



Lors de cette étude, nous allons revenir sur le graphisme 2D en nous intéressant cette fois-ci plus particulièrement sur l'aspect vectoriel associée à la notion de calque.

Jusqu'à présent, lorsque nous faisons du tracé personnalisé, nous devons systématiquement créer un nouveau composant, un **QWidget**, sur lequel nous redéfinissons la méthode **paintEvent()**. C'est à l'intérieur de cette dernière que nous élaborons nos différents tracés spécifiques.

Depuis les toutes dernières versions de Qt, pour faire du tracé personnalisé, il est maintenant possible de ne plus créer de composant spécifique. Vous créez alors les différents graphismes voulus sous forme vectorielle à l'intérieur d'une scène (équivalente finalement à un calque), le tout étant ensuite rendu visible par un composant qui ne s'occupe que de l'affichage.

Cette plateforme vectorielle est donc construite autour de trois éléments principaux : la vue, la scène et l'ensemble des vecteurs (les formes) :

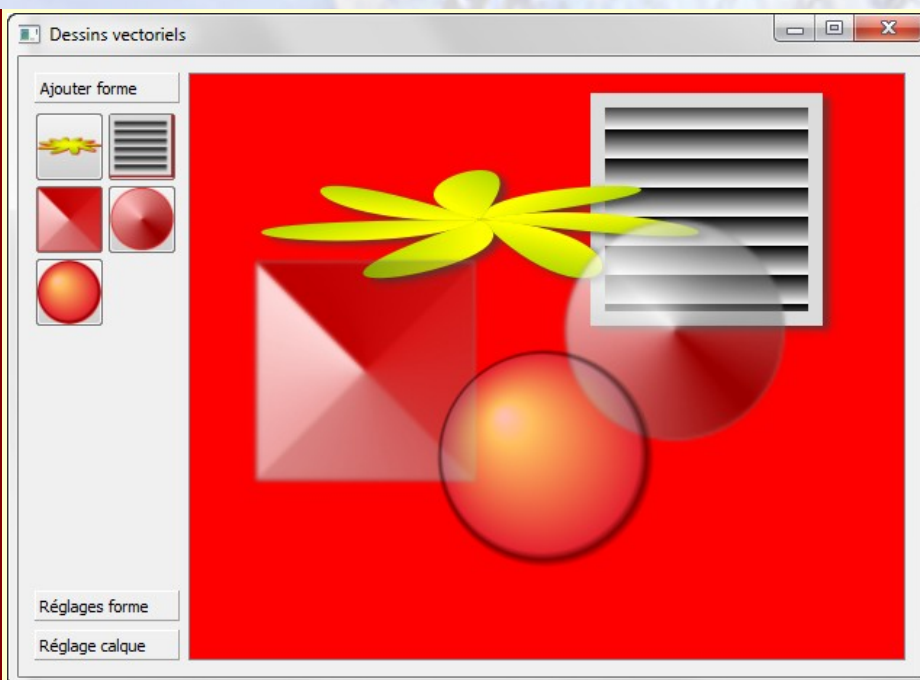
- x La vue est représentée par la classe **QGraphicsView** qui est un **QWidget** qui a pour simple but d'afficher tout le contenu de la scène.
- x La scène, représentée par la classe **QGraphicsScene**, sorte de calque qui contient l'ensemble des éléments graphiques, est capable également de gérer tous les événements associés à ces formes. La scène ne peut être affichée directement. Vous êtes obligé de passer par un **QGraphicsView**. Nous sommes là en présence d'un modèle **MVC (Modèle-Vue-Contrôleur)**. L'intérêt de cette structure, c'est qu'il est possible d'avoir plusieurs aspects pour un même calque avec par exemple des zooms ou des points de vues différents.
- x Chaque vecteur est ensuite interprété par une sous-classe de **QGraphicsItem** qui représente une simple forme élémentaire ou éventuellement un groupe de formes.

