# Tutorial Qt: Création d'une fenêtre personnalisée

par Olivier Boucard (Pixvault.org)

Date de publication : 19/03/2009

Cet article retrace la création d'une fenêtre personnailsée avec Qt.
Rendez-vous sur le forum pour tout commentaire ou question :
Source **customwindow**

# 1 - Introduction

Commençons par la question qui m'a donné envie d'écrire cet article : est-il possible de modifier l'aspect de ma fenêtre avec Qt? La réponse est oui, mais pas directement. En effet ce n'est pas Qt qui gère la fenêtre mais le window manager sous-jacent (Windows, KDE, Gnome, Xfce, MacOS X ...).

La souplesse de Qt va nous permettre de contourner le problème afin de ne pas se soucier du window manager. N'étant pas évidente de prime abord, la solution utilise pourtant les fonctionnalités de base de Qt et plus particulièrement de **QWidget**. Cet article introduit ma façon de faire (il en existe surement d'autres) qui, je l'espère, est simple et suffisamment efficace pour s'adapter à tout type de projet.

Avant de démarrer à proprement parler le tutoriel, voici le squelette de notre classe CustomWindow qui sera complété au fur et à mesure des explications.

***Attention !***
*Dans un premier temps le tutoriel n'est compatible qu'avec Qt 4.5 et supérieur (utilisation de WA_TranslucentBackground). Se référer à la section 6 (Adaptation du code à des versions plus anciennes de Qt4) pour une adapdation aux versions plus anciennes de Qt 4.*

customwindow.h

```
#ifndef CUSTOMWINDOW_H
#define CUSTOMWINDOW_H

#include <QtGui/QWidget>

class CustomWindow : public QWidget
{
Q_OBJECT

public:
  CustomWindow(QWidget *parent = 0);
  ~CustomWindow();

protected:
  void showEvent(QShowEvent *event);

private:
  void CenterOnScreen();
};

#endif
```

customwindow.cpp

```
#include "customwindow.h"

#include <QDesktopWidget>

CustomWindow::CustomWindow(QWidget *parent) : QWidget(parent)
{
  resize(512, 512);
  setWindowTitle(tr("Tutorial Qt: CustomWindow"));
}

CustomWindow::~CustomWindow()
{

}

void CustomWindow::showEvent(QShowEvent *event)
{
  Q_UNUSED(event);
```

customwindow.cpp
```
    CenterOnScreen();
}

void CustomWindow::CenterOnScreen()
{
  QDesktopWidget screen;

  QRect screenGeom = screen.screenGeometry(this);

  int screenCenterX = screenGeom.center().x();
  int screenCenterY = screenGeom.center().y();

  move(screenCenterX - width () / 2,
       screenCenterY - height() / 2);
}
```

Voici à quoi ressemble notre fenêtre au départ. Cette capture et les suivantes sont réalisées sous KDE 4 avec un thème Oxygen. À part coïncidence vous n'avez probablement pas le même résultat chez vous. Notre but est donc de faire ne sorte que tout le monde ait le même aspect quelque soit l'environnement :


*Fenêtre de base sous KDE*

## 2 - Création d'une fenêtre sans bordure

La première étape est donc d'enlever les bordures de la fenêtre, pour cela il suffit de rajouter cette ligne dans le constructeur :

customwindow.cpp

```
setWindowFlags(Qt::FramelessWindowHint);
```

Comme le fond de la fenêtre dépend aussi du window manager, ajoutons le code qui va permettre de mettre le notre. Nous allons pour cela surcharger la méthode paintEvent (**Doc Qt**) provenant de la classe **QWidget** (**Doc Qt**) .Ce n'est pas la façon Qt de changer la couleur de fond, il faudrait normalement passer par la palette et **QPalette::Window** (**Doc Qt**). Mais on aura besoin plus tard de travailler avec paintEvent (**Doc Qt**). Il faut commencer par ajouter la déclaration dans la section protected de CustomWindow :

customwindow.h

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

Ensuite, il faut rajouter la définition :

customwindow.cpp

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

  QBrush background(QColor(23, 23, 34));

  painter.setBrush(background);
  painter.setPen  (Qt::NoPen ); // No stroke

  painter.drawRect(0, 0, width(), height());
}
```

On obtient alors :

*Fenêtre sans bordure*

## 3 - Ajout d'une barre de titre

Le résultat est pour le moment plus que limité, les fonctionnalités basiques d'une fenêtre ne sont plus accessible directement :

1* La fenêtre n'a plus de titre;
2* La fenêtre ne peut plus être fermée;
3* La fenêtre ne peut plus être maximisée;
4* La fenêtre ne peut plus être minimisée;
5* La fenêtre ne peut plus être déplacée.

Ces fonctions sont normalement offertes par la barre de titre de la fenêtre. Mais le fait de créer une fenêtre sans bordure retire aussi cette partie. Il va donc falloir mettre en place notre propre barre de titre.

### 3-1 - Ajout d'un layout pour les éléments de la fenêtre

La première chose à faire est de préparer un layout qui va contenir notre barre de titre et la placer au bon endroit. On va donc créer un **QGridLayout** (**Doc Qt**) qui sera ajouté à notre CustomWindow. Commençons par ajouter l'include qu'il faut au début de notre fichier d'entête de CustomWindow :

customwindow.h
```
#include <QGridLayout>
```

Ensuite, ajouter la déclaration du membre dans la section private de CustomWindow :

customwindow.h
```
QGridLayout m_MainLayout;
```

Pour finir, dans le constructeur de CustomWindow, ajoutons ces lignes :

customwindow.cpp
```
m_MainLayout.setMargin(0);  // No space between window's element and the border
m_MainLayout.setSpacing(0); // No space between window's element
setLayout(&m_MainLayout);
```

### 3-2 - Création d'un widget WindowTitleBar

Nous allons maintenant ajouter une nouvelle classe qui va regrouper les éléments et les fonctionnalités de notre barre de titre. Voici le squelette :

windowtitlebar.h
```
#ifndef WINDOWTITLEBAR_H
#define WINDOWTITLEBAR_H

#include <QtGui/QWidget>

class WindowTitleBar : public QWidget
{
Q_OBJECT

public:
```

windowtitlebar.h
```
   WindowTitleBar(QWidget *parent = 0);
   ~WindowTitleBar();
};

