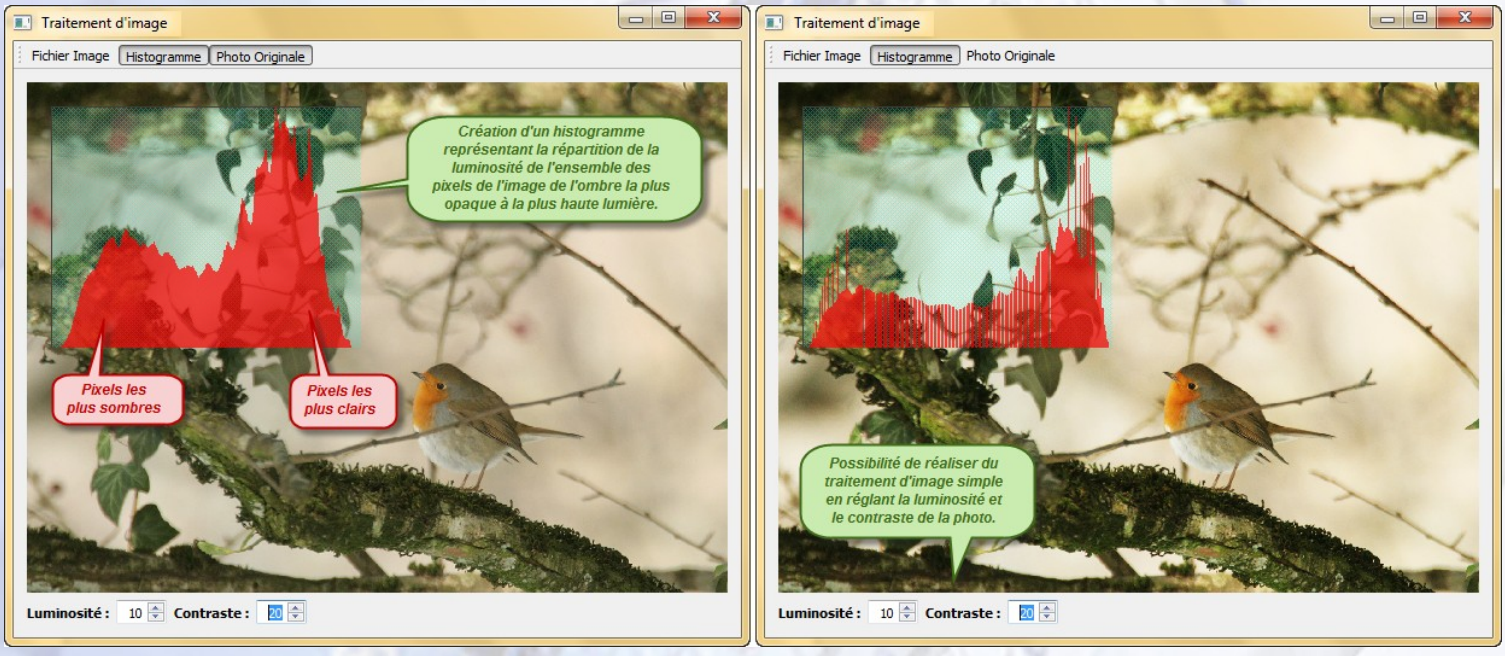
Lors de cette étude, nous allons compléter notre formation sur les images dans Qt Creator. Nous verrons également toutes les possibilités de tracé : les couleurs, les couleurs transparentes, les dégradés de couleurs, le choix du crayon avec les différents capuchons et jointures, le choix du pinceau avec les motifs associés, le choix des formes à tracé, etc.

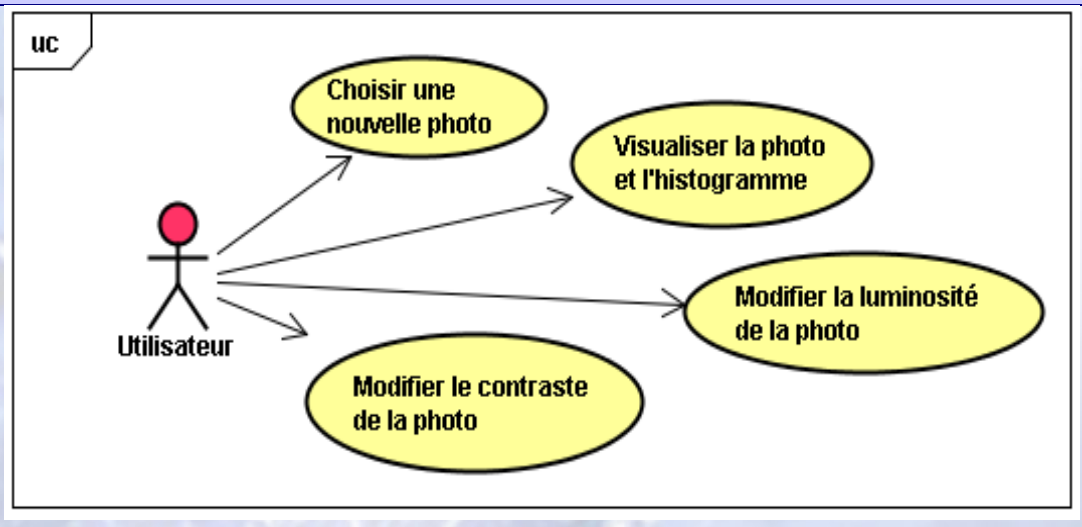
Dans le graphisme 2D, d'autres notions importantes sont également à prendre en compte, comme les transformations : les translations, les rotations, les changements d'échelle, les notions de vue relative avec les viewport, etc.

Pour bien évaluer ces différentes technologies, je vous propose de mettre en œuvre une application qui réalise un petit traitement de photo avec les fonctionnalités suivantes :

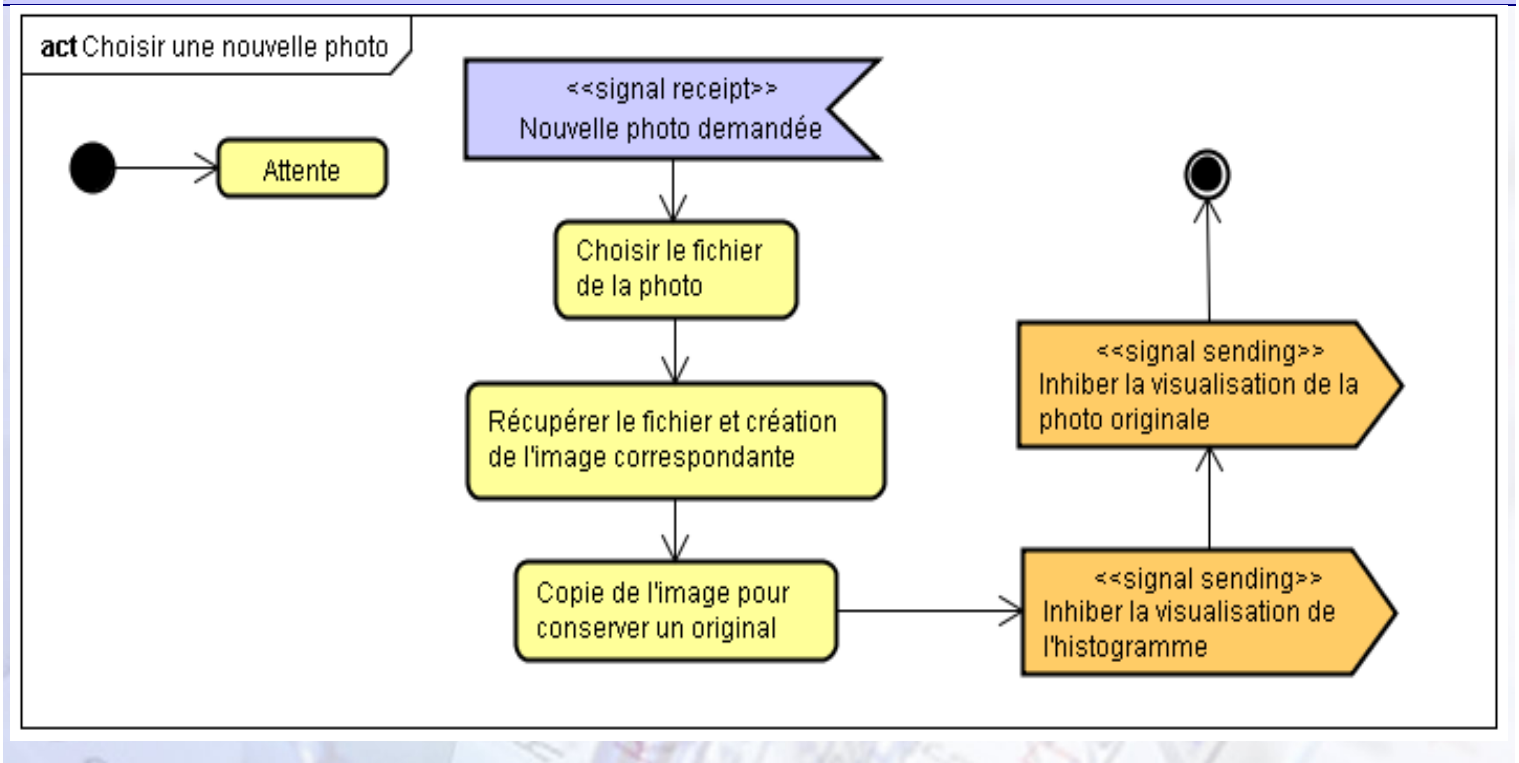
- ***Possibilité de visualiser à la demande l'histogramme correspondant à la photo choisie.***
- ***Possibilité de régler la luminosité de l'image sans destruction des ombres et des hautes lumières.***
- ***Possibilité de régler le contraste de l'image toujours sans destruction des ombres et des hautes lumières.***
- ***Possibilité, à tout moment, de visualisée la photo originale pour la confronter au traitement proposé.***