L'idée générale de cette approche est de créer tout d'abord l'ensemble des formes plus ou moins complexes qui vous intéressent, de les placer ensuite sur un calque et de visualiser le tout au travers du composant qui gère la vue.

- x Le gros intérêt de cette façon de procéder, c'est que vous pouvez faire des traitements sophistiqués sur chacune des formes en particulier, comme des effets d'ombres, de flou, des rotations, des changements d'échelle, des distorsions, etc.
- x Il est à noter que tous ces réglages peuvent se faire soit sur une forme en particulier, soit sur une des vues du calque. Nous obtenons ainsi une grande souplesse d'utilisation et libère notre créativité.

## x PROJET D'ÉTUDE

Comme d'habitude, je vous propose de vous montrer l'ensemble de ces techniques au travers d'un projet relativement sophistiqué. Nous en profiterons pour mettre en œuvre des notions que nous avons déjà abordée lors d'une étude précédente, comme la transparence, les différents types de dégradé ainsi que la plupart des transformations intéressantes.



Dans notre applications vectorielle, nous avons la possibilité de rajouter cinq formes spécifiques :

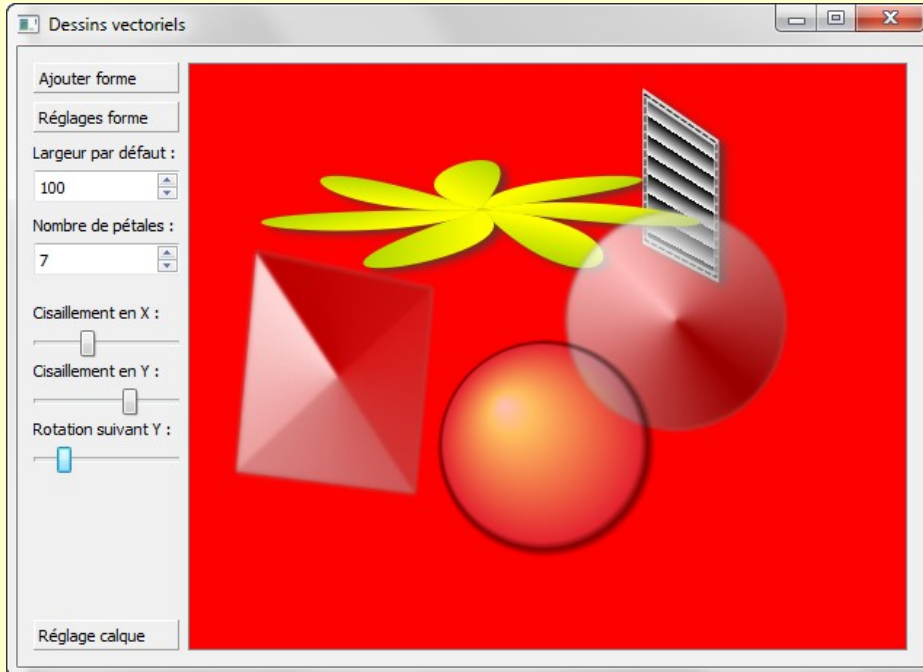
> Une fleur qui possède un nombre de pétales variable avec un dégradé linéaire du jaune vers le vert. La fleur est inclinée et possède dans son ensemble une ombre portée.

> Une pyramide transparente dont l'aspect a été mis en évidence en jouant sur le dégradé conique.