#endif
```

windowtitlebar.cpp
```
#include "windowtitlebar.h"

WindowTitleBar::WindowTitleBar(QWidget *parent) : QWidget(parent)
{

}

WindowTitleBar::~WindowTitleBar()
{

}
```

Il faut ensuite ajouter une instance de WindowTitleBar à notre CustomWindow. Premièrement mettons l'include :

customwindow.h
```
#include "windowtitlebar.h"
```

Ensuite, mettons la déclaration en tant que membre private de CustomWindow :

customwindow.h
```
WindowTitleBar m_TitleBar;
```

Et dans le constructeur de CustomWindow, rajoutons ces lignes (la seconde va permettre de placer la barre de titre en haut de la fenêtre) :

customwindow.cpp
```
m_MainLayout.addWidget(&m_TitleBar, 0, 0, 1, 1);

m_MainLayout.setRowStretch(1, 1); // Put the title bar at the top of the window
```

Pour le moment rien n'apparait encore dans notre fenêtre. Commençons par donner une hauteur fixe à notre barre titre en rajoutant ceci dans le constructeur de WindowTitleBar :

windowtitlebar.cpp
```
setFixedHeight(33);
```

Dessinons maintenant notre barre. Pour gagner en performance, nous allons utiliser un cache qui ne sera mis à jour que lors d'un redimensionnement, pour cela la classe **QPixmap** (**Doc Qt**) est la plus appropriée. Tout d'abord mettons l'include :

windowtitlebar.h
```
#include <QPixmap>
```

windowtitlebar.h

Ensuite, mettons la déclaration en tant que membre private de WindowTitleBar :

windowtitlebar.h

```
QPixmap *m_Cache;
```

**Attention** : pensez à initialiser à NULL le pointeur m_Cache dans le constructeur.

Pour mettre à jour lors d'un redimensionnement et afficher le cache, il faut surcharger la méthode resizeEvent (**Doc Qt**) et paintEvent (**Doc Qt**). Ajoutons donc ces fonctions à la section protected de WindowTitleBar :

windowtitlebar.h

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

Et voici l'implémentation de nos deux méthodes :

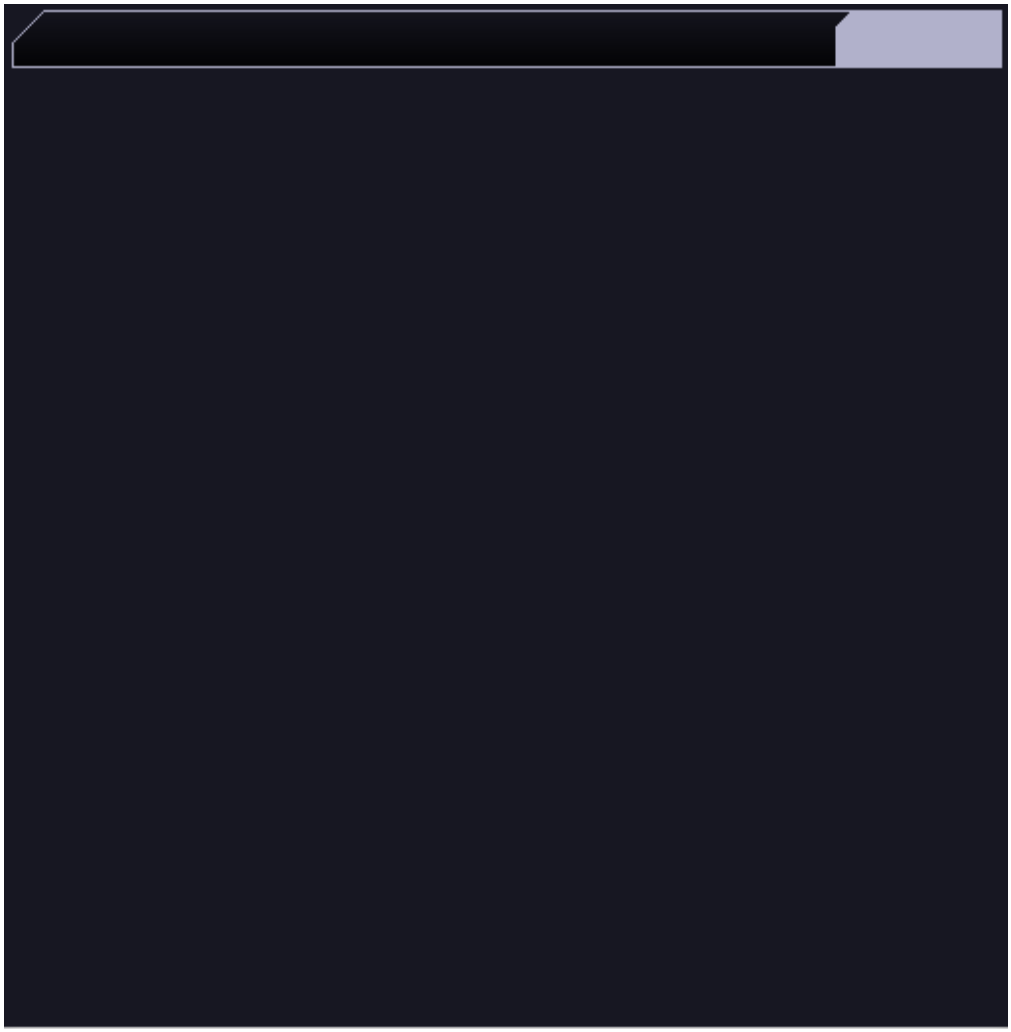
windowtitlebar.cpp

```
void WindowTitleBar::resizeEvent(QResizeEvent *event)
{
  Q_UNUSED(event);

  delete m_Cache; // Remove old cache

  m_Cache = new QPixmap(size());  // Create a cache with same size as the widget

  m_Cache->fill(Qt::transparent);  // Create a the transparent background

  QPainter painter(m_Cache); // Start painting the cache

  QColor lightBlue    (177, 177, 203, 255);
  QColor gradientStart(  0,   0,   0,   0);
  QColor gradientEnd  (  0,   0,   0, 220);

  QLinearGradient linearGrad(QPointF(0, 0), QPointF(0, height()));
  linearGrad.setColorAt(0, gradientStart);
  linearGrad.setColorAt(1, gradientEnd);