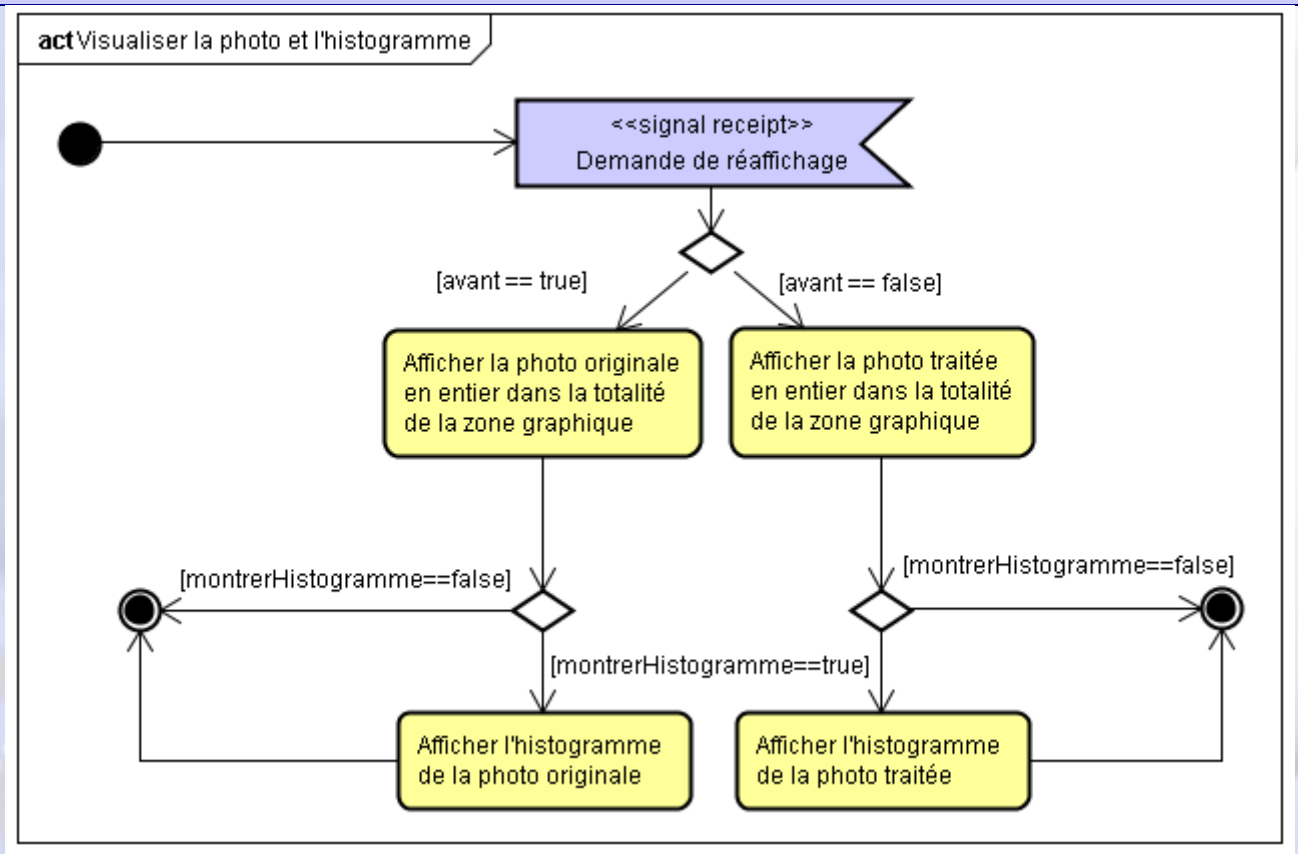
MODÉLISATION – DIAGRAMME DE CAS D'UTILISATION



