);
 numSlider.setMinimum(0);
 numSlider.setMaximum(100);
```

Signaux et Slots

Connexion composants graphiques

```
QLCDNumber numLCD(2, &mw);
numLCD.setGeometry(60, 50, 50, 50);
numLCD.display(50);
QObject::connect(&numSlider, SIGNAL(sliderMoved(int)),
                 &numLCD, SLOT(display(int)) );
mw.show();
return app.exec();
```



Signaux et Slots

Méta-Object Compiler : `moc`

- pré-processeur C++
- génération de code (tables de signaux/slots)
- déclaration IMPERATIVE du mot-clé `Q_OBJECT`

Compilation : `qmake`

- `qmake -project` : générer le fichier `nomDeProjet.pro`
- `qmake nomDeProjet.pro` : générer le fichier `Makefile`
- `make` : générer les fichiers `moc_*.cpp, *.o` et l'éxécutable

Des signaux et des slots

Transmettre des données entre signaux et slots

- illustration du problème : **Pavé numérique**
- l'utilisateur clique (`clicked()`) sur un chiffre (`int`)
- les touches émettent leur valeur (`digitClicked(int)`)
- un objet peut déclencher une action (`displayValue(int)`)

Des signaux et des slots

Pavé numérique : plusieurs solutions

1. un seul signal `clicked()` pour n slots
2. récupération de l'émetteur (`sender()`) dans un slot
3. définition d'un nouveau signal par héritage de `QPushButton`
4. multiplexage de signaux : `QSignalMapper`

Pavé numérique : première solution

- on crée un signal avec un argument `digitClicked(int)`
- on implémente un slot par touche (N) `buttonNClicked()`
- chaque slot émet le signal avec la valeur correspondante

Des signaux et des slots

Pavé numérique : première solution

```
class Keypad : public QWidget {
 Q_OBJECT
  public:
    Keypad(QWidget *parent = 0);
  signals:
    void digitClicked(int digit);
  private slots:
    void button0Clicked();
    ...
    void button9Clicked();
private:
    QGridLayout* _createLayout();
    QPushButton *_buttons[10];
};
```

Des signaux et des slots

Pavé numérique : première solution

```
Keypad::Keypad(QWidget *parent) : QWidget(parent) {
 for (int i = 0; i < 10; ++i) {
  QString text = QString::number(i);
  _buttons[i] = new QPushButton(text, this);
 }
 connect(_buttons[0], SIGNAL(clicked()),
         this, SLOT(button0Clicked()));
 ...
 connect(_buttons[9], SIGNAL(clicked()),
         this, SLOT(button9Clicked()));
 createLayout();
}
```

Des signaux et des slots

Pavé numérique : première solution

```
void Keypad::button0Clicked()
{
    emit digitClicked(0);
}
...
void Keypad::button9Clicked()
{
    emit digitClicked(9);
}
```

Première solution : bilan

- uniquement pour les adeptes du copier-coller

Des signaux et des slots

Pavé numérique : utilisation

```
int main(int argc, char* argv[]) {
   QApplication app(argc, argv);
   QWidget box;
   Keypad pad(&box);
   MyObject obj;
   QObject::connect(&pad, SIGNAL(digitClicked(int)),
                    &obj, SLOT(displayValue(int)) );
   box.show();
   return app.exec();
}
```

Des signaux et des slots

Pavé numérique : utilisation

```
#include <QObject>
#include <QDebug>
class MyObject : public QObject {
Q_OBJECT
public:
    MyObject(void) {}
public slots:
    void displayValue(int nb) { qDebug() << nb;}
};
```

Des signaux et des slots

Pavé numérique : deuxième solution

```
class Keypad : public QWidget {
 Q_OBJECT
 public:
  Keypad(QStringList, QWidget *parent = 0);
 signals:
  void digitClicked(int digit);
 private slots:
  void buttonClicked();
 private:
  QGridLayout * _createLayout(int n);
  QPushButton* _buttons[10];
};
```

Des signaux et des slots

Pavé numérique : deuxième solution

```
Keypad::Keypad(QStringList texts, QWidget *parent)
   : QWidget(parent) {
 for (int i = 0; i < texts.size(); ++i) {
  _buttons[i] = new QPushButton(texts[i], this);
   connect(_buttons[i],SIGNAL(clicked()),
           this,SLOT(buttonClicked()));
 }
 _createLayout(texts.size()-1);
}
void Keypad::buttonClicked(void) {
  QPushButton *button = (QPushButton *)sender();
  emit digitClicked(button->text()[0].digitValue());
}
```

Des signaux et des slots

Deuxième solution : bilan

Uniquement pour les amateurs de boutons-poussoirs :

- on utilise un slot privé pour faire le démultiplexage
- on présuppose que l'émetteur est un `QPushButton`
- que se passe t'il si on change le texte des touches ?

Deuxième solution : petite amélioration

```
void Keypad::buttonClicked(void) {
 QObject* emetteur=sender();
 QPushButton* buttonCasted=
                qobject_cast<QPushButton*>(emetteur);
 if(buttonCasted) {
  emit digitClicked(buttonCasted->text()[0].digitValue())
 }
}
```

Des signaux et des slots

Pavé numérique : troisième solution

```
class KeypadButton : public QPushButton {
 Q_OBJECT
 public:
  KeypadButton(int digit, QWidget *parent);
 signals:
  void clicked(int digit);
 private slots:
  void reemitClicked();
 private:
    int _digit;
};
```

Des signaux et des slots

Pavé numérique : troisième solution

```
KeypadButton::KeypadButton(QString digit,
                           QWidget *parent)
    : QPushButton(parent)
{
 setText(digit);
 _digit = digit.toInt();
 connect(this,SIGNAL(clicked()),
         this,SLOT(reemitClicked()));
}
void KeypadButton::reemitClicked()
{
 emit clicked(_digit);
}
```

Des signaux et des slots

Pavé numérique : troisième solution