  /********** Title bar's frame **********/
  QPolygon frame;

  frame << QPoint(         20,  4)
        << QPoint(width() - 4,  4)
        << QPoint(width() - 4, 32)
        << QPoint(          4, 32)
        << QPoint(          4, 20);

  painter.setPen  (QPen  (lightBlue ));
  painter.setBrush(QBrush(linearGrad));

  painter.drawPolygon(frame);
  /*************************************/

  /********** Title bar's buttons area **********/
  QPolygon buttons;

  buttons << QPoint(width() - 80,  4)
          << QPoint(width() -  4,  4)
```

windowtitlebar.cpp

```
            << QPoint(width() -  4, 32)
            << QPoint(width() - 88, 32)
            << QPoint(width() - 88, 12);

  painter.setPen  (QPen  (lightBlue));
  painter.setBrush(QBrush(lightBlue));

  painter.drawPolygon(buttons);
  /*****************************************/
}

void WindowTitleBar::paintEvent(QPaintEvent *event)
{
  Q_UNUSED(event);

  if(m_Cache != NULL)
  {
    QPainter painter(this);

    painter.drawPixmap(0, 0, *m_Cache);
  }
}
```

Voici ce que l'on obtient :



*Fenêtre avec le contour de la barre de titre*

## 3-3 - Ajout du titre de la fenêtre

Pour afficher le titre à notre fenêtre nous allons utiliser un **QLabel** (**Doc Qt**). Tout d'abord mettons l'include :

windowtitlebar.h
```
#include <QLabel>
```

Ensuite, mettons la déclaration en tant que membre private de WindowTitleBar :

windowtitlebar.h
```
QLabel m_Title;
```

**Attention** : pensez à l'initialiser avec this dans le constructeur.

Nous allons ici utiliser les style sheets afin de modifier la font de notre titre grâce à cette ligne dans le constructeur :

windowtitlebar.cpp
```
m_Title.setStyleSheet("color: white; font-family: Sans; font-weight: bold; font-size: 14px");
```

Pour le positionnement et les dimensions du titre, ajoutons ces deux lignes à resizeEvent (**Doc Qt**) :

windowtitlebar.cpp
```
m_Title.move  (           28,  4);
m_Title.resize(width() - 116, 29);
```

Maintenant, pour pouvoir garder notre barre de titre synchronisée avec le windowTitle défini dans **QWidget** (**Doc Qt**), nous allons utiliser le système de signaux et de slots. L'implémentation de base de setWindowTitle (**Doc Qt**) n'émettant pas de signal, il nous faut surcharger cette fonction. Ajoutons la déclaration dans la section public de CustomWindow :

customwindow.h
```
void setWindowTitle(const QString &title);
```

Ainsi que le signal en question :

customwindow.h
```
signals:
  void WindowTitleChanged();
```

L'implémentation de notre fonction est la suivante :

customwindow.cpp
```
void CustomWindow::setWindowTitle(const QString &title)
{
  QWidget::setWindowTitle(title);
```

customwindow.cpp
```
   emit WindowTitleChanged();
}
```

Ajoutons maintenant le slot correspondant dans notre WindowTitleBar, en commençant par le header :

windowtitlebar.h
```
public slots:
  void UpdateWindowTitle();
```

Dont voici la définition :

windowtitlebar.cpp
```
void WindowTitleBar::UpdateWindowTitle()
{
  m_Title.setText(window()->windowTitle());
}
```

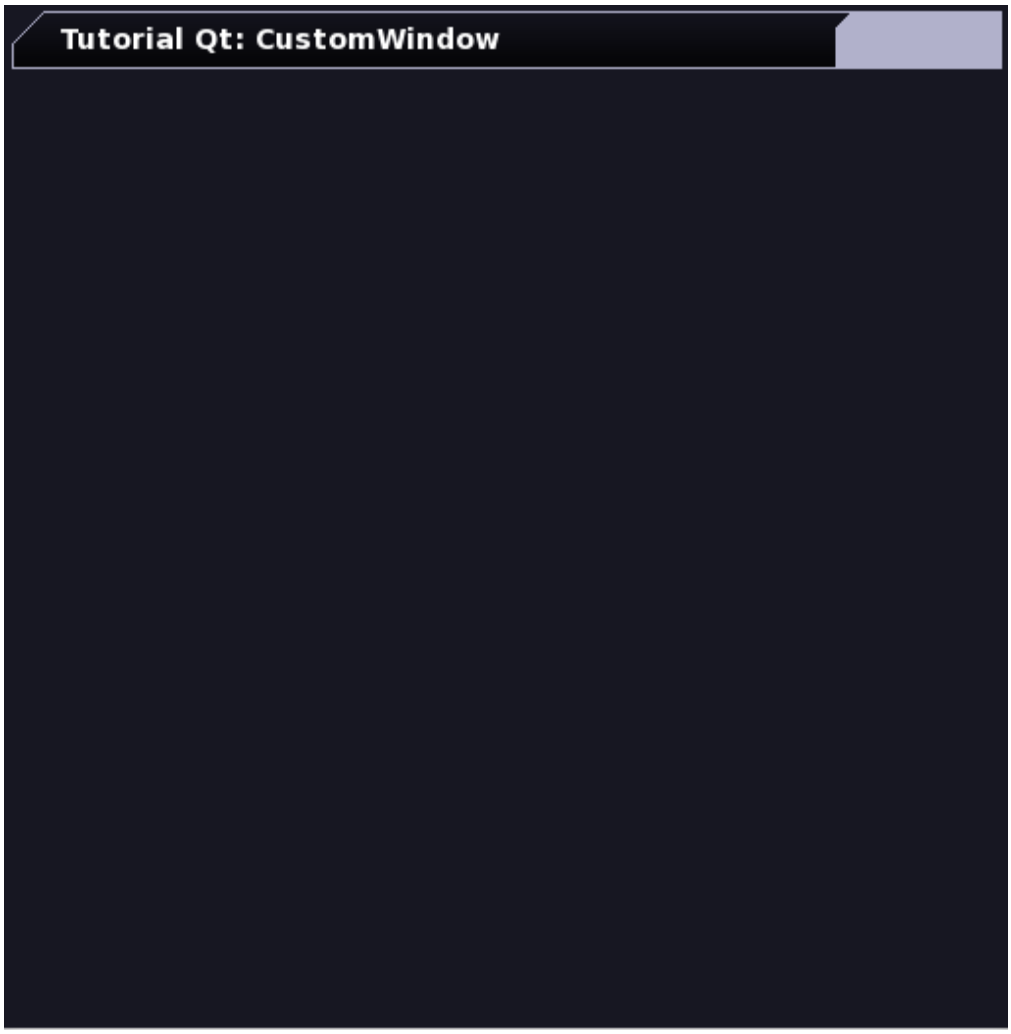
Pour s'assurer qu'un titre soit bien présent dès le départ, rajoutons cette ligne au constructeur de WindowTitleBar :

windowtitlebar.cpp
```
UpdateWindowTitle();
```

Pour finir, il reste à connecter le signal et le slot, pour cela, rajoutons cette ligne au tout début du constructeur de CustomWindow :

customwindow.cpp
```
connect(this       , SIGNAL(WindowTitleChanged()),
        &m_TitleBar, SLOT  (UpdateWindowTitle ()));
```

On arrive à ce résultat :




*Fenêtre avec le titre*

## 3-4 - Ajout des boutons standards

Notre fenêtre commence à s'améliorer niveau apparence mais pas niveau fonctionnalités. On va donc y remédier en rajoutant les boutons suivants :

1* Réduire la fenêtre;
2* Maximiser/restaurer la fenêtre;
3* Fermer la fenêtre.

Pour cela, nous allons créer une classe WindowButton. On ne reviendra pas sur les fonctions détaillées dans les autres classes. Cependant, vous noterez l'utilisation d'énumérations afin de gérer les différents type de button ainsi que les différents états dans la même classe. Voici le squelette :

windowbutton.h