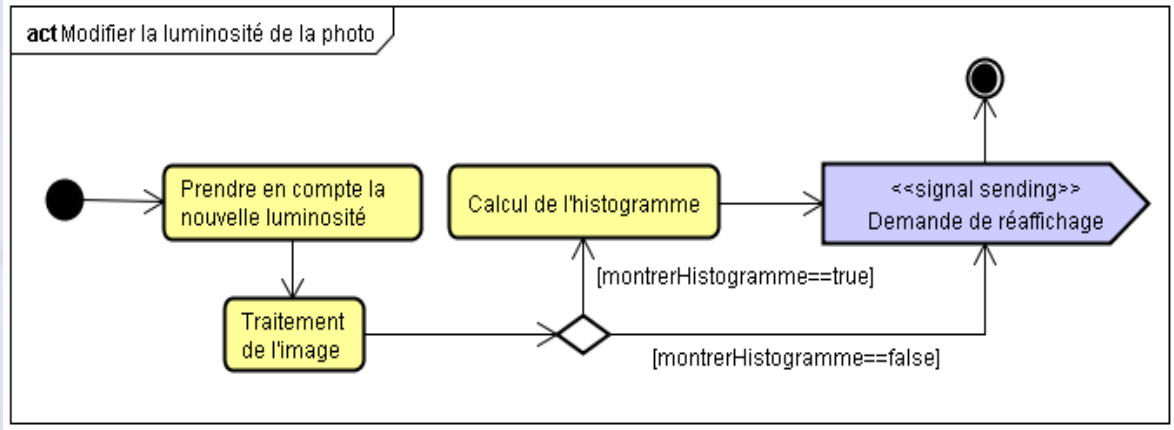
x MODÉLISATION – DIAGRAMME D'ACTIVITÉ CHOISIR UNE NOUVELLE PHOTO



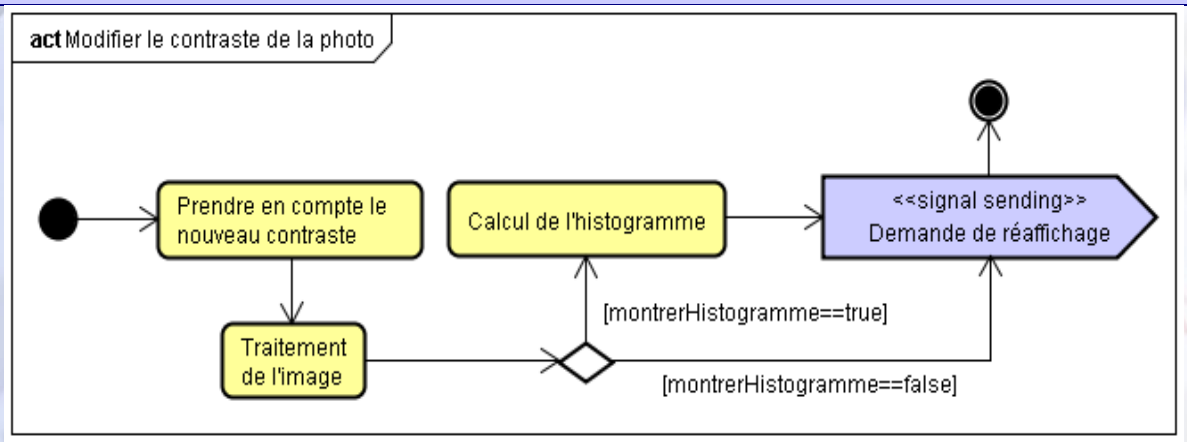
x MODÉLISATION – DIAGRAMME D'ACTIVITÉ DE VISUALISÉ LA PHOTO



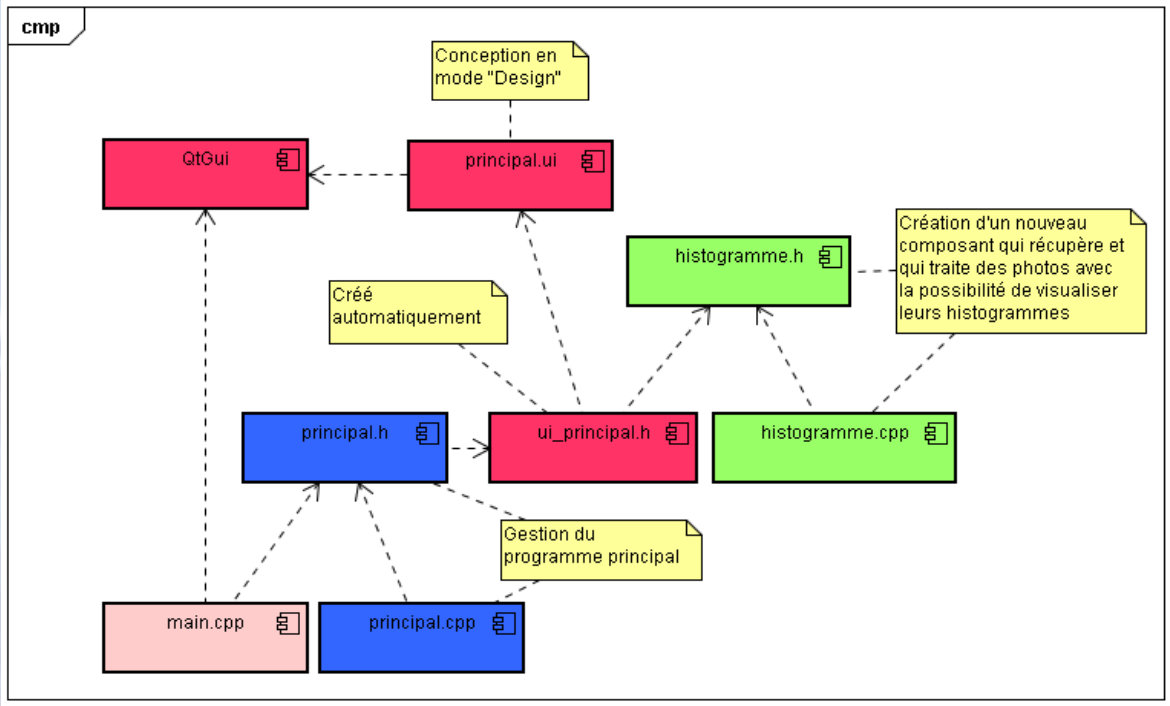
MODÉLISATION – DIAGRAMME D'ACTIVITÉ DE MODIFIER LA LUMINOSITÉ DE LA PHOTO



MODÉLISATION – DIAGRAMME D'ACTIVITÉ DE MODIFIER LE CONTRASTE DE LA PHOTO

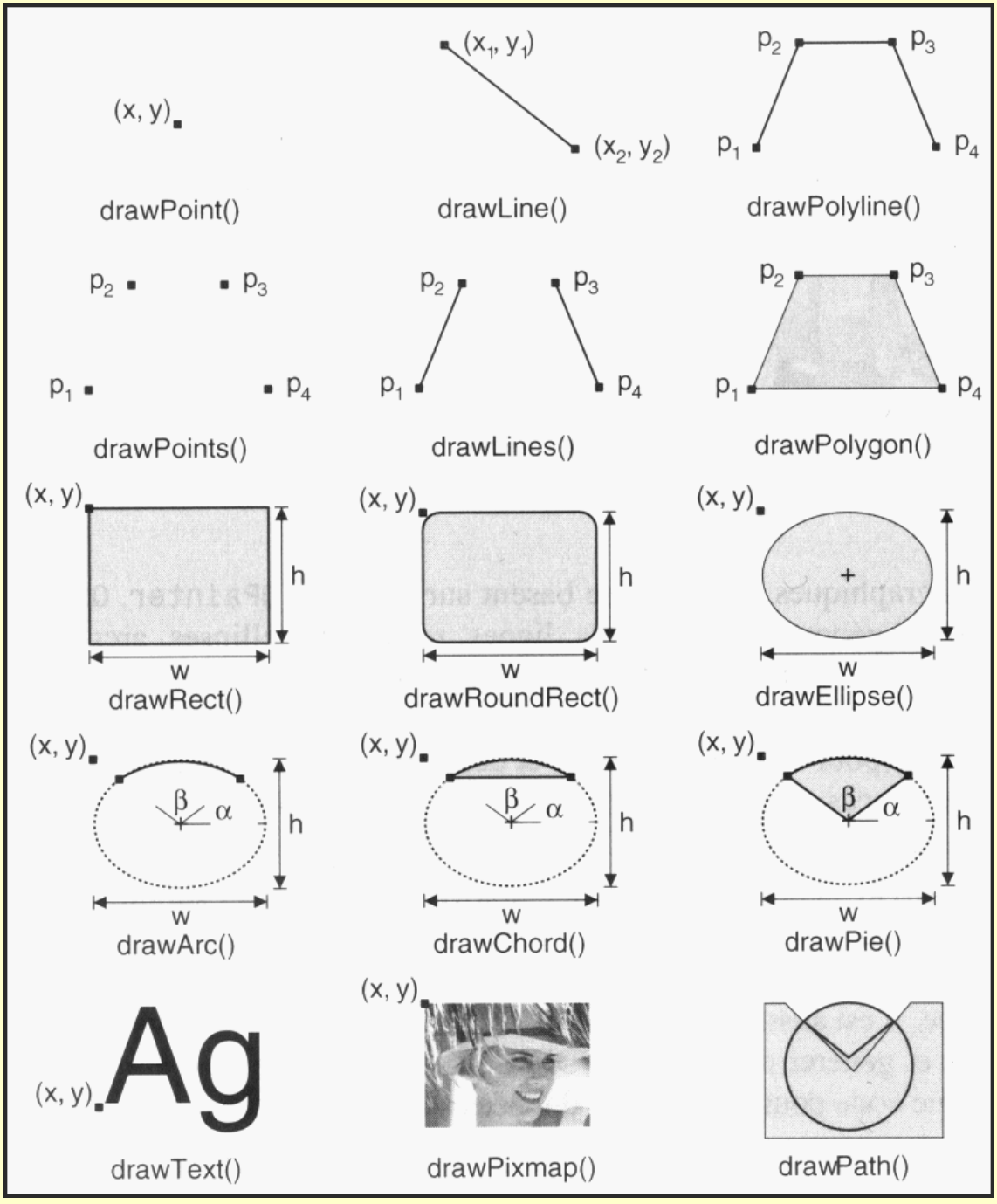


MODÉLISATION – DIAGRAMME DE COMPOSANTS



x GRAPHISME 2D

Les graphisme 2D de Qt se basent sur la classe **QPainter**. **QPainter** peut tracer des formes géométriques (points, lignes, rectangles, ellipses, arcs, cordes, segments, polygones et courbes de Bézier), de même que des objets pixmaps, des images et du texte.



*De plus, **QPainter** prend en charge des fonctionnalités avancées, telle que l'anticrénelage (les bords du texte et des formes s'adoucissent), le mélange alpha (la transparence), le remplissage dégradé et les tracés de vecteur.*

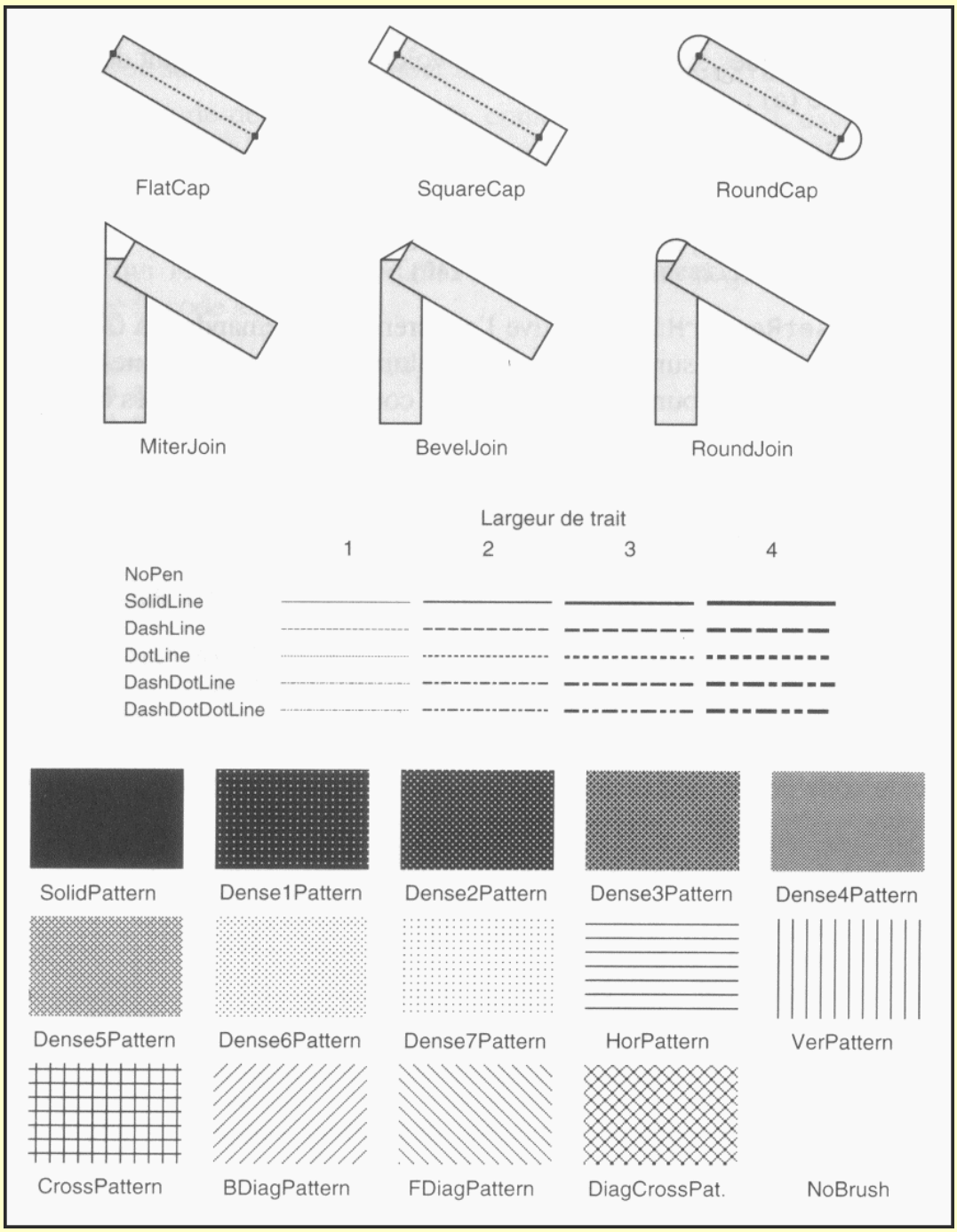
***QPainter** supporte aussi les transformations qui permettent de dessiner des graphismes 2D indépendants de la résolution. **QPainter** peut également être employé pour dessiner sur un « périphérique de dessin » tel qu'un **QWidget**, **QPixmap** ou **QImage**. C'est utile quand nous créons nos propres classe personnalisées avec leurs propres apparences.*

*Il est aussi possible d'utiliser **QPainter** en association avec **QPrinter** pour imprimer et générer des fichiers PDF. Cela signifie que nous pouvons souvent nous servir du même code pour afficher des données à l'écran et pour produire des rapports imprimés.*

LES OUTILS DE DESSIN

Les outils de **QPainter** influencent la façon de dessiner. Certains d'entre eux proviennent de périphériques, d'autres sont initialisés à leurs valeurs par défaut. Les trois principaux outils sont le crayon, le pinceau et la police :

- ***Le crayon est utilisé pour tracer des lignes et les contours des formes. Il est constitué d'une couleur, d'une largeur, d'un style de trait, de capuchon et de jointure.***
- ***Le pinceau permet de remplir des formes géométriques. Il est composé normalement d'une couleur d'un style, mais peut également appliquer une texture (un pixmap répété à l'infini) ou un dégradé.***
- ***La police est utilisée pour dessiner du texte. Une police possède de nombreux attributs, dont une famille et une taille.***

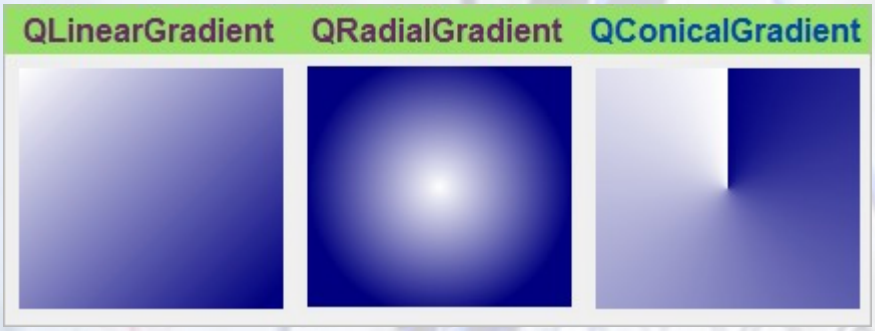


*Ces outils peuvent être modifiés à tout moment en faisant appel aux méthodes respectives **setPen()**, **setBrush()** et **setFont()** avec des objets correspondants **QPen**, **QBrush** ou **Qfont**.*

- ***dessin.setPen(QPen(Qt::red, 12, Qt::DashDotLine, Qt::RounCap)).***
- ***dessin.setBrush(QBrush(Qt::green, Qt::SolidPattern)).***