```
Keypad::Keypad(QStringList texts, QWidget *parent)
    : QWidget(parent)
{
 for (int i = 0; i < texts.size(); ++i) {
  _buttons[i] = new KeypadButton(texts[i], this);
   connect(_buttons[i], SIGNAL(clicked(int)),
           this, SIGNAL(digitClicked(int)));
 }
 _createLayout(texts.size()-1);
}
```

Troisème solution : bilan

- uniquement pour les adeptes de l'héritage forcé

Des signaux et des slots

Pavé numérique : quatrième solution

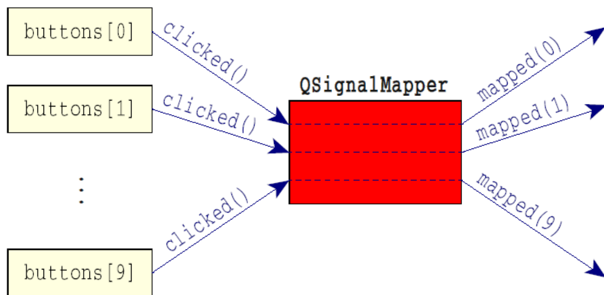
```
class Keypad : public QWidget {
 Q_OBJECT
 public:
  Keypad(QStringList texts, QWidget *parent = 0);
 signals:
  void digitClicked(int digit);
 private:
  QGridLayout * _createLayout(int n);
  QPushButton* _buttons[10];
  QSignalMapper *_signalMapper;
};
```

Des signaux et des slots

Pavé numérique : quatrième solution

```
Keypad::Keypad(QStringList texts, QWidget *parent)
   : QWidget(parent) {
 _signalMapper = new QSignalMapper(this);
 connect(_signalMapper,SIGNAL(mapped(int)),
         this, SIGNAL(digitClicked(int)));
 for (int i = 0; i < texts.size(); ++i) {
  _buttons[i] = new QPushButton(texts[i], this);
  _signalMapper->setMapping(_buttons[i], i);
  connect(_buttons[i],SIGNAL(clicked()),
          _signalMapper, SLOT(map()));
 }
 _createLayout(texts.size()-1);
}
```

Des signaux et des slots



Pavé numérique : quatrième solution

- il suffit de mapper un signal existant avec un nouveau signal

Limitations

- `mapped(int i)`, `mapped(const QString & text)`
- `mapped(QWidget * widget)`, `mapped(QObject * object)`

Des signaux et des slots

Pavé numérique : gestionnaire de positionnement

```
QGridLayout*  Keypad::_createLayout(int n)
{
 QGridLayout *layout = new QGridLayout(this);
 layout->setMargin(6);
 layout->setSpacing(6);
 for (int i = 0; i < n; ++i) {
  layout->addWidget(_buttons[i+1], i / 3, i % 3);
 }
 layout->addWidget(_buttons[0], 3, 1);

 return layout;
}
```

Des signaux et des slots

Pavé numérique : programme de test

```
int main(int argc, char* argv[]) {
 QApplication app(argc, argv);
 QWidget box;
 QStringList numbers;
 for (int i=0;i< 10;i++) numbers << QString::number(i);
 Keypad kpad(numbers, &box);
 MyObject obj;
 QObject::connect(&kpad, SIGNAL(digitClicked(int)),
                 &obj, SLOT(displayValue(int)) );
 box.show();
 return app.exec();
}
```

Gestion des évènements

Héritage `QWidget` : Surdéfinition des méthodes

- `void mousePressEvent(QMouseEvent* evt);`
- `void mouseMoveEvent(QMouseEvent* evt);`
  - `void setMouseTracking(bool) ;`
  - `bool hasMouseTracking() ;`
- `void mouseReleaseEvent(QMouseEvent* evt);`
- `void mouseDoubleClickEvent(QMouseEvent* evt);`

Gestion des évènements

myWidget.h : mousePressEvent()

```
#include <QWidget>
#include <QMouseEvent>
#include <QDebug>

class MyWidget : public QWidget {
 public:
  MyWidget();
 protected:
  void mousePressEvent(QMouseEvent*);
};
```

Gestion des évènements

myWidget.cpp : mousePressEvent()

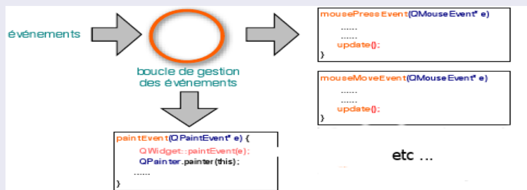
```
void MyWidget::mousePressEvent(QMouseEvent* evt) {
 if (evt->button() == Qt::LeftButton) {
   // TO DO
   update();
 }
}
```

myWidget.cpp : paintEvent()

```
void MyWidget::paintEvent(QPaintEvent* evt) {
 qDebug() << "MyWidget::paintEvent()";
 QWidget::paintEvent(evt);
 QPainter painter(this);
 painter.drawLine(50, 10, 100, 20);
}
```

Gestion des évènements

`MyWidget : paintEvent(), ...Event(), update()`



`paintEvent()` : demande de réaffichage

- à chaque retour dans la boucle de gestion des évènements
  - lorsqu'une fenêtre passe sur le composant
  - lorsque l'on déplace le composant
- lors d'une demande explicite
  - `update()` : demande de rafraîchissement (en file d'attente)
  - `repaint()` : rafraichissement immédiat (à éviter)

Dessiner dans un composant

Classes de base

- `Qpainter` : outil de dessin
- `QpaintDevice` : objet sur lequel on peut dessiner
- `QpaintEngine` : moteur de rendu

Héritage `QpaintDevice`

- `QWidget`
- `QPixmap, QImage, QPicture`
- `QPrinter`
- `QGLPixelBuffer, QGLFrameBufferObject`
- ...

Dessiner dans un composant

MyPainter : Ma classe de dessin