```
#ifndef WINDOWBUTTON_H
#define WINDOWBUTTON_H

#include <QtGui/QAbstractButton>
#include <QPixmap>

class WindowButton : public QAbstractButton
{
Q_OBJECT
```

windowbutton.h

```
public:
  enum ButtonType
  {
    BUTTON_MINIMIZE, BUTTON_MAXIMIZE, BUTTON_CLOSE
  };

  WindowButton(ButtonType type, QWidget *parent = 0);
  ~WindowButton();

protected:
  void resizeEvent(QResizeEvent *event);
  void paintEvent (QPaintEvent  *event);

private:
  enum ButtonState
  {
    STATE_NORMAL, STATE_HOVERED, STATE_CLICKED
  };

  ButtonType  m_Type   ;
  ButtonState m_State  ;
  QPixmap    *m_Normal ;
  QPixmap    *m_Hovered;
  QPixmap    *m_Clicked;

  void InitPixmaps (                );
  void InitPixmap  (QPixmap **pixmap);
  void InitMinimize(                );
  void InitMaximize(                );
  void InitClose   (                );
};

#endif
```

windowbutton.cpp

```
#include "windowbutton.h"

#include <QPainter>

WindowButton::WindowButton(ButtonType type, QWidget *parent) : QAbstractButton(parent),
  m_Type   (type        ),
  m_State  (STATE_NORMAL),
  m_Normal (NULL        ),
  m_Hovered(NULL        ),
  m_Clicked(NULL        )
{
}

WindowButton::~WindowButton()
{
  delete m_Normal ;
  delete m_Hovered;
  delete m_Clicked;
}

void WindowButton::resizeEvent(QResizeEvent *event)
{
  Q_UNUSED(event);

  InitPixmaps();
}

void WindowButton::paintEvent(QPaintEvent  *event)
{
  Q_UNUSED(event);

  QPainter painter(this);
```

windowbutton.cpp

```
  if(isEnabled())
  {
    switch(m_State)
    {
    case STATE_NORMAL:
      if(m_Normal  != NULL) painter.drawPixmap(0, 0, *m_Normal );
      break;
    case STATE_HOVERED:
      if(m_Hovered != NULL) painter.drawPixmap(0, 0, *m_Hovered);
      break;
    case STATE_CLICKED:
      if(m_Clicked != NULL) painter.drawPixmap(0, 0, *m_Clicked);
      break;
    }
  }
  else
  {
    if(m_Normal != NULL) painter.drawPixmap(0, 0, *m_Normal);
  }
}

void WindowButton::InitPixmaps()
{
  // Delete previous button
  InitPixmap(&m_Normal );
  InitPixmap(&m_Hovered);
  InitPixmap(&m_Clicked);

  switch(m_Type)
  {
  case BUTTON_MINIMIZE:
    InitMinimize();
    break;
  case BUTTON_MAXIMIZE:
    InitMaximize();
    break;
  case BUTTON_CLOSE:
    InitClose();
    break;
  }
}

void WindowButton::InitPixmap(QPixmap **pixmap)
{
  delete *pixmap;

  *pixmap = new QPixmap(size());

  (*pixmap)->fill(Qt::transparent);
}

void WindowButton::InitMinimize()
{
  /********** Button's border **********/
  QPolygon border;

  border << QPoint(          0,            4)
         << QPoint(          4,            0)
         << QPoint(width() - 1,            0)
         << QPoint(width() - 1, height() - 1)
         << QPoint(          0, height() - 1);
  /************************************/

  /********** Button's symbol **********/
  QPolygon symbol;

  symbol << QPoint(          4, height() - 8)
         << QPoint(width() - 4, height() - 8)
         << QPoint(width() - 4, height() - 4)
         << QPoint(          4, height() - 4);
  /************************************/
```

windowbutton.cpp

```
  QColor gradientStart(  0,   0,   0,   0);
  QColor gradientEnd  (  0,   0,   0, 220);

  QLinearGradient linearGrad(QPointF(0, 0), QPointF(0, height()));
  linearGrad.setColorAt(0, gradientStart);
  linearGrad.setColorAt(1, gradientEnd  );

  QLinearGradient invertlinearGrad(QPointF(0, 0), QPointF(0, height()));
  invertlinearGrad.setColorAt(0, gradientEnd  );
  invertlinearGrad.setColorAt(1, gradientStart);

  QPainter painter;