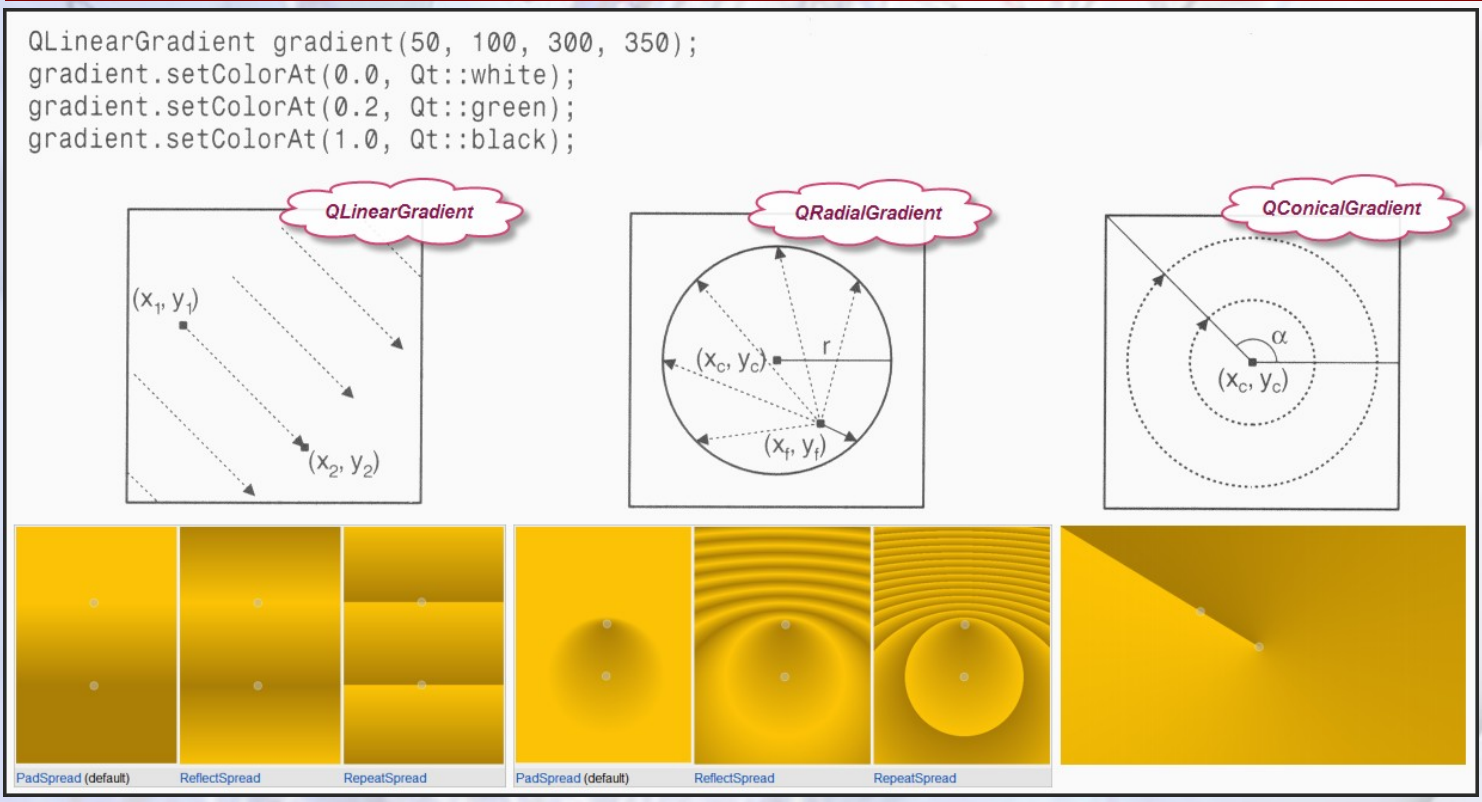
LES DÉGRADÉS

Les dégradés reposent sur une interpolation de couleur permettant d'obtenir des transitions homogènes entre deux ou plusieurs couleurs. Ils sont fréquemment utilisés pour produire des effets 3D. Qt prend en charge trois types de dégradés : linéaire, conique et circulaire.



- **Les dégradés linéaires sont définis par deux points de contrôle et par une série « d'arrêt de couleur » sur la ligne qui relie ces deux points. Attention, les positions sont indiquées comme des valeurs à virgule flottante entre 0 et 1, où 0 correspond au premier point de contrôle et 1 au second. Les couleurs situées entre les interruptions spécifiées sont interpolées. QLinearGradient(x_1, y_1, x_2, y_2)**
- **Les dégradés circulaires sont définis par un centre (x_c, y_c), un rayon r et une focale (x_f, y_f), en complément des interruptions de dégradé. Le centre et le rayon spécifient un cercle. Les couleurs se diffusent vers l'extérieur à partir de la focale, qui peut être le centre ou tout autre point dans le cercle. QRadialGradient(x_c, y_c, r, x_f, y_f)**
- **Les dégradés coniques sont définis par un centre (x_c, y_c), un angle a. Les couleurs se diffusent autour du point central comme la trajectoire de la petite aiguille d'une montre. QConicalGradient(x_c, y_c, a)**

```
QLinearGradient gradient(50, 100, 300, 350);
gradient.setColorAt(0.0, Qt::white);
gradient.setColorAt(0.2, Qt::green);
gradient.setColorAt(1.0, Qt::black);
```



*Nous pouvons contrôler la façon dont les dégradés se répandent (spread) au delà du point de départ et du point final au moyen de la méthode **setSpread()** en proposant l'une des constantes suivantes :*

- ***QGradient::PadSpread** (par défaut)*
- ***QGradient::ReflectSpread***
- ***QGradient::RepeadSpread***

x LES TRANSFORMATIONS

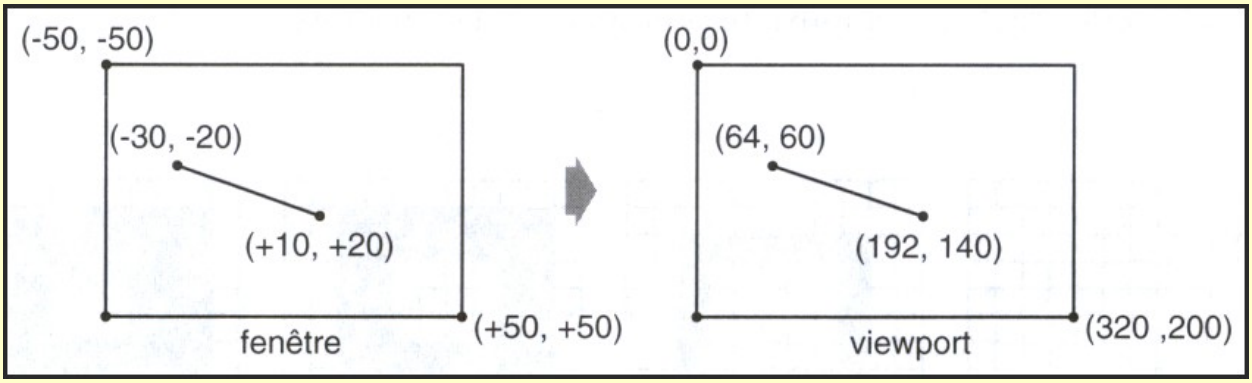
Avec le système de coordonnées par défaut du **QPainter**, le point (0, 0) se situe dans le coin supérieur gauche du périphérique de dessin ; les coordonnées x augmentent vers la droite et les coordonnées y sont orientées vers le bas. Par ailleurs, dans ce système, chaque pixel occupe une zone d'une taille de 1x1.

Il est possible de travailler avec deux systèmes de coordonnées simultanément :

- ***Le viewport est un rectangle spécifié en coordonnées physiques.***
- ***La fenêtre spécifie le même rectangle, mais en coordonnées logiques.***

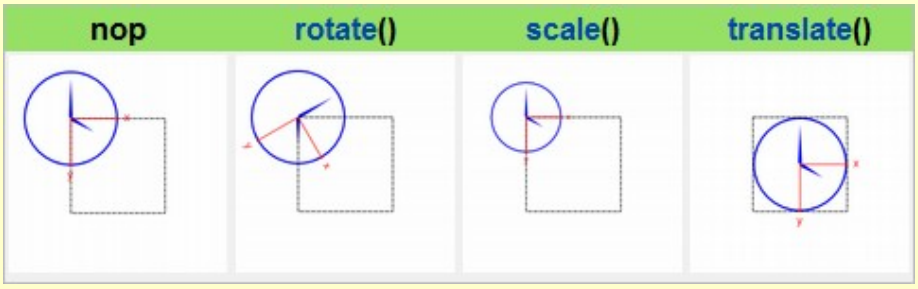
*Par défaut, le **viewport** et la **fenêtre** correspondent au rectangle du périphérique. Par exemple, si ce dernier est un composant graphique de 320x200, le **viewport** et la **fenêtre** représentent le même rectangle de 320x200 avec son point supérieur gauche à la position (0, 0). Dans ce cas, les systèmes de coordonnées logiques et physiques sont identiques.*

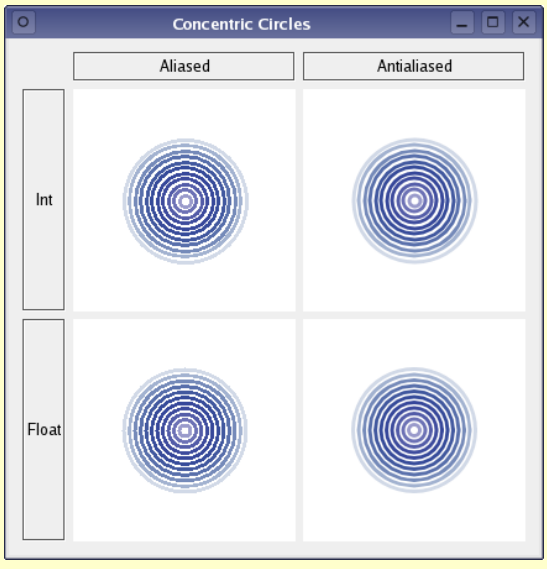
*Le mécanisme fenêtre-viewport est utile pour que le code de dessin soit indépendant de la taille ou de la résolution du périphérique de dessin. Par exemple, si nous voulons que les coordonnées logiques s'étendent de (-50, -50) à (+50, +50) avec (0, 0) au milieu, nous pouvons configurer la fenêtre comme suit : **dessin.setWindow(-50, -50, 100, 100)**.*



Il est également possible de proposer des transformations qui s'appliquent en plus de la conversion **fenêtre-viewport**. Par ailleurs, à tout moment, vous pouvez mettre en place l'anticrénelage.