```
class MyPainter : public QWidget {
public:
 MyPainter(void);
 virtual ~MyPainter(void);
protected:
 void mousePressEvent(QMouseEvent*);
 void mouseMoveEvent(QMouseEvent*);
 void mouseReleaseEvent(QMouseEvent*);
 void paintEvent(QPaintEvent*);
private :
 QPoint _startPoint, _endPoint;
 QRubberBand *_rubberBand;
};
```

Dessiner dans un composant

MyPainter : Je me prépare pour dessiner

```
MyPainter::MyPainter(void) {
 _buffer = new QPixmap(this->size());
 _buffer->fill(Qt::white);
 _rubberBand=new QRubberBand(QRubberBand::Rectangle,
                              this);
}
void MyPainter::mousePressEvent(QMouseEvent* evt)
{
 if (evt->button() == Qt::LeftButton) {
  _startPoint = _endPoint = evt->pos();
  _rubberBand->setGeometry(QRect(_startPoint,QSize()));
  _rubberBand->show();
 }
}
```

Dessiner dans un composant

MyPainter : Je dessine en mode élastique

```
void MyPainter::mouseMoveEvent(QMouseEvent* evt) {
 _rubberBand->setGeometry(
            QRect(_startPoint,evt->pos()).normalized()
             );
}
void MyPainter::mouseReleaseEvent(QMouseEvent* evt) {
 if (evt->button() == Qt::LeftButton) {
   _rubberBand->hide();
   _endPoint = event->pos();
   update();
 }
}
```

Dessiner dans un composant

MyPainter : J'affiche mes beaux dessins

```
void MyPainter::paintEvent(QPaintEvent* evt) {
 QWidget::paintEvent(evt);
 QPainter paintWindow(this);
 QPainter paintBuffer(_buffer);
 paintWindow.drawPixmap(0,0, *_buffer);
 paintWindow.drawLine(_startPoint,_endPoint);
 paintBuffer.drawLine(_startPoint,_endPoint);
 paintBuffer.end();
 paintWindow.end();
}
```

classe QPainter

Dessin en 2D : mode matriciel (bitmap)

- attributs de dessin (stylo, pinceau, texte ...)
- lignes et contours (ligne, polygone, chemins ...)
- remplissage (couleur, gradient, texture ...)
- découpage (régions, intersections ...)
- transformations (affines ...)
- anti-crénelage (dépend du moteur de rendu)
- composition image/fond (Porter Duff)
- ...

classe `QPainter`

Classes utiles au dessin

- `QPen`,`QBrush`,`QColor`,`QFont` : attributs de dessin
- `QPoint`, `QLine`, `QRect`, `QPolygon` (entiers)
- `QPointF`, `QLineF` ... (flottants)
- `QPainterPath` : chemins complexes
- `QRegion` : découpage (régions, intersections ...)
- `QTransform` : transformations 2D (matrice 3x3)
- `QImage` : manipulation d'images
- `QPixmap`, `QBitmap` : affichage à l'écran
- `QPicture` : stockage d'images
- ...

classe QPainter

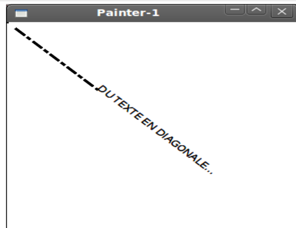
Attributs de dessin

```
QPen pen;
pen.setStyle(Qt::DashDotLine);
pen.setWidth(3);
pen.setBrush(Qt::green);
paintBuffer.setPen(pen);
QBrush brush(Qt::DiagCrossPattern);
paintWindow.setBrush(brush);
paintBuffer.setBrush(brush);
```

classe QPainter

Transformations affines

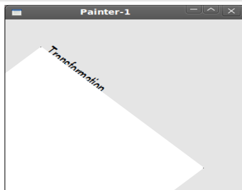
```
paintBuffer.save();
paintBuffer.translate(100,100);
paintBuffer.rotate (45);
paintBuffer.drawText(0, 0, "DU TEXTE EN DIAGONALE...");
paintBuffer.restore();
paintBuffer.drawLine(10,10,100,100);
```



classe QPainter

Transformations affines : classe QTransform

```
QTransform transform;
transform.translate(50, 50);
transform.rotate(45);
transform.scale(0.5, 1.0);
paintWindow.setTransform(transform);
paintWindow.setFont(QFont("Helvetica", 24));
paintWindow.setPen(QPen(Qt::black, 1));
paintWindow.drawText(20, 10, "Transformation");
```



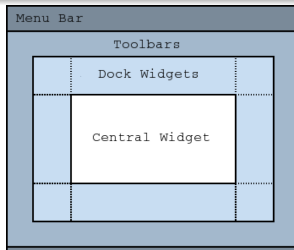
QMainWindow : Fenêtre principale

Répartition en zones de travail

- barres de menu, d'outils, de statut
- zone centrale (cliente)
- autres fonctionnalités

Utilisation

- création d'une sous-classe de `QMainWindow`
- création des zones de travail dans le constructeur



QMainWindow : Fenêtre principale

QMainWindow : Héritage, constructeur

```
#include <QApplication>
#include <QtGui>
class MainWindow : public QMainWindow {
Q_OBJECT
 public:
  MainWindow();
 private :
  void _createMenus(void);
  void _createToolbars(void);
  void _createActions(void);
  void _connectActions(void);
  void _connectSignals(void);
 ...
```

QMainWindow : Fenêtre principale

QMainWindow : Héritage, constructeur