  /********** Normal **********/
  painter.begin(m_Normal);

  painter.setPen  (QPen(Qt::black   ));
  painter.setBrush(QBrush(linearGrad));

  painter.drawPolygon(border);

  painter.setPen  (Qt::NoPen        );
  painter.setBrush(QBrush(Qt::black));

  painter.drawPolygon(symbol);

  painter.end();
  /****************************/

  /********** Hovered **********/
  painter.begin(m_Hovered);

  painter.setPen  (QPen  (Qt::black ));
  painter.setBrush(QBrush(linearGrad));

  painter.drawPolygon(border);

  painter.setPen  (Qt::NoPen        );
  painter.setBrush(QBrush(Qt::white));

  painter.drawPolygon(symbol);

  painter.end();
  /****************************/

  /********** Clicked **********/
  painter.begin(m_Clicked);

  painter.setPen  (QPen  (Qt::black       ));
  painter.setBrush(QBrush(invertlinearGrad));

  painter.drawPolygon(border);

  painter.setPen  (Qt::NoPen        );
  painter.setBrush(QBrush(Qt::white));

  painter.drawPolygon(symbol);

  painter.end();
  /****************************/
}

void WindowButton::InitMaximize()
{
  /********** Button's border **********/
  QPolygon border;

  border << QPoint(          0,            0)
         << QPoint(width() - 5,            0)
         << QPoint(width() - 1,            4)
```

windowbutton.cpp

```
        << QPoint(width() - 1, height() - 1)
        << QPoint(          0, height() - 1);
/*************************************/

/********** Button's symbol **********/
QPolygon symbol1, symbol2;

symbol1 << QPoint(          4, 4)
        << QPoint(width() - 4, 4)
        << QPoint(width() - 4, 8)
        << QPoint(          4, 8);

symbol2 << QPoint(          4,            8)
        << QPoint(width() - 4,            8)
        << QPoint(width() - 4, height() - 4)
        << QPoint(          4, height() - 4);
/*************************************/

QColor gradientStart(  0,   0,   0,   0);
QColor gradientEnd   (  0,   0,   0, 220);

QLinearGradient linearGrad(QPointF(0, 0), QPointF(0, height()));
linearGrad.setColorAt(0, gradientStart);
linearGrad.setColorAt(1, gradientEnd  );

QLinearGradient invertlinearGrad(QPointF(0, 0), QPointF(0, height()));
invertlinearGrad.setColorAt(0, gradientEnd  );
invertlinearGrad.setColorAt(1, gradientStart);

QPainter painter;

/********** Normal **********/
painter.begin(m_Normal);

painter.setPen  (QPen(Qt::black   ));
painter.setBrush(QBrush(linearGrad));

painter.drawPolygon(border);

painter.setPen  (QPen  (Qt::black));
painter.setBrush(QBrush(Qt::black));

painter.drawPolygon(symbol1);

painter.setPen  (QPen(Qt::black));
painter.setBrush(Qt::NoBrush    );

painter.drawPolygon(symbol2);

painter.end();
/*************************/

/********** Hovered **********/
painter.begin(m_Hovered);

painter.setPen  (QPen  (Qt::black       ));
painter.setBrush(QBrush(linearGrad));

painter.drawPolygon(border);

painter.setPen  (QPen  (Qt::white));
painter.setBrush(QBrush(Qt::white));

painter.drawPolygon(symbol1);

painter.setPen  (QPen(Qt::white));
painter.setBrush(Qt::NoBrush    );

painter.drawPolygon(symbol2);

painter.end();
```

windowbutton.cpp

```
  /****************************/

  /********** Clicked **********/
  painter.begin(m_Clicked);

  painter.setPen  (QPen  (Qt::black ));
  painter.setBrush(QBrush(invertlinearGrad));

  painter.drawPolygon(border);

  painter.setPen  (QPen  (Qt::white));
  painter.setBrush(QBrush(Qt::white));

  painter.drawPolygon(symbol1);

  painter.setPen  (QPen(Qt::white));
  painter.setBrush(Qt::NoBrush    );

  painter.drawPolygon(symbol2);

  painter.end();
  /****************************/
}

void WindowButton::InitClose()
{
  /********** Button's border **********/
  QPolygon border;

  border << QPoint(          0,            4)
         << QPoint(          4,            0)
         << QPoint(width() - 5,            0)
         << QPoint(width() - 1,            4)
         << QPoint(width() - 1, height() - 5)
         << QPoint(width() - 5, height() - 1)
         << QPoint(          4, height() - 1)
         << QPoint(          0, height() - 5);
  /************************************/

  /********** Button's symbol **********/
  QLine symbol1(QPoint(          4, 4), QPoint(width() - 5, height() - 5));
  QLine symbol2(QPoint(width() - 5, 4), QPoint(          4, height() - 5));
  /************************************/

  QColor gradientStart(  0,   0,   0,   0);
  QColor gradientEnd  (  0,   0,   0, 220);

  QLinearGradient linearGrad(QPointF(0, 0), QPointF(0, height()));
  linearGrad.setColorAt(0, gradientStart);
  linearGrad.setColorAt(1, gradientEnd  );

  QLinearGradient invertlinearGrad(QPointF(0, 0), QPointF(0, height()));
  invertlinearGrad.setColorAt(0, gradientEnd  );
  invertlinearGrad.setColorAt(1, gradientStart);

  QPainter painter;

  /********** Normal **********/
  painter.begin(m_Normal);

  painter.setPen  (QPen(Qt::black   ));
  painter.setBrush(QBrush(linearGrad));

  painter.drawPolygon(border);

  painter.setPen(QPen(QBrush(Qt::black), 2.0));

  painter.drawLine(symbol1);
  painter.drawLine(symbol2);

  painter.end();
```

windowbutton.cpp

```
    /***************************/

    /********** Hovered **********/
    painter.begin(m_Hovered);

    painter.setPen  (QPen  (Qt::black ));
    painter.setBrush(QBrush(linearGrad));

    painter.drawPolygon(border);

    painter.setPen(QPen(QBrush(Qt::white), 2.0));

    painter.drawLine(symbol1);
    painter.drawLine(symbol2);

    painter.end();
    /***************************/

    /********** Clicked **********/
    painter.begin(m_Clicked);

    painter.setPen  (QPen  (Qt::black       ));
    painter.setBrush(QBrush(invertlinearGrad));

    painter.drawPolygon(border);

    painter.setPen(QPen(QBrush(Qt::white), 2.0));

    painter.drawLine(symbol1);
    painter.drawLine(symbol2);

    painter.end();
    /***************************/
}
```

Pour le moment nos buttons ne font encore rien de particulier. Nous allons donc ajouter le code qui va permettre le changement d'image sur les évènements souris. Plaçons dans la section protected de WindowButton les lignes suivantes :

windowbutton.h