*Elles nous permettent de translater, de mettre à l'échelle, de pivoter et de faire glisser les éléments (déformation, perspective) que nous dessinons, au moyen respectivement des méthodes **translate()**, **scale()**, **rotate()** et **shear()**.*





*L'appel de **setRenderHint()** active l'anticrénelage, demandant à **QPainter** d'utiliser diverses intensités de couleur sur les bords pour réduire la distorsion visuelle qui se produit habituellement quand les contours d'une forme sont convertis en pixels. Les bords sont donc plus homogènes sur les plates-formes et les périphériques qui prennent en charge cette fonctionnalité.*

***dessin.setRenderHint(QPainter::Antialiasing, true)**.*

LES IMAGES ET LES SYSTÈMES DE COULEUR

La classe **QImage** stocke une image indépendamment du matériel. Elle peut être définie avec une qualité de 1, 8 ou 32 bits. Une image avec une qualité de 32 bits utilise 8 bits pour chaque composante rouge, vert et bleu d'un pixel. Les 8 bits restants stockent le canal alpha du pixel (niveau de transparence).

Par exemple, les composants rouge, vert, bleu et alpha d'une couleur rouge pure présentent les valeurs 255, 0, 0, 255. Dans Qt, cette couleur peut être spécifiée de deux manières différentes :

```
QRgb rouge = qRgba(255, 0, 0, 255) ;
QRgb rouge = qRgb(255, 0, 0) ; // la couleur étant parfaitement opaque.
```

***QRgb** est simplement une redéfinition de type correspondant à un **unsigned int**. **qRgba()** et **qRgb()** sont par contre des fonctions (en ligne) qui combinent leurs arguments en une valeur entière équivalente de 32 bits, traduisible par exemple par la classe **QImage**. Il est finalement aussi possible d'écrire :*

```
QRgb rouge = 0xFFFF0000 ; // la premier FF correspond au canal alpha et le second FF à la composante rouge.
```

*Qt propose deux types permettant de stocker les couleurs : **QRgb** et **QColor**. Alors que **QRgb** est finalement un simple entier non signé employé par **QImage** pour stocker les données 32 bits du pixel, **QColor** est une classe dotée de nombreuses fonctions pratiques qui est souvent utilisée dans Qt pour stocker des couleurs.*

```
Il existe d'autres fonctions bien utiles pour contrôler chacune des composantes d'une couleur, respectivement qRed(), qGreen(), qBlue() et qAlpha().
Il existe même la méthode qGray() qui renvoie le niveau de gris sous forme d'octet.
```

La classe **QImage** est capable de gérer un nombre considérable de type de fichiers d'image. Il ne faut pas oublier que le seul format d'image qui peut être exploité dans la mémoire est le format bitmap qui permet d'accéder à chacune des composantes du pixel, mais qui prend du coup énormément de place.

*Les fichiers images sont donc systématiquement compressés avec des algorithmes différents suivant le type de compression utilisé. Lorsque nous ouvrons une photo à partir de la classe **QImage**, elle s'occupe automatiquement de décompresser cette dernière pour aboutir au bitmap équivalent. De la même façon, la classe **QImage** compresse votre bitmap, suivant le format désiré, pour le stocker sur le disque dur de votre système.*

*Comme nous l'avons déjà découvert lors de l'étude précédente, il est possible de récupérer directement une image à partir d'un fichier à l'aide de la méthode **load()** de **QImage**.*

*Nous pouvons également créer une image de toute pièce en la remplissant ensuite par les couleurs que vous voulez pour chacun des pixels ou en faisant une copie partielle (ou pas) d'une autre image. Lorsque vous créez une image, vous devez alors préciser les dimensions requises et le type de format d'image, correspondant à une énumération de **QImage**, dont voici quelques éléments :*

Format	Description	Qt's support
BMP	Windows Bitmap	Read/write
GIF	Graphic Interchange Format (optional)	Read
JPG	Joint Photographic Experts Group	Read/write
JPEG	Joint Photographic Experts Group	Read/write
PNG	Portable Network Graphics	Read/write
PBM	Portable Bitmap	Read
PGM	Portable Graymap	Read
PPM	Portable Pixmap	Read/write
TIFF	Tagged Image File Format	Read/write
XBM	X11 Bitmap	Read/write
XPM	X11 Pixmap	Read/write

Format d'image	Description
QImage::Format_Mono	Noir et blanc, un seul bit par pixel
QImage::Format_Indexed8	8 bits par pixel
QImage::Format_RGB32	32 bits par pixel, 1 octet pour chacune les canaux rouge, vert, bleu sans le canal alpha
QImage::Format_ARGB32	32 bits par pixels, 1 octet pour toutes les canaux rouge, vert, bleu et alpha
QImage::Format_RGB16	16 bits par pixels (5-6-5) bits respectivement pour les canaux rouge, vers, bleu

Ainsi, par exemple, pour construite une nouvelle image de 320x200 en 32 bits, prenant en compte tous les canaux rouge vert bleu avec la transparence, voici le code à écrire :

```
QImage image = QImage(320, 200, Qimage::Format_ARGB32);
```

Ou, plus directement :

```
QImage image(320, 200, Qimage::Format_ARGB32);
```

*Attention, les pixels de cette image sont totalement aléatoire, c'est ce qui se trouve actuellement dans la mémoire. Il est donc nécessaire de remplir toute la surface de l'image avec un la couleur de fond que vous désirez à l'aide de la méthode **fill()**. Ainsi, si vous désirez une image totalement transparente, voici ce que vous devez écrire :*

```
image.fill(qRgba(0, 0, 0, 0));
```

LES MÉTHODES UTILES DE QIMAGE

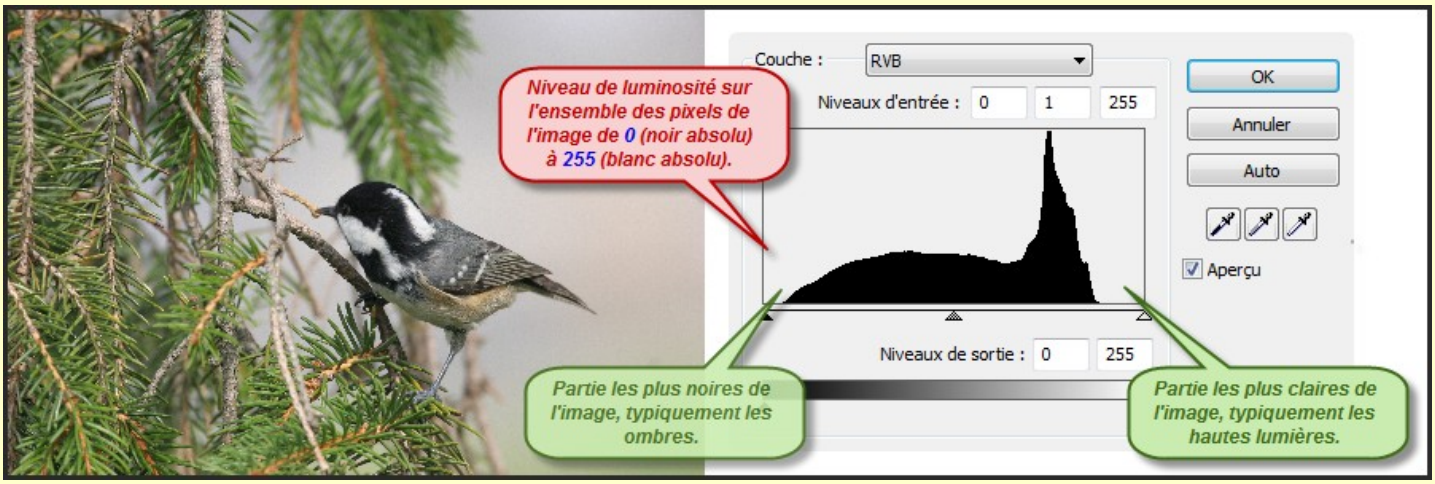
Voici ci-dessous quelques méthodes bien utiles pour manipuler les images et faire des traitements adaptés. Utilisez l'aide de Qt Creator pour savoir comment manipuler correctement ces différentes méthodes.

- ***byteCount() : renvoie le poids de l'image, le nombre d'octet.***
- ***convertFormat(nouveau format) : permet de changer de format d'image.***
- ***copy(x, y, largeur, hauteur) : prend une portion de l'image (ou la totalité).***
- ***copy(rectangle) : prend une portion de l'image (ou la totalité).***
- ***createHeuristicMask() : création d'un masque sous forme d'image monochrome. Cette méthode analyse les bords de l'image et propose un fond blanc jusqu'à ce qu'elle détecte un contraste prononcé. Le reste de l'image est alors en noir.***
- ***fill(couleur) : remplie toute l'image avec la couleur spécifiée en argument.***
- ***height() : largeur en pixel de l'image.***
- ***invertPixels() : négatif de l'image.***
- ***load(nom du fichier) : charge la totalité de l'image à partir du nom du fichier.***
- ***mirrored(horizontal, vertical) : propose des symétries d'image comme un miroir suivant l'axe horizontal et/ou l'axe vertical.***
- ***pixel(x, y) : détermine la couleur du pixel sélectionné.***
- ***rect() : renvoie les dimensions de toute la zone de l'image.***
- ***save(nom du fichier, «JPG », 85) : sauvegarde le bitmap correspondant à l'image dans le fichier sélectionné dans le type de compression souhaité, en précisant si nécessaire le taux de compression.***
- ***save(nom du fichier, «PNG ») : autre exemple.***
- ***scaled(largeur, hauteur, [aspect, type d'interpolation]) : changement d'échelle de l'image par interpolation en précisant les nouvelles dimensions. Il est possible de conserver le ratio de l'image sans avoir de déformation si vous le souhaitez.***
- ***scaledToHeight(hauteur, [type d'interpolation]) : changement d'échelle de l'image par interpolation en précisant que la hauteur. Le ratio de l'image est conservé.***
- ***scaledToWidth(largeur, [type d'interpolation]) : changement d'échelle de l'image par interpolation en précisant que la largeur. Le ratio de l'image est conservé.***
- ***setPixel(x, y, couleur) : propose une nouvelle couleur au pixel sélectionné.***
- ***size() : renvoie le poids de l'image, le nombre d'octet.***

L'HISTOGRAMME, LES COURBES AVEC LES TRAITEMENTS ADAPTÉES

L'histogramme représente, sous forme graphique, la luminosité de l'ensemble des pixels d'une image. Elle nous montre ainsi l'homogénéité de l'éclairage de votre photo.