```
#include <QtDebug>
#include "mainWindow.h"

MainWindow::MainWindow(void) {
 _createActions();
 _createMenus();
 _createToolbars();
 _connectActions();
 _connectSignals();
 ...
}
```

QMainWindow : Actions

QAction : sur un ou plusieurs composants

```
void MainWindow::_createActions(void)
{
 _newAction = new QAction(QIcon(":/images/new.png"),
                          tr("&New..."), this);
 _newAction->setShortcut(tr("Ctrl+N"));
 _newAction->setToolTip(tr("New file"));
 _newAction->setStatusTip(tr("New file"));
 _newAction->setData(QVariant("_newAction data"));
 ...
```

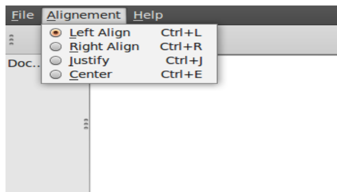
QMainWindow : Actions

QActionGroup : sur un groupe de composants

```
 _leftAlignAction=new QAction(tr("&Left Align"),this);
 _leftAlignAction->setCheckable(true);
 ...
 _centerAction = new QAction(tr("&Center"),this);
 _centerAction->setCheckable(true);

 _alignGA = new QActionGroup(this);
 _alignGA->addAction(_leftAlignAction);
 _alignGA->addAction(_rightAlignAction);
 _alignGA->addAction(_justifyAction);
 _alignGA->addAction(_centerAction);
 _leftAlignAction->setChecked(true);
...
}
```

QMainWindow : Zone client et palette d'outils



QMainWindow::setCentralWidget(),QDockWidget

```
// in MainWindow::MainWindow()
QTextEdit* text = new QTextEdit(this);
setCentralWidget(text);
QDockWidget *dock = new QDockWidget(tr("Dock"),this);
dock->setAllowedAreas(Qt::LeftDockWidgetArea |
                      Qt::RightDockWidgetArea);
addDockWidget(Qt::LeftDockWidgetArea, dock);
```

QMainWindow : Menus

QMenuBar : barre d'Actions

```
void MainWindow::_createMenus(void) {
 QMenuBar* menubar = menuBar( );  //QMainWindow method
 _fileMenu = menubar->addMenu( tr("&File") );
 _alignMenu = menubar->addMenu( tr("&Alignement") );
 _helpMenu = menubar->addMenu( tr("&Help") );
}
```

QToolBar : barre d'outils

```
void MainWindow::_createToolbars(void) {
 _toolBar=addToolBar( tr("File") ); //QMainWindow method
}
```

QMainWindow : Menus avec actions

QMenuBar : barre d'Actions

```
void MainWindow::_connectActions(void) {
 _fileMenu->addAction( _newAction );
 _alignMenu->addAction(_leftAlignAction);
 _alignMenu->addAction(_rightAlignAction);
 _alignMenu->addAction(_justifyAction);
 _alignMenu->addAction(_centerAction);
 _toolBar->addAction(_newAction);
 _helpMenu->addAction(_aboutAction);
 _helpMenu->addAction(_aboutQtAction);
}
```

QMainWindow : Actions

QObject : connexion des actions

```
void MainWindow::_connectSignals(void) {
 connect( _newAction, SIGNAL(triggered( )),
          this,SLOT(_newFile( )) );
 connect(_leftAlignAction, SIGNAL(triggered()),
         this,SLOT(_leftAlign()));
 ...
 connect(_aboutAction, SIGNAL(triggered()),
         this, SLOT(_about()));
 connect(_aboutQtAction,SIGNAL(triggered()),
         this, SLOT(_aboutQt()));
 ...
}
```

QMainWindow : Actions

MainWindow : implémentation des comportements

```
void MainWindow::_newFile(void) {
 qDebug() << "Date: " << QDate::currentDate();
 QString str = _newAction->data().toString();
 qDebug() << str ;
}
void MainWindow::_about(void) {
 qDebug() << "Date: " << QDate::currentDate();
 QMessageBox::information( this,
                           "About Us",
                           "Dupond - Dupont",
                           "Back to work !");
}
```

QMainWindow : Barre d'Outils

MainWindow : MainWindow.qrc

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file>images/new.png</file>
    <file>images/open.png</file>
    ...
</qresource>
</RCC>
```

MainWindow : mainWindow.cpp

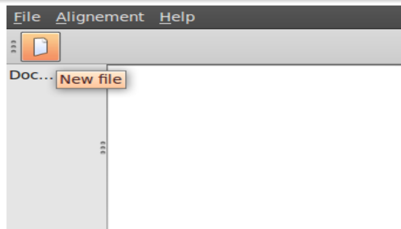
```
void MainWindow::_createActions(void) {
 _newAction = new QAction(QIcon(":/images/new.png"),
                          tr("&New..."), this );
 ...
}
```

QMainWindow : Barre d'Outils

MainWindow : retrouver les icônes

Regénérer le fichier MainWindow.pro (qmake -project)

```
...
# Input
HEADERS += mainWindow.h
SOURCES += main.cpp mainWindow.cpp
RESOURCES += MainWindow.qrc
```



Architecture MVC

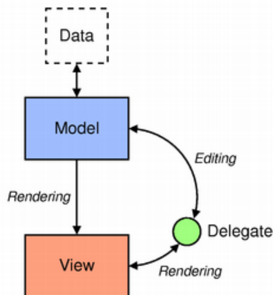
Modèle-Vue-Contrôleur : Patron de conception (smalltalk)

"MVC consists of three kinds of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input. Before MVC, user interface designs tended to lump these objects together. MVC decouples them to increase flexibility and reuse."