```
void enterEvent       (QEvent      *event);
void leaveEvent       (QEvent      *event);
void mousePressEvent  (QMouseEvent *event);
void mouseReleaseEvent(QMouseEvent *event);
```

Voici l'implémentation, à noter deux astuces :

1* L'appel à la fonction parente pour mousePressEvent (**Doc Qt**) et mouseReleaseEvent (**Doc Qt**). Cela permet de garder l'émission du signal clicked
2* L'utilisation de underMouse (**Doc Qt**) pour réafficher la bonne image lorsque le button de la souris est relâché

windowbutton.cpp

```
void WindowButton::enterEvent(QEvent *event)
{
  Q_UNUSED(event);

  m_State = STATE_HOVERED;

  update();
}

void WindowButton::leaveEvent(QEvent *event)
```

windowbutton.cpp

```
{
  Q_UNUSED(event);

  m_State = STATE_NORMAL;

  update();
}

void WindowButton::mousePressEvent(QMouseEvent *event)
{
  QAbstractButton::mousePressEvent(event);

  m_State = STATE_CLICKED;

  update();
}

void WindowButton::mouseReleaseEvent(QMouseEvent *event)
{
  QAbstractButton::mouseReleaseEvent(event);

  if(underMouse()) m_State = STATE_HOVERED;
  else             m_State = STATE_NORMAL;

  update();
}
```

Il ne reste plus qu'à lier ces boutons aux fonctionnalités souhaitées. Tout d'abord, ajoutons, dans la classe WindowTitleBar, les slots nécessaires :

windowtitlebar.h

```
void Minimized();
void Maximized();
void Quit     ();
```

Pour l'implémention des slots, voici le code :

windowtitlebar.cpp

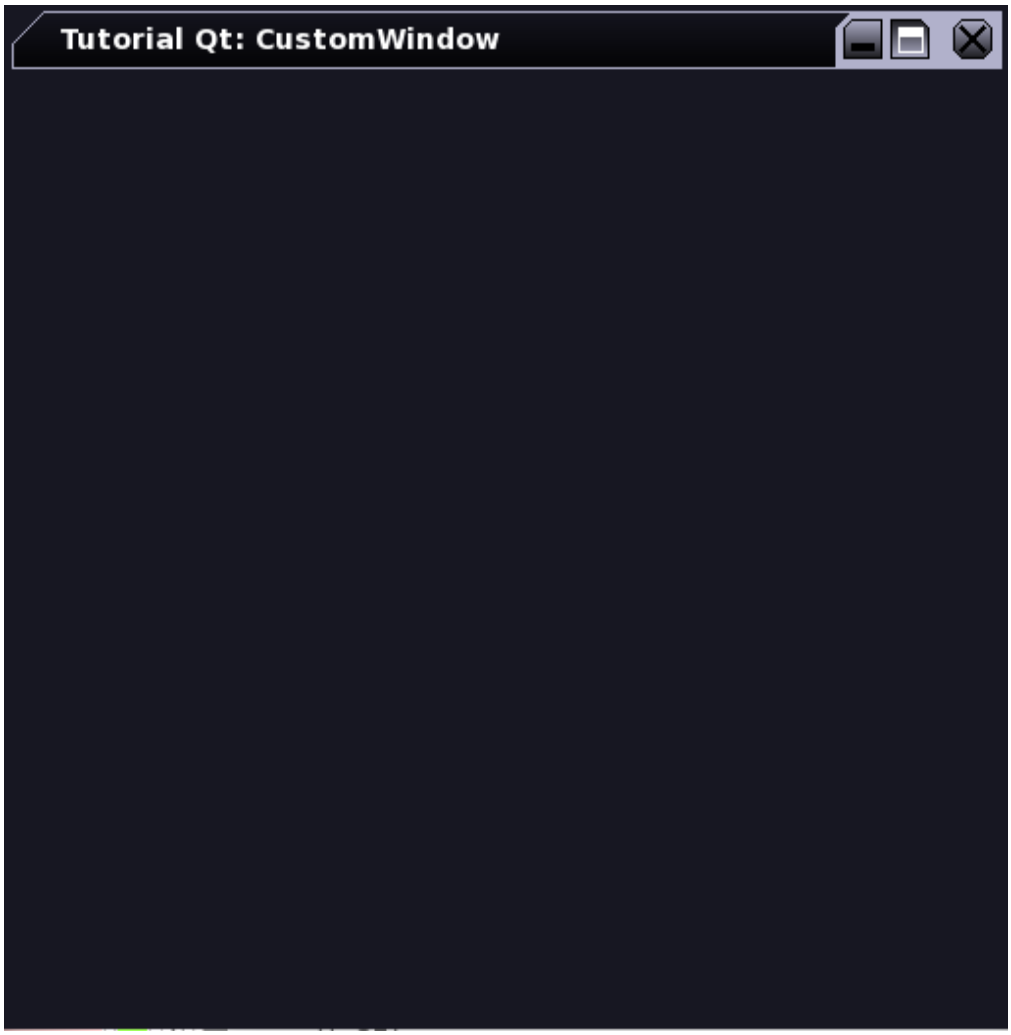
```
void WindowTitleBar::Minimized()
{
  window()->showMinimized();
}

void WindowTitleBar::Maximized()
{
  if(window()->windowState() == Qt::WindowMaximized)
  {
    window()->showNormal();
  }
  else
  {
    window()->showMaximized();
  }
}

void WindowTitleBar::Quit()
{
  qApp->quit();
}
```

Enfin il faut relier les signaux et slots. Pour ce faire, plaçons ces lignes dans le constructeur de WindowTitleBar :

windowtitlebar.cpp