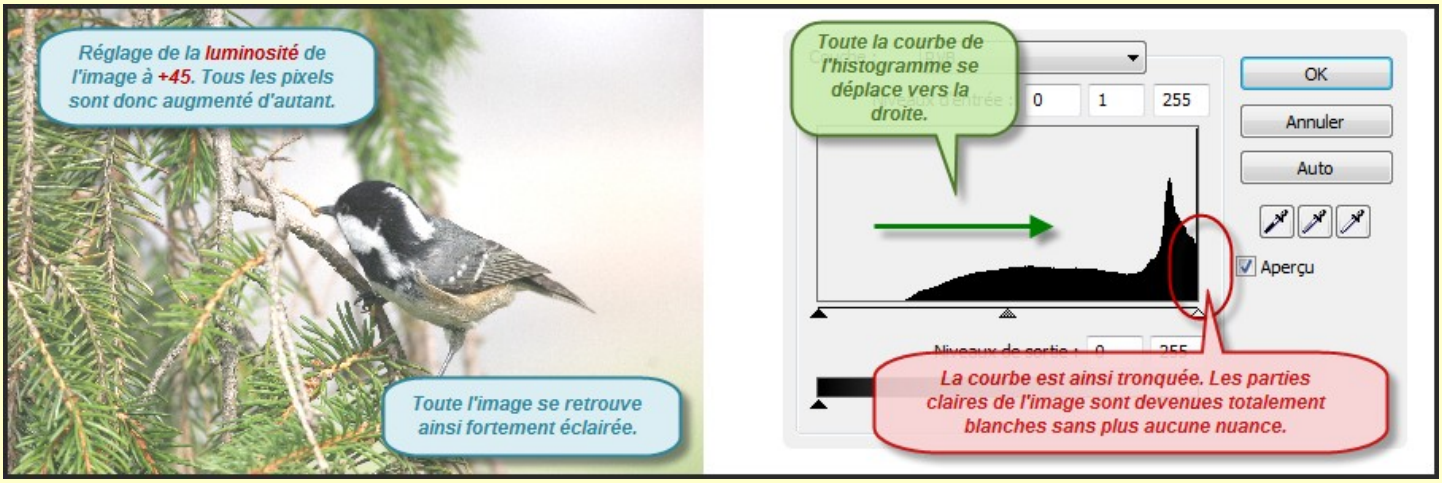
L'axe des abscisses de l'histogramme va de 0 à 255 (octet) indiquant ainsi le nombre de pixels correspondant respectivement aux valeurs les plus sombres jusqu'aux valeurs les plus claires en passant par la luminosité moyenne de l'image.



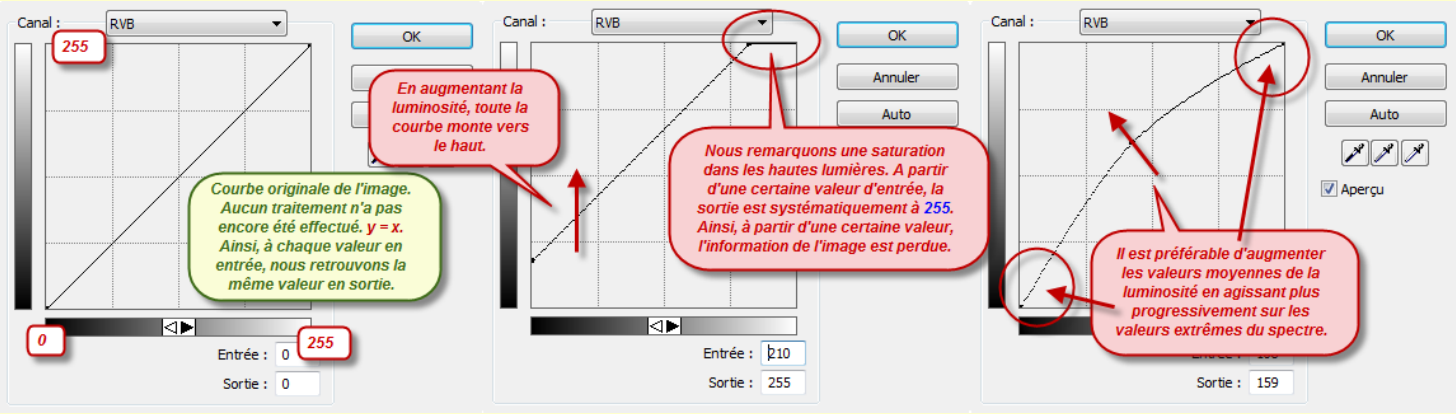
L'idéal, c'est d'avoir un histogramme qui s'étale au maximum et qui offre ainsi une image contrastée. Dans l'exemple ci-dessus, notre image est relativement fade. Effectivement, nous remarquons dans l'histogramme que les lumières hautes ainsi que les ombres ne sont pas très bien représentées. Nous remarquons également un pic important sur une partie assez claire. Cela correspond au fond gris lumineux sur la droite de la photo.

Nous pouvons faire un premier traitement sur notre image de telle sorte que nous puissions mieux voir la tête de l'oiseau.

*Nous pouvons agir sur la **luminosité** de l'image qui consiste à augmenter ou diminuer chaque pixel d'une unité de valeur. Par exemple, pour mieux voir la tête de l'oiseau, je décide d'augmenter la luminosité globale de l'image de **+45**.*

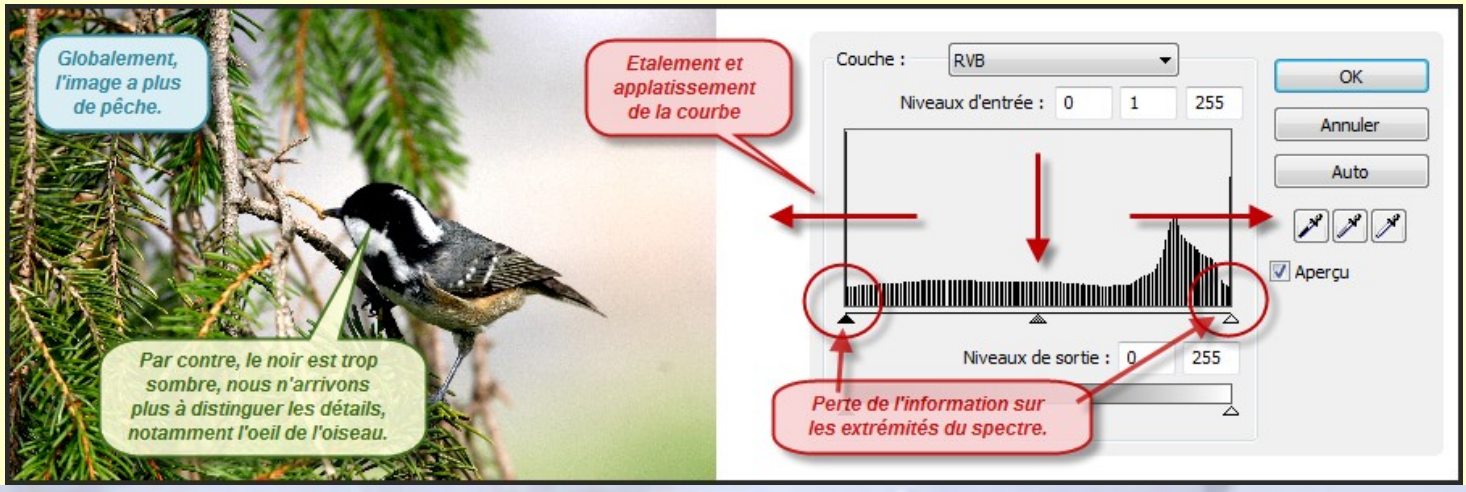


Nous remarquons tout de suite que la photo s'est fortement éclairée. Toutefois, en utilisant cette technique, une partie de l'image devient totalement blanche sans plus aucune nuance dans les teintes. Une partie de l'information est ainsi perdue. Ce type de traitement est à proscrire. Il serait plutôt souhaitable d'augmenter les valeurs moyennes de luminosité sans toucher aux extrémités du spectre, du moins le faire plus progressivement.

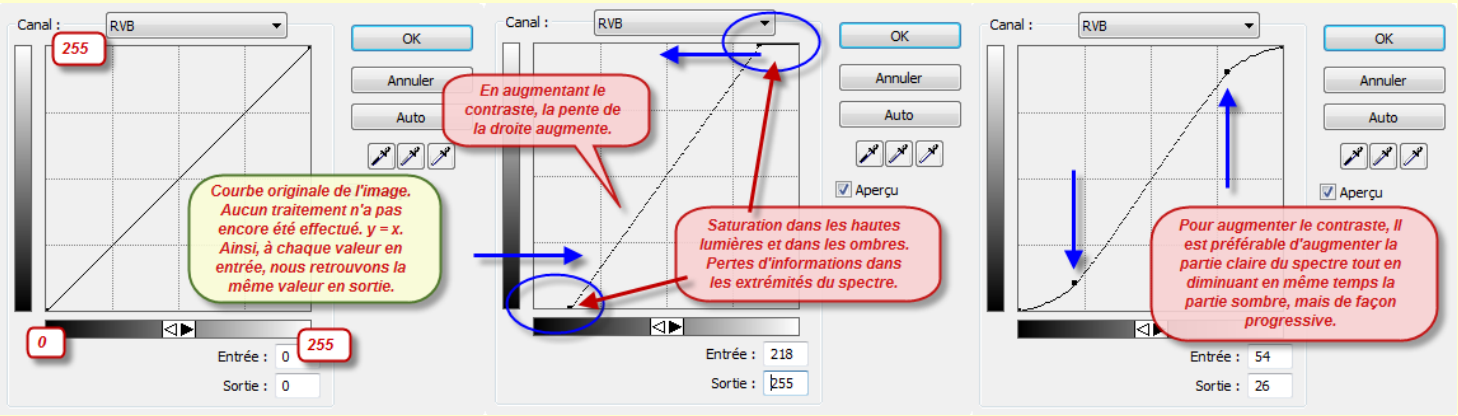


Le deuxième type de traitement souvent utilisé est le réglage du **contraste**. Cette fois-ci, dans le cas où nous augmentons le contraste, les pixels lumineux deviennent encore plus lumineux, et les pixels sombres deviennent encore plus sombres.