Modèle-Vue-Contrôleur

- Modèle : données relatives à l'application (logique métier)
- Vue : diverses présentations du modèle (IHM)
- Contrôleur : modifier le modèle et la présentation en réponse aux actions de l'utilisateur

Qt : architecture Modèle-Vues et délégués



Modèle-Vue-Délégué

Classes abstraites pour créer ses propres modèles

- `QAbstractItemModel` : interface du modèle avec les données
- `QAbstractItemView` : référence sur les données sources
- `QAbstractItemDelegate` : communication entre la vue et le modèle

Qt : architecture Modèle-Vues et délégués

Modèles prédéfinis

- `QStringListModel` : listes de chaînes de caractères
- `QStandardItemModel` : organisation arborescente
- `QFileSystemModel` : gestion des systèmes de fichiers
- `QSqlQueryModel, QSqlTableModel et QSqlRelationalTableModel` : gestion des bases de données)

Architecture d'accès aux Bases de Données

- `QSqlTableModel` : le modèle de données
- `QTableView` : vue correspondante
- `QSqlRelationalDelegate` : un délégué (controleur)

Modèle-Vue architecture

Architecture d'accès aux Bases de Données

```
int main(int argc, char *argv[]) {
 QApplication app(argc, argv);
 QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
 db.setDatabaseName("toto");
 if (!db.open()) { // alert the user }
 createRelationalTables();
 QSqlRelationalTableModel model;
 initializeModel(&model);
 QTableView *view =
      createView(QObject::tr("Table Model"),&model);
 view->show();
 return app.exec();
}
```

Modèle-Vue architecture

Architecture d'accès aux Bases de Données

```
void createRelationalTables() {
 QSqlQuery query;
 query.exec("create table employee(id int primary key,
            name varchar(20), city int, country int)");
 query.exec("insert into employee values(1, 'Toto',
                                        5000, 47)");
 ...
 query.exec("create table city(id int,
                              name varchar(20))");
 ...
 query.exec("create table country(id int,
                                 name varchar(20))");
 ...
```

Modèle-Vue architecture

Architecture d'accès aux Bases de Données

```
void initializeModel(QSqlRelationalTableModel *model) {
 model->setTable("employee");
 model->setEditStrategy(QSqlTableModel::OnManualSubmit);
 model->setRelation(2,QSqlRelation("city",
                                   "id","name"));
 model->setRelation(3,QSqlRelation("country",
                                   "id","name"));
 model->setHeaderData(0, Qt::Horizontal,
                      QObject::tr("ID"));
 model->setHeaderData(1, Qt::Horizontal,
                      QObject::tr("Name"));
 ...
 model->select();
}
```

Qt : architecture Modèle-Vues et délégués

Architecture d'accès aux Bases de Données

```
QTableView *
createView(const QString &title, QSqlTableModel *model)
{
  QTableView *view = new QTableView;
  view->setModel(model);
  view->setItemDelegate(
                new QSqlRelationalDelegate(view)
      );
  view->setWindowTitle(title);
  return view;
}
```

Modèle-Vue architecture

Architecture d'accès aux Bases de Données

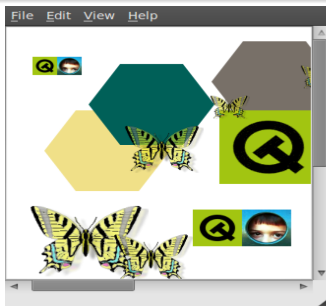
	ID	Name	City	Country
1	1	Toto	Oslo	Norway
2	2	Harald	Munich	Germany
3	3	Sam	San Jose	USA

Framework Graphics View

Dessiner des objets

Basé sur le modèle MVC, remplace la classe `QCanvas` de Qt3

- `QGraphicsScene` : la scène
- `QGraphicsView` : les vues
- `QGraphicsItem` : les objets



Framework Graphics View

`QGraphicsScene` : conteneur d'objets

- gérer un grand nombres d'éléments graphiques
- propager les évènements aux objets graphiques
- gérer les états des éléments (sélection, focus ...)
- fonctionnalités de rendu
- ...

`QGraphicsView` : vue de la scène

- widget de visualisation de la scène
- ...

Framework Graphics View

`QGraphicsItem` : éléments de scène

- éléments standards :
  - rectangle : `QGraphicsRectItem`
  - ellipse : `QGraphicsEllipseItem`
  - texte : `QGraphicsTextItem`
  - ...
- fonctionnalités pour écrire ses propres items
  - évènements souris, clavier
  - glisser-déposer
  - groupement d'éléments
  - détection de collisions
  - ...

Framework Graphics View

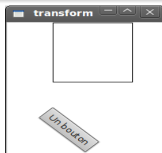
QGraphicsScene : exemple

```
#include <QApplication>
#include <QGraphicsScene>
#include <QGraphicsRectItem>
#include <QGraphicsView>
int main(int argc, char **argv) {
 QApplication app(argc, argv);
 QGraphicsScene scene;
 QGraphicsRectItem *rect =
                    scene.addRect(QRectF(0,0,100,100));
 rect->setFlag(QGraphicsItem::ItemIsMovable);
 QGraphicsView view(&scene);
 view.show();
 return app.exec();
}
```

Framework Graphics View

QGraphicsScene : widgets, transformation

```
QPushButton *bouton = new QPushButton("Un bouton");
QTransform matrix;
matrix.rotate(45);
matrix.translate(100, 100);
QGraphicsProxyWidget *proxy =
                       new QGraphicsProxyWidget();
proxy->setWidget(bouton);
proxy->setTransform(matrix);
scene.addItem(proxy);
```



Framework Graphics View

QGraphicsScene : évènements

```
#include <QGraphicsProxyWidget>
#include <QPointF>
class MyProxy : public QGraphicsProxyWidget {
public:
    MyProxy();
    qreal rotY();
    void setRotY(qreal rotation);
    QPointF center();
private:
    qreal _rotY;
    QPointF _center;
};
```