```
connect(&m_Minimize, SIGNAL(clicked  ()),
        this       , SLOT  (Minimized()));

connect(&m_Maximize, SIGNAL(clicked  ()),
        this,        SLOT  (Maximized()));

connect(&m_Close   , SIGNAL(clicked  ()),
        this       , SLOT  (Quit     ()));
```

Notre fenêtre commence maitenant à ressembler et à fonctionner comme une vrai :




*Fenêtre avec les boutons*

## 3-5 - Ajout du déplacement de la fenêtre

Nous allons maintenant nous pencher sur la manière de déplacer notre fenêtre qui reste pour le moment désespérément au milieu de l'écran. Il faut, pour cela, regarder du côté des mouseEvent de notre classe WindowTitleBar. Commençons par les déclarer dans la section protected :

windowtitlebar.h

```
void mousePressEvent  (QMouseEvent *event);
void mouseReleaseEvent(QMouseEvent *event);
void mouseMoveEvent   (QMouseEvent *event);
```

windowtitlebar.h

Nous avons aussi besoin d'une variable qui stockera les données relatives au déplacement, cette ligne sont à placer dans la section private :

windowtitlebar.h

```
QPoint m_Diff;
```

L'idée de l'implémentation est simple. Au moment où l'on clique dans la barre de titre, on sauvegarde la position de souris dans la fenêtre. Ensuite lors des déplacements, on utilise la position de la souris dans l'espace écran auquel on soustrait la position de la souris dans la fenêtre. De cette façon le déplacement se fait relativement au coin haut gauche de la fenêtre. Autre astuce, l'utilisation du setCursor (**Doc Qt**) pour apporter un feedback visuel :

windowtitlebar.cpp

```
void WindowTitleBar::mousePressEvent(QMouseEvent *event)
{
  m_Diff = event->pos();

  setCursor(QCursor(Qt::SizeAllCursor));
}

void WindowTitleBar::mouseReleaseEvent(QMouseEvent *event)
{
  Q_UNUSED(event);

  setCursor(QCursor(Qt::ArrowCursor));
}

void WindowTitleBar::mouseMoveEvent(QMouseEvent *event)
{
  QPoint p = event->globalPos();

  window()->move(p - m_Diff);
}
```

# 4 - Ajout du redimensionnement de la fenêtre

## 4-1 - Utilisation de QSizeGrip

Qt propose un classe toute faite pour le redimenssionement des fenêtres, il s'agit de **QSizeGrip** (**Doc Qt**). Nous allons donc l'utiliser en plaçant une instance dans CustomWindow :

customwindow.h
```
#include <QSizeGrip>
```

Déclarons le nouveau membre dans la section private :

customwindow.h
```
QSizeGrip m_SizeGrip;
```

**Attention** : pensez à l'initialiser en passant this comme parent.

Pour le placer correctement nous allons utiliser resizeEvent (**Doc Qt**) que nous déclarons dans la section protected. Il serait tout à fait possible de le mettre dans le layout mais ce serait une perte de place du fait de sa forme triangulaire.

customwindow.h
```
void resizeEvent(QResizeEvent *event);
```

L'implémentaition de resizeEvent (**Doc Qt**) est la suivante :

customwindow.cpp
```
void CustomWindow::resizeEvent(QResizeEvent *event)
{
  m_SizeGrip.move   (width() - 32, height() - 32);
  m_SizeGrip.resize(          32,            32);
}
```

On arrive à ce résultat :


*Fenêtre avec le QSizeGrip*

## 5 - Modifier la forme de votre fenêtre

Nous allons maintenant terminer de modifier l'apparence de notre fenêtre en lui donnant une forme particulière (i.e. non rectangulaire). La méthode est toujours la même, on dessine l'image dans un cache lors du redimensionnement et on l'affiche dans le paintEvent (**Doc Qt**). Je vous laisse donc rajouter la déclaration du cache dans l'entête de la classe CustomWindow. Intéressons-nous directement à la réimplémentation des méthodes resizeEvent (**Doc Qt**) et paintEvent (**Doc Qt**). Notez que le **QSizeGrip** (**Doc Qt**) voit sa position modifiée. Ces deux méthodes deviennent pour le coup :

customwindow.cpp

```
void CustomWindow::paintEvent(QPaintEvent *event)
{
  Q_UNUSED(event);

  if(m_Cache != NULL)
  {
    QPainter painter(this);

    painter.drawPixmap(0, 0, *m_Cache);
  }
}

void CustomWindow::resizeEvent(QResizeEvent *event)
{
  Q_UNUSED(event);

  delete m_Cache;

  m_Cache = new QPixmap(size());

  m_Cache->fill(Qt::transparent);

  QPainter painter(m_Cache);

  QColor darkBlue ( 23,  23,  34);
  QColor lightBlue(177, 177, 203);

  /********** Window's background **********/
  QPolygon background;

  background << QPoint(           0,            16)
             << QPoint(          16,             0)
             << QPoint(width() -  1,             0)
             << QPoint(width() -  1, height() - 33)
             << QPoint(width() - 17, height() - 17)
             << QPoint(         272, height() - 17)
             << QPoint(         256, height() -  1)
             << QPoint(          16, height() -  1)
             << QPoint(          16,           272)
             << QPoint(           0,           256);

  painter.setPen  (QPen  (darkBlue));
  painter.setBrush(QBrush(darkBlue));

  painter.drawPolygon(background);
  /*****************************************/

  /********** Window's frame **********/
  QPolygon frame;

  frame << QPoint(           4,            20)
        << QPoint(          20,             4)
        << QPoint(width() -  4,             4)
        << QPoint(width() -  4, height() - 37)
        << QPoint(width() - 20, height() - 21)
        << QPoint(         268, height() - 21)
        << QPoint(         252, height() -  5)
        << QPoint(          20, height() -  5)
```

customwindow.cpp
```
        << QPoint(          20,           268)
        << QPoint(           4,           252);

  painter.setPen  (QPen(lightBlue));
  painter.setBrush(Qt::NoBrush    );