*Là aussi, l'utilisation de l'outil **contraste** est rarement judicieux. Une partie du spectre risque d'être tronquée.*



Cette fois-ci, l'idéal est d'agir directement sur la courbe, comme cela vous est montré ci-dessous :



Toutefois, lorsque l'histogramme est relativement étriqué, comme c'est un petit peu le cas pour notre image de base, nous pouvons utiliser la technique centrale, comme dans l'exemple ci-dessus. Nous obtenons du coup une pente plus prononcée qui permet d'élargir le spectre, afin d'obtenir finalement une image avec plus de pêche. Le tout, c'est que l'histogramme reste à l'intérieur des limites extrêmes.

x Dans notre programme de traitement d'image, nous allons fabriquer un algorithme qui tient compte de ces particularités, en faisant en sorte que les réglages de la luminosité et du contraste se fasse progressivement comme nous venons de le découvrir.

x HISTOGRAMME

```
#ifndef HISTOGRAMME_H
#define HISTOGRAMME_H

#include <QWidget>

class Histogramme : public QWidget
{
  Q_OBJECT
public:
  Histogramme(QWidget *parent);
protected:
  void paintEvent(QPaintEvent *evt);
private slots:
  void changerPhoto();
  void activerHistogramme(bool choix);
  void changerLuminosite(int valeur);
  void changerContraste(int valeur);
  void choisirOriginal(bool choix);
signals:
  void raz(int);
  void raz(bool);
private:
  void calculHistogramme();
  void traitement();
private:
  QImage photo, original, histogramme;
  bool montrerHistogramme, avant;
  int luminosite, contraste;
};

#endif // HISTOGRAMME_H
```

Rafraîchissement de la photo suivant les traitements effectués avec la possibilité de montrer en même temps l'histogramme.

récupération de la nouvelle photo avec remise à zéro de tous les réglages

Visualisation ou pas de l'histogramme.

Prises en compte de nouveaux réglages pour le traitement de l'image.

Choix entre l'affichage de l'image originale ou de l'image traitée.

Envoi de signaux pour mettre tous les réglages de l'IHM à zéro.

Calcul de l'histogramme et réalisation du traitement souhaité.

Différents attributs qui mémorisent la photo orginale, la photo traitée, l'histogramme ainsi que les valeurs de luminosité et de contraste choisis.

```
#include "histogramme.h"
#include <QtGui>
#include <math.h>

Histogramme::Histogramme(QWidget *parent) : QWidget(parent)
{
  histogramme = QImage(257, 200, QImage::Format_ARGB32);
  montrerHistogramme = avant = false;
  luminosite = contraste = 0;
}

void Histogramme::paintEvent(QPaintEvent *evt)
{
  if (!photo.isNull()) {
    QPainter dessin(this);
    QImage image = avant ? original.scaledToWidth(rect().width()) : photo.scaledToWidth(rect().width());
    dessin.drawImage(rect(), image, rect());
    if (montrerHistogramme) dessin.drawImage(QPoint(20, 20), histogramme);
  }
}

void Histogramme::activerHistogramme(bool choix)
{
  if (montrerHistogramme = choix) calculHistogramme();
  update();
}

void Histogramme::changerPhoto()
{
  QString nomFichier = QFileDialog::getOpenFileName(this, "Choix de votre photo", "", "Photos (*.gif *.png *.jpg)");
  if (!nomFichier.isEmpty())
  {
    photo.load(nomFichier);
    original = photo;
    raz(0);
    raz(false);
    update();
  }
}

void Histogramme::calculHistogramme()
{
  QImage &image = avant ? original : photo;
  int rouge[256]={}, vert[256]={}, bleu[256]={};
  int lumiere[256], maximum = 0;
  for (int ligne=0; ligne < image.height(); ligne++)
    for (int colonne=0; colonne < image.width(); colonne++) {
      rouge[qRed(image.pixel(colonne, ligne))]++;
      vert[qGreen(image.pixel(colonne, ligne))]++;
      bleu[qBlue(image.pixel(colonne, ligne))]++;
    }
  for (int i=0; i<256; i++) {
    lumiere[i] = (rouge[i]+vert[i]+bleu[i]) / 3;
    if (lumiere[i] > maximum) maximum=lumiere[i];
  }
  for (int i=0; i<256; i++) lumiere[i] = lumiere[i] * 200 / maximum;
  histogramme.fill(qRgba(0, 0, 0, 0));
  QPainter dessin(&histogramme);
  dessin.setBrush(QBrush(QColor(0, 255, 255, 90), Qt::Dense5Pattern));
  dessin.setPen(QColor(64, 64, 64, 255));
  dessin.drawRect(histogramme.rect());
  dessin.translate(1, 200);
  dessin.scale(1, -1);
  dessin.setPen(QColor(255, 0, 0, 180));
  for (unsigned i=1; i<257; i++) dessin.drawLine(i, 0, i, lumiere[i]);
}

void Histogramme::traitement()
{
  int table[256];
  for (int i=0; i<256; i++) table[i] =  i + luminosite*sin(i*M_PI/256) - contraste*sin(2*i*M_PI/256);
  for (int i=0; i<256; i++)
    if (table[i]>255) table[i] = 255;
    else if (table[i]<0) table[i] = 0;
  for (int ligne=0; ligne < photo.height(); ligne++)
    for (int colonne=0; colonne < photo.width(); colonne++) {
      int rouge = table[qRed(original.pixel(colonne, ligne))];
      int vert = table[qGreen(original.pixel(colonne, ligne))];
      int bleu = table[qBlue(original.pixel(colonne, ligne))];
      photo.setPixel(colonne, ligne, qRgb(rouge, vert, bleu));
    }
}

void Histogramme::changerLuminosite(int valeur)
{
  luminosite = valeur;
  traitement();
  if (montrerHistogramme) calculHistogramme();
  update();
}

void Histogramme::changerContraste(int valeur)
{
  contraste = valeur;
  traitement();
  if (montrerHistogramme) calculHistogramme();
  update();
}

void Histogramme::choisirOriginal(bool choix)
{
  avant = choix;
  if (montrerHistogramme) calculHistogramme();
  update();
}
```

Création de l'image correspondant à l'histogramme en intégrant la transparence.

Lorsque nous chargeons une nouvelle image, tous les réglages sont remis à zéro dans l'IHM.

calcul de l'histogramme, soit de la photo originale, soit de la photo traitée.

Tableaux des 256 valeurs possibles pour chacune des composantes.

Parcours de la totalité de l'image, pour chaque pixel, en enregistrant les luminosités de chacune des composante dans l'ensemble du spectre.

calcul de la valeur moyenne de la lumière par rapport aux trois composantes fondamentales, dans l'ensemble du spectre et recherche de l'intensité maximale atteinte dans le spectre.

Changement d'échelle pour la lumière pour être en corélation avec l'image de l'histogramme correspondante.

Tracé du fond de l'histogramme en relief (ombre) avec un motif représentant le verre.

Changement d'axe et d'orientation pour que le tracé s'effectue de façon plus conventionnelle.

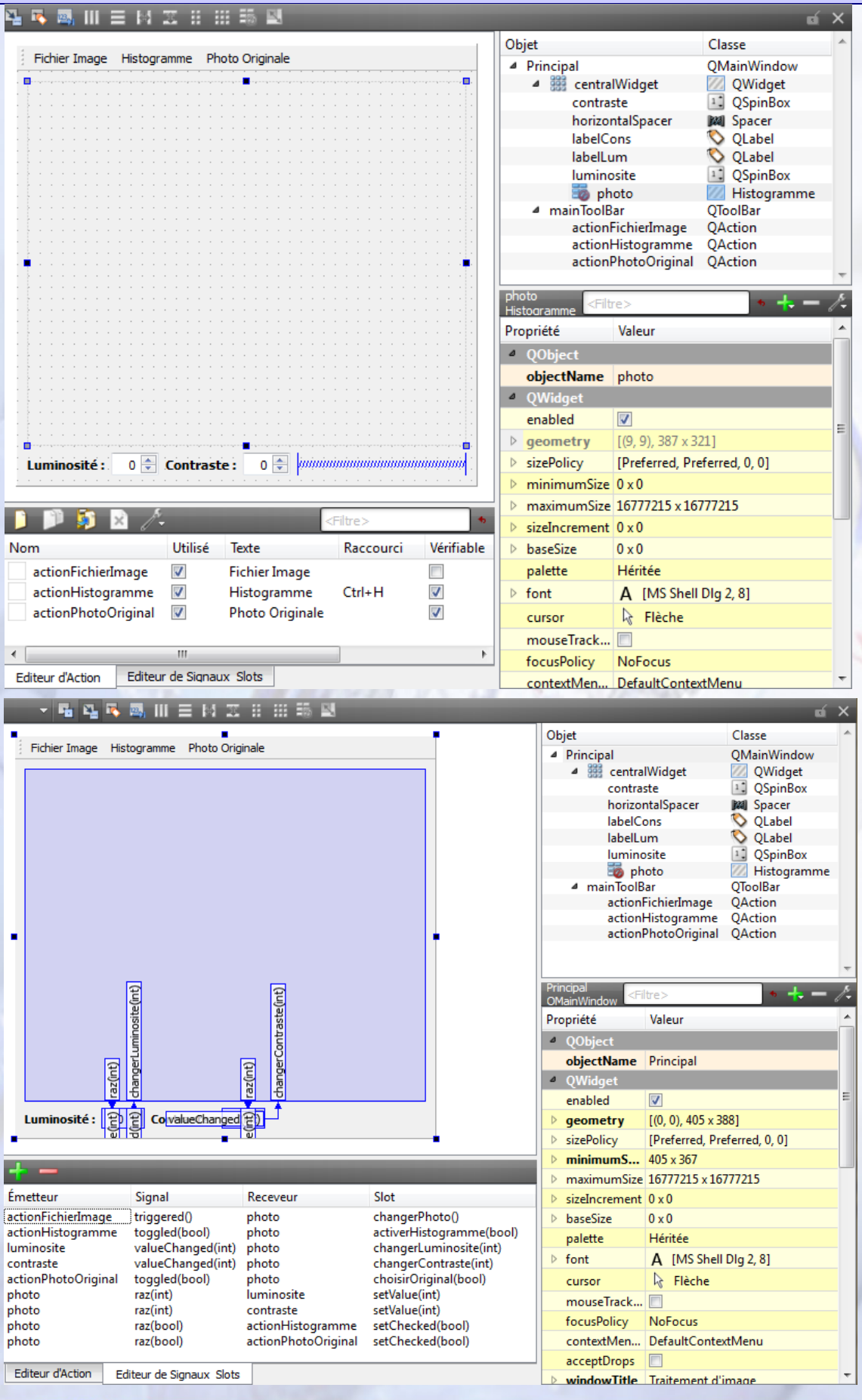
Tracé de l'histogramme.