Framework Graphics View

QGraphicsScene : évènements

```
#include <QPushButton>
#include <QtWebKit/QWebView>
#include "myScene.h"

MyScene::MyScene() : QGraphicsScene() {
 QWebView *web = new QWebView();
 web->load(QUrl("http://www.developpez.com"));
 _proxy = new MyProxy();
 _proxy->setWidget(web);
 this->addItem(_proxy);
}
```

Framework Graphics View

QGraphicsScene : évènements

```
void MyScene::mouseMoveEvent(QGraphicsSceneMouseEvent*e)
 if(e->buttons() & Qt::LeftButton) {
  QPointF delta(e->scenePos() - e->lastScenePos());
  qreal rotation = delta.x();
  _proxy->setRotY(rotation + _proxy->rotY());
  QTransform matrix;
  matrix.rotate(_proxy->rotY(), Qt::ZAxis);
  matrix.translate(- _proxy->widget()->width()/2,
                   - _proxy->widget()->height()/2);
  _proxy->setTransform(matrix);
 }
}
```

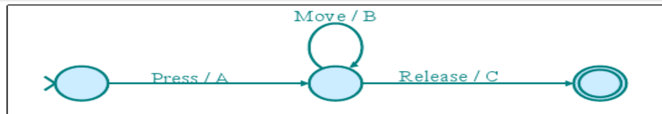
Framework Graphics View

QGraphicsScene : évènements

Modes d'interaction : Machines à Etats

Exemple : tracé élastique (Scott Hudson)

```
Accept Press for endpoint p1;
p2 = p1;
Draw line p1-p2;
Repeat
     Erase line p1-p2;
     p2 = current_position();
     Draw line p1-p2;
Until release event;
Act on line input;
```



Modes d'interaction : Machines à Etats

Etat A

```
Accept Press for endpoint p1;
p2 = p1;
Draw line p1-p2;
```

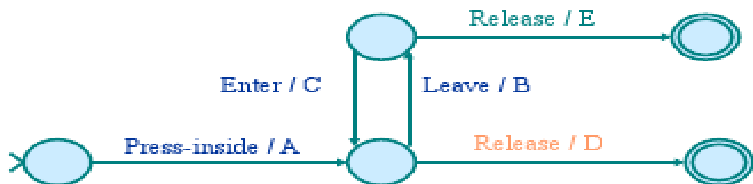
Etat B

```
Erase line p1-p2;
p2 = current_position();
Draw line p1-p2;
```

Etat C

```
Act on line input;
```

Modes d'interaction : Machines à Etats



Autre exemple : Bouton-poussoir

- action A : highlight button
- action B : unhighlight button
- action C : highlight button (action A)
- action D : do button action
- action E : do nothing

Modes d'interaction : Machines à Etats

Machines à Etats : boucle d'évènements

```
state = start_state;
for ( ; ; ) {
     raw_evt = wait_for_event();
     evt = transform_event(raw_evt);
     state = fsm_transition(state, evt);
}
```

Modes d'interaction : Machines à Etats

Machines à Etats : boucle d'évènements

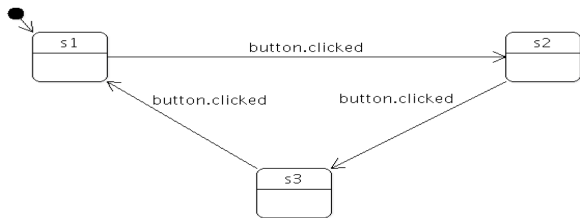
```
State fsm_transition(state, event) {
  switch(state) {
      case 1:
         switch(event.kind) {
         ...
           case MouseMove:
              action_B()
              state = 1
           case MouseRelease:
              action_C()
              state = 2
         }
         break;
      ...
```

Qt : State Machine Framework

classe `QtStateMachine`

- modèle basé sur SCXML
- hiérarchique : groupes d'états
- parallèle : éviter l'explosion combinatoire
- historique : sauvegarde de l'état courant
- permet de gérer ses propres évènements

Exemple : Bouton à trois états



Qt : State Machine Framework

QtStateMachine : Bouton à trois états

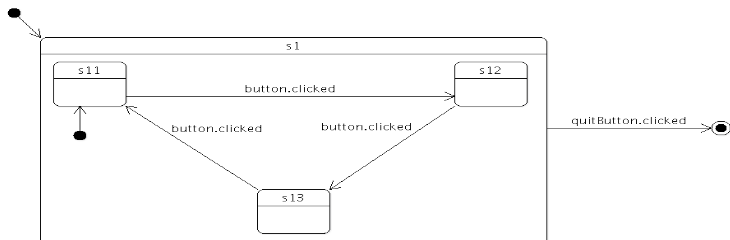
```
QStateMachine * sm = new QStateMachine();
QState *s1 = new QState();
QState *s2 = new QState();
QState *s3 = new QState();
s1->addTransition(button, SIGNAL(clicked()), s2);
s2->addTransition(button, SIGNAL(clicked()), s3);
s3->addTransition(button, SIGNAL(clicked()), s1);
sm->addState(s1);
sm->addState(s2);
sm->addState(s3);
sm->setInitialState(s1);
sm->start();
```

Qt : State Machine Framework

QtStateMachine : Bouton à trois états

```
s1->assignProperty(label, "text", "In state s1");
s2->assignProperty(label, "text", "In state s2");
s3->assignProperty(label, "text", "In state s3");
QWidget *box = new QWidget();
QVBoxLayout *layout = new QVBoxLayout(box);
layout->addWidget(button);
layout->addWidget(label);
QObject::connect(s3, SIGNAL(entered()),
                 box, SLOT(showMaximized()) );
QObject::connect(s3, SIGNAL(exited()),
                 box, SLOT(showMinimized()) );
```