  painter.drawPolygon(frame);
  /*************************************/

  m_SizeGrip.move  (width() - 32, height() - 49);
  m_SizeGrip.resize(          32,            32);
}
```

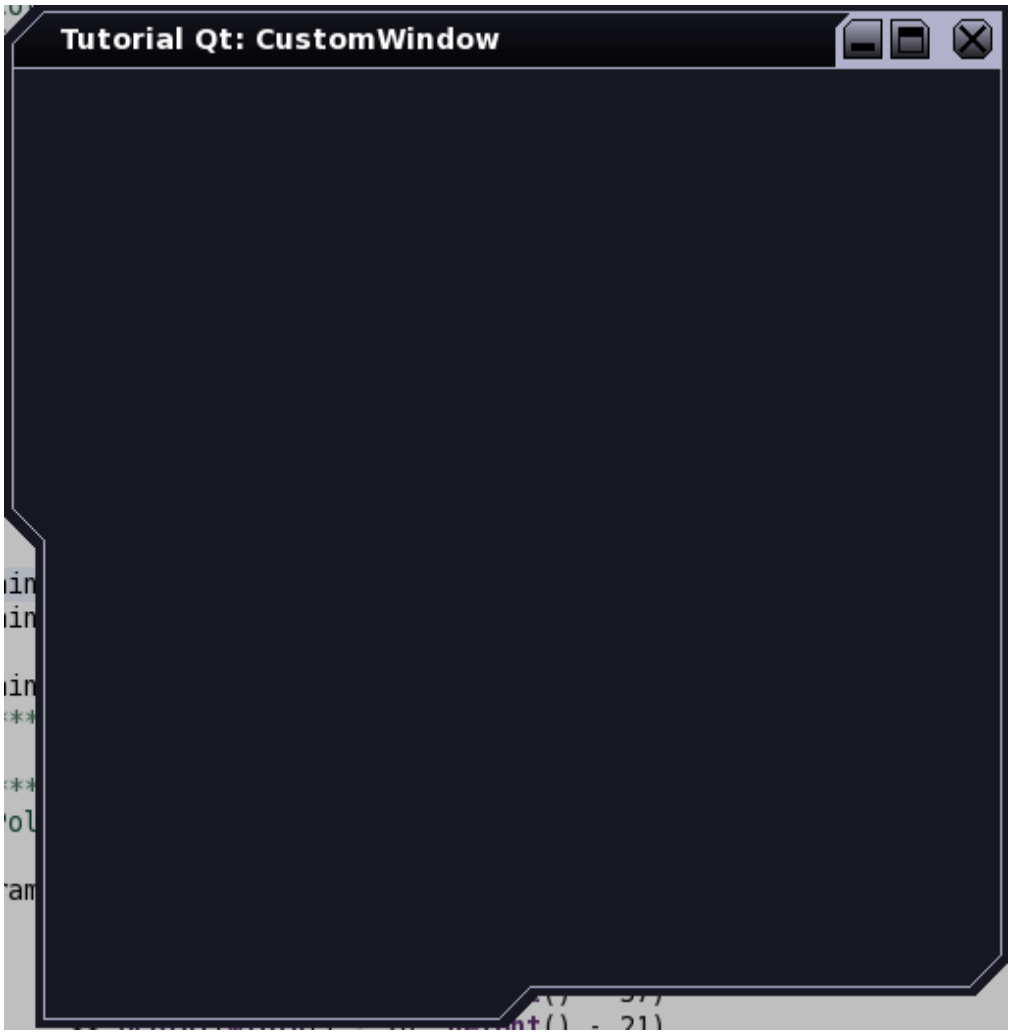
Le rendu du **QSizeGrip** (**Doc Qt**) ne collant plus avec le design de notre fenêtre, je vous propose de le cacher tout en gardant la fonctionnalité. Il aurait aussi été tout à fait possible d'en changer l'apparence, je vous le laisse dons en exercice. Dans le constructeur, rajoutons cette ligne :

customwindow.cpp
```
m_SizeGrip.setStyleSheet("image: none");
```

Malgré tout, les contours de notre fenêtre restent toujours rectangulaires. Ne vous inquiétez pas, il reste une ligne de code à ajouter au constructeur pour corriger ça :

customwindow.cpp
```
setAttribute(Qt::WA_TranslucentBackground);
```

Voila, notre fenêtre personalisée est maintenant terminée :




*Fenêtre terminée*

## 6 - Adaptation du code à des versions plus anciennes de Qt4

Pour le moment nous nous sommes intéressés à une implémentation pour Qt 4.5 et supérieur uniquement. Le passage à une version supportant des versions antérieures de Qt est cependant aisé. De plus, étant donné que la méthode utilisée nécessite un compositor sur X11, nous allons aussi corriger ceci. Ajoutons tout d'abord ceci avant la définition du constructeur de CustomWindow :

customwindow.cpp

```
#ifdef Q_WS_X11
  #include <QX11Info>
#endif
```

Ensuite, dans le constructeur remplaçons la ligne setAttribute(Qt::WA_TranslucentBackground); par :

customwindow.cpp

```
#if QT_VERSION >= 0x040500
  #ifdef Q_WS_X11
    if(x11Info().isCompositingManagerRunning()) setAttribute(Qt::WA_TranslucentBackground);
  #else
    setAttribute(Qt::WA_TranslucentBackground);
  #endif
#endif
```

Pour finir, il suffit de redéfinir le paintEvent (**Doc Qt**) ainsi :

customwindow.cpp

```
void CustomWindow::paintEvent(QPaintEvent *event)
{
  Q_UNUSED(event);

  if(m_Cache != NULL)
  {
    QPainter painter(this);

    painter.drawPixmap(0, 0, *m_Cache);

#if QT_VERSION >= 0x040500
    if(!testAttribute(Qt::WA_TranslucentBackground)) setMask(m_Cache->mask());
#else
    setMask(m_Cache->mask());
#endif
  }
}
```

## 7 - Ajout d'une ombre sous la fenêtre

N'ayant pas trouvé pour le moment une solution satisfaisante, cette partie est laissée en suspend (toutes idées est la bienvenue). Voici pour le moment les pistes explorées sans succès :

1* Dessiner la forme de la fenêtre et passer un algorithme de flou gaussien dessus : trop lent
2* Utiliser une brosse en dégradé radial sur la composante alpha pour dessiner la forme de la fenêtre : ne donne pas le résultat attendu

## 8 - Aller plus loin...

Voici pour le moment les points sur lequel on peut travailler pour rendre notre fenêtre encore plus utilisable :

1* Simuler des bordures pour le redimensionnement de la fenêtre
2* Ajouter une barre d'état
3* Comment rendre la fenêtre thèmable à l'aide de dessin SVG ?
4* Comment adapter le contenu à une fenêtre non rectangulaire ? Comment gérer ça au niveau des layouts ?