Calcul de la courbe mathématique à appliquer afin de tenir compte de la luminosité et du contraste.

Mise en place d'un système de saturation dans le cas où les valeurs dépasse les capacités prévues.

Réglage de chaque pixel de l'image en tenant compte de la courbe mathématique calculée.

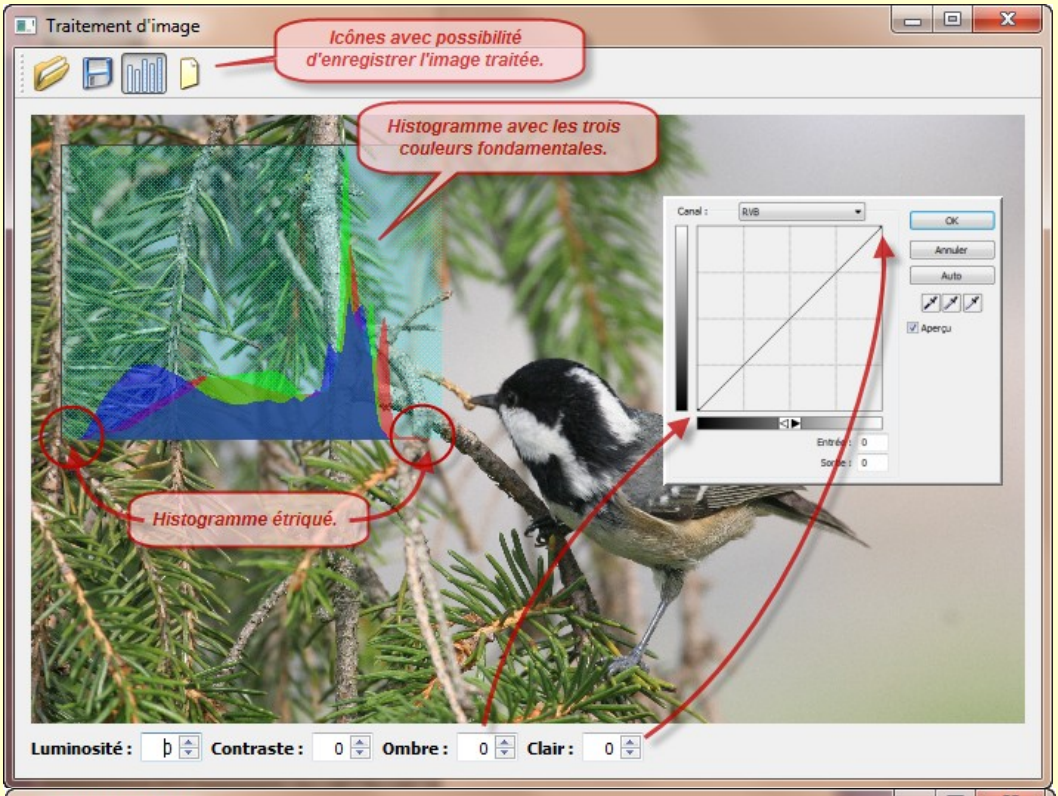
x PRINCIPAL.UI

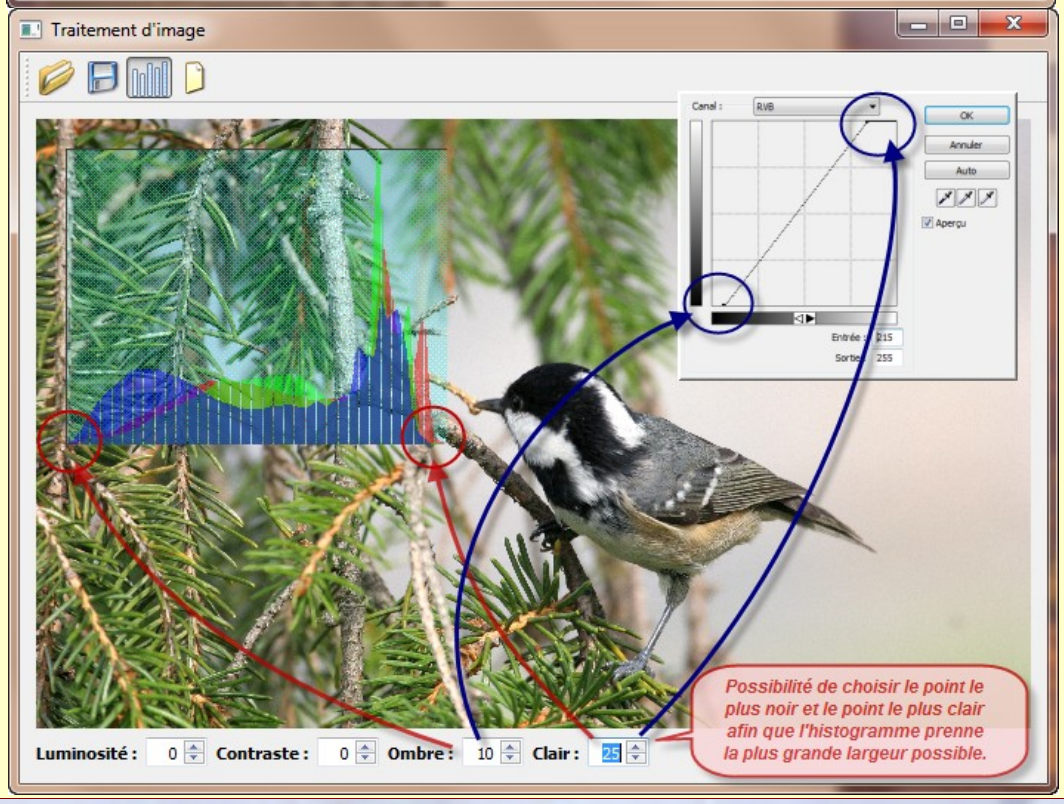


x TRAVAUX PRATIQUES EN AUTONOMIE

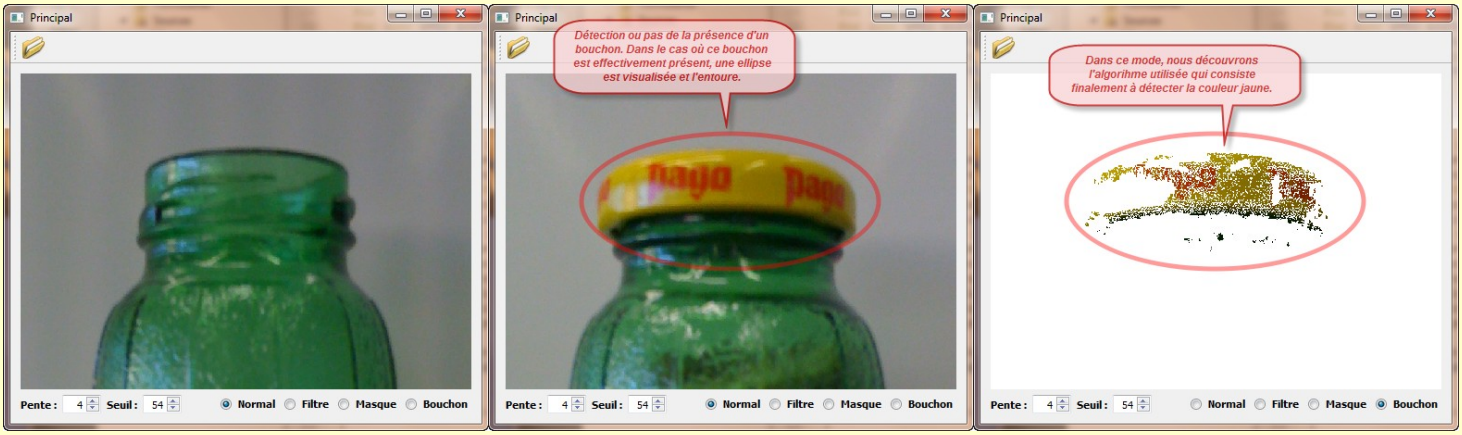
Je vous propose de faire de nouveaux projets spécifiques au traitements d'image en exploitant les différentes compétences de la classe **QImage**.

x Le premier TP consiste à faire la suite du projet réalisé en cours en rajoutant quelques fonctionnalités supplémentaires. Tout d'abord, proposez des icônes sur les actions. Rajoutez une nouvelle action qui permettra d'enregistrer la photo traitée. Par ailleurs, l'histogramme visualise cette fois-ci les trois couches fondamentales. Pour finir, vous rajouterez deux réglages supplémentaires pour élargir les histogrammes un peu étriqués afin d'obtenir des images plus contrastées.





- *Le deuxième TP nous permet de rentrer dans les systèmes de reconnaissance de formes avec en même temps la possibilité de détecter si une pièce est présente comme le bouchon d'une bouteille.*



- *Pour visualiser le contour d'une pièce, l'idéal est de travailler avec un masque qui finalement travaille avec des couleurs monochromes. Toutefois, pour que l'analyse se passe dans de bonnes condition, il est nécessaire de filtrer l'image originale afin de lui proposer un contraste très élevé avec un taux de luminosité également très élevé.*
- *Suivant les cas de figures, il est préférable de prévoir des réglages (pente et seuil) qui tiennent compte des conditions de prise de vue. Lorsque vous travailler plus spécialement sur les reconnaissances de forme, il n'est pas nécessaire de travailler en mode RGB, mais plutôt avec des colorations plus modeste où l'ensemble des couleurs est codée sur 8 bits. Cela permet de réaliser des traitements d'image beaucoup plus rapidement.*