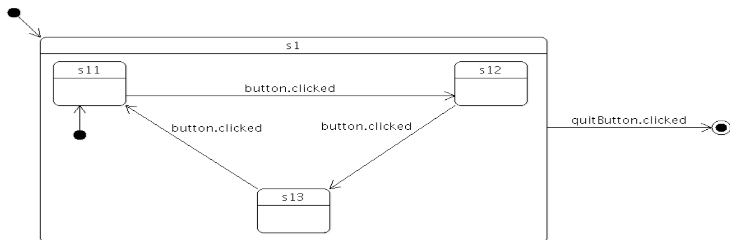
Qt : State Machine Framework



QtStateMachine : Groupement d'états

```
QState *s1 = new QState();
QState *s11 = new QState(s1);
QState *s12 = new QState(s1);
QState *s13 = new QState(s1);
s1->setInitialState(s11);
sm->addState(s1);
sm->setInitialState(s1);
```

Qt : State Machine Framework



QtStateMachine : Groupement d'états

```
QFinalState *s2 = new QFinalState( );
s1->addTransition(quitButton, SIGNAL(clicked()), s2);
sm->addState(s2);
QObject::connect(sm, SIGNAL(finished()),
                QApplication::instance(),SLOT(quit()));
sm->start();
```

Qt et OpenGL

Héritage `QGLWidget` : fenêtre 3D

Trois méthodes à surdéfinir :

- `virtual void initializeGL()`
- `virtual void paintGL()`
- `virtual void resizeGL(int w, int h)`

Qt et OpenGL

glwidget.h : surdéfinition de méthodes

```
#include <QGLWidget>
class QtLogo;
class GLWidget : public QGLWidget {
 Q_OBJECT
 public:
  GLWidget(QWidget *parent = 0);
  ~GLWidget();
  ...
 protected:
  void initializeGL();
  void paintGL();
  void resizeGL(int width, int height);
  void mousePressEvent(QMouseEvent *event);
  void mouseMoveEvent(QMouseEvent *event);
```

Qt et OpenGL

glwidget.h : signaux et slots

```
public slots:
 void setXRotation(int angle);
 void setYRotation(int angle);
 void setZRotation(int angle);
signals:
 void xRotationChanged(int angle);
 void yRotationChanged(int angle);
 void zRotationChanged(int angle);
```

Qt et OpenGL

glwidget.h : propriétés privées

```
private:
  QtLogo *logo;
  int xRot;
  int yRot;
  int zRot;
  QPoint lastPos;
  QColor qtGreen;
  QColor qtPurple;
};
```

Qt et OpenGL

glwidget.cpp : le constructeur

```
#include <math.h>
#include <QtGui>
#include <QtOpenGL>
#include "glwidget.h"
#include "qtlogo.h"

#ifndef GL_MULTISAMPLE
#define GL_MULTISAMPLE  0x809D
#endif
```

Qt et OpenGL

glwidget.cpp : constructeur, destructeur

```
GLWidget::GLWidget(QWidget *parent)
  : QGLWidget(QGLFormat(QGL::SampleBuffers), parent) {
 logo = 0;
 xRot = 0;
 yRot = 0;
 zRot = 0;
 qtGreen = QColor::fromCmykF(0.40, 0.0, 1.0, 0.0);
 qtPurple = QColor::fromCmykF(0.39, 0.39, 0.0, 0.0);
}
GLWidget::~GLWidget()
{
}
```

Qt et OpenGL

glwidget.cpp : les rotations

```
static void qNormalizeAngle(int &angle)
{
 while (angle < 0) angle += 360 * 16;
 while (angle > 360 * 16) angle -= 360 * 16;
}
void GLWidget::setXRotation(int angle)
{
 qNormalizeAngle(angle);
 if (angle != xRot) {
  xRot = angle;
  emit xRotationChanged(angle);
  updateGL();
 }
}
```

Qt et OpenGL

glwidget.cpp : les rotations

```
void GLWidget::setYRotation(int angle) {
 qNormalizeAngle(angle);
 if (angle != yRot) {
  yRot = angle;
  emit yRotationChanged(angle);
  updateGL();
 }
}
void GLWidget::setZRotation(int angle) {
 qNormalizeAngle(angle);
  if (angle != zRot) {
   zRot = angle;
   emit zRotationChanged(angle);
   updateGL();
```

Qt et OpenGL

glwidget.cpp : initialisation OpenGL

```
void GLWidget::initializeGL()
{
 qglClearColor(qtPurple.dark());
 logo = new QtLogo(this, 64);
 logo->setColor(qtGreen.dark());
 glEnable(GL_DEPTH_TEST);
 glEnable(GL_CULL_FACE);
 glShadeModel(GL_SMOOTH);
 glEnable(GL_LIGHTING);
 glEnable(GL_LIGHT0);
 glEnable(GL_MULTISAMPLE);
 static GLfloat lightPosition[4] = {0.5,5.0,7.0,1.0};
 glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
}
```

Qt et OpenGL

glwidget.cpp : affichage OpenGL

```
void GLWidget::paintGL()
{
 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
 glLoadIdentity();
 glTranslatef(0.0, 0.0, -10.0);
 glRotatef(xRot/16.0, 1.0, 0.0, 0.0);
 glRotatef(yRot/16.0, 0.0, 1.0, 0.0);
 glRotatef(zRot/16.0, 0.0, 0.0, 1.0);
 logo->draw();
 }
```

Qt et OpenGL

glwidget.cpp : retaillage OpenGL

```
void GLWidget::resizeGL(int width, int height)
{
 int side = qMin(width, height);
 glViewport((width-side)/2,(height-side)/2,side,side);
 glMatrixMode(GL_PROJECTION);
 glLoadIdentity();
 #ifdef QT_OPENGL_ES_1
  glOrthof(-0.5, +0.5, -0.5, +0.5, 4.0, 15.0);
 #else
  glOrtho(-0.5, +0.5, -0.5, +0.5, 4.0, 15.0);
 #endif
  glMatrixMode(GL_MODELVIEW);
}
```

Qt et OpenGL

glwidget.cpp : les événements

```
void GLWidget::mousePressEvent(QMouseEvent *event) {
 lastPos = event->pos();
}
void GLWidget::mouseMoveEvent(QMouseEvent *event) {
 int dx = event->x() - lastPos.x();
 int dy = event->y() - lastPos.y();
 if (event->buttons() & Qt::LeftButton) {
  setXRotation(xRot + 8 * dy);
  setYRotation(yRot + 8 * dx);
 } else if (event->buttons() & Qt::RightButton) {
  setXRotation(xRot + 8 * dy);
  setZRotation(zRot + 8 * dx);
 }
 lastPos = event->pos();
```

Qt et OpenGL

`window.h` : la fenêtre Qt

```
#ifndef WINDOW_H
#define WINDOW_H
#include <QWidget>
class QSlider;
class GLWidget;
class Window : public QWidget {
 Q_OBJECT
 public:
  Window();
 protected:
  void keyPressEvent(QKeyEvent *event);
```

Qt et OpenGL

`window.h` : les contrôleurs

```
 private:
  QSlider *createSlider();
  GLWidget *glWidget;
  QSlider *xSlider;
  QSlider *ySlider;
  QSlider *zSlider;
};
#endif
```

Qt et OpenGL

window.cpp : le constructeur

```
#include <QtGui>
#include "glwidget.h"
#include "window.h"
Window::Window()
{
 glWidget = new GLWidget;
 xSlider = createSlider();
 ySlider = createSlider();
 zSlider = createSlider();

 connect(xSlider, SIGNAL(valueChanged(int)),
         glWidget, SLOT(setXRotation(int)));
 connect(glWidget, SIGNAL(xRotationChanged(int)),
         xSlider, SLOT(setValue(int)));
```

Qt et OpenGL

`window.cpp` : les signaux et les slots

```
connect(ySlider, SIGNAL(valueChanged(int)),
        glWidget, SLOT(setYRotation(int)));
connect(glWidget, SIGNAL(yRotationChanged(int)),
        ySlider, SLOT(setValue(int)));
connect(zSlider, SIGNAL(valueChanged(int)),
        glWidget, SLOT(setZRotation(int)));
connect(glWidget, SIGNAL(zRotationChanged(int)),
        zSlider, SLOT(setValue(int)));
```

Qt et OpenGL

`window.cpp` : les contrôleurs

```
 QHBoxLayout *mainLayout = new QHBoxLayout;
 mainLayout->addWidget(glWidget);
 mainLayout->addWidget(xSlider);
 mainLayout->addWidget(ySlider);
 mainLayout->addWidget(zSlider);
 setLayout(mainLayout);

 xSlider->setValue(15 * 16);
 ySlider->setValue(345 * 16);
 zSlider->setValue(0 * 16);
 setWindowTitle(tr("Hello GL"));
}
```

Qt et OpenGL

window.cpp : la création des sliders

```
QSlider *Window::createSlider() {
 QSlider *slider = new QSlider(Qt::Vertical);
 slider->setRange(0, 360 * 16);
 slider->setSingleStep(16);
 slider->setPageStep(15 * 16);
 slider->setTickInterval(15 * 16);
 slider->setTickPosition(QSlider::TicksRight);
 return slider;
}
void Window::keyPressEvent(QKeyEvent *e) {
 if (e->key() == Qt::Key_Escape) close();
 else QWidget::keyPressEvent(e);
}
```

Qt et OpenGL

main.cpp : l'application

```
int main(int argc, char *argv[]) {
 QApplication app(argc, argv);
 Window window;
 window.resize(window.sizeHint());
 int desktopArea = QApplication::desktop()->width() *
                  QApplication::desktop()->height();
 int widgetArea = window.width() * window.height();
 if (((float)widgetArea / (float)desktopArea) < 0.75f)
   window.show();
 else
  window.showMaximized();
 return app.exec();
}
```

Conclusion

Débuter en Qt

- consulter la documentation `assistant`
- créer ses IHM à la main" sous son éditeur préféré `vi`, `emacs`, `gedit` ...
- s'inspirer des tutos, démos : `qtdemo`

aller plus loin en Qt

- utiliser un IDE : `qtcreator`
- faire ses IHM dans une IHM : `designer`

Bibliographie

Livres

- Yves Bailly :
  "Initiation à la programmation avec Python et C++" (2011)
  ed : Pearson
- Marc Summerfield:
  "Advanced Qt Programming : creating great software with C++ and Qt 4" (2010)
  ed : Prentice Hall
- Jasmine Blanchette, Marc Summerfield:
  "C++ GUI programming with Qt 4 second edition" (2011)
  ed : Prentice Hall
- Jasmine Blanchette, Marc Summerfield :
  "Qt 4 et C++ Programmation d'interfaces GUI" (2007)
  ed : Pearson

Bibliographie

Adresses “au Net”

- site officiel : `http://doc.qt.digia.com`
- la communauté française : `http://www.qtfr.org`
- club des pro. de l’info. : `http://qt-devnet.developpez.com`
- Eric Lecolinet : `http://www.infres.enst.fr/~elc/graph`
- Pierre Puiseux : `http://web.univ-pau.fr/~puiseux/enseignement`
- Thierry Vaira : `http://tvaira.free.fr`