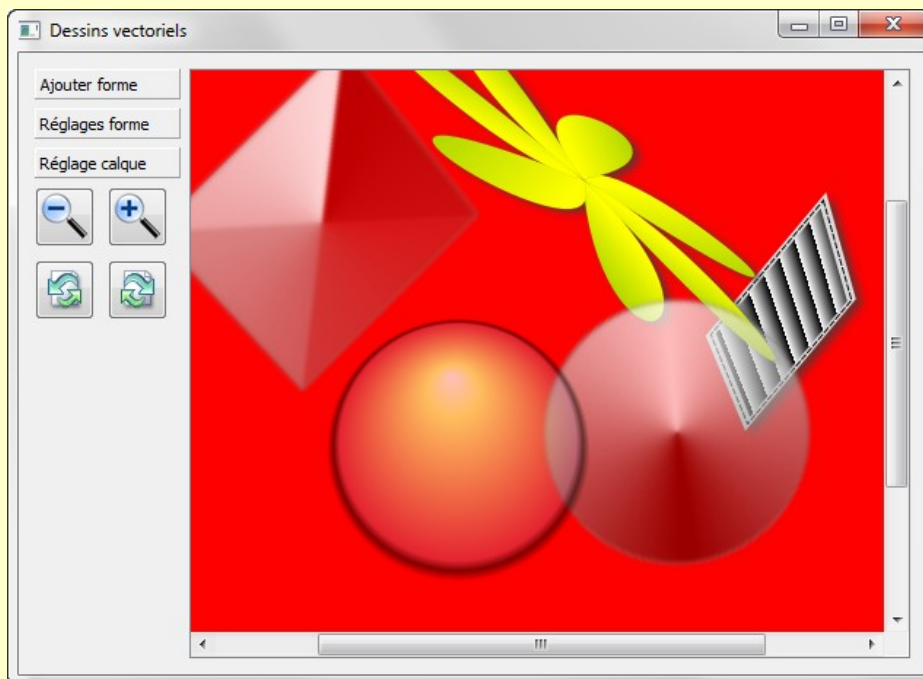
> Un cône également transparent avec des réglages tout autres sur ce même type de dégradé. La pyramide et le cône sont tous les deux légèrement flous.

> Une lentille convexe avec un ensemble de dégradé de type circulaire.

> Enfin un volet qui lui aussi a été mis en œuvre par un dégradé, mais cette fois-ci de type linéaire.



Chaque forme peut ensuite subir un certain nombre de réglages, comme le choix de la dimension, une rotation suivant l'axe des Y pour offrir une perspective et des déformations plus catégoriques comme les cisaillements suivant les deux axes.



Pour finir, il est possible de faire des réglages sur l'ensemble du calque (ou de la vue puisque une seule vue est associée au calque) avec possibilité de zoomer ou de tourner le calque suivant l'axe des Z.

## x LA VUE

Comme nous venons de le voir en introduction, la vue est représentée par la classe **QGraphicsView** qui est un **QWidget** qui a pour simple but d'afficher tout le contenu d'une scène.

x La méthode la plus importante est **setScene()** qui permet de choisir le calque à visualiser. Vous serez systématiquement obligé de faire appel à cette méthode. Une vue est toujours associé à une scène. Vous pouvez aussi passer par le constructeur pour spécifier la scène à visualiser.

x La vue propose automatiquement des ascenseurs dans le cas où le calque est plus grand que la fenêtre visuelle.

x La vue est bien entendu capable de prendre en compte tous les types d'événement que vous souhaitez, comme par exemple le fait de cliquer sur une zone particulière du calque. Ceci dit, la plupart du temps, il est préférable que ce soit le calque lui-même qui gère les événements plutôt que la vue. Dans certains cas toutefois, il peut être intéressant de gérer quelques événements spécifiques à partir de la vue. C'est notamment le cas si vous désirez gérer le zoom de l'ensemble des éléments, zoom complet du calque. Il suffit alors de redéfinir la méthode associée à la molette de la souris.

x Sur l'ensemble de la vue, vous pouvez réaliser toute sorte de transformations et d'effets, des changements d'échelles,





*des rotations, des translations, des cisaillements, des ombres portées, des flous gaussiens, etc.*

Je vous propose ci-dessous quelques méthodes bien utiles de la classe **QGraphicsView** pour être sûr de bien gérer correctement votre vue :

- x **QGraphicsView(parent) - QGraphicsView(scène, parent)** : Création du composant vu avec ou sans le calque.
- x **centerOn(x, y) - centerOn(point)** : change la position du centre de la vue.
- x **drawBackground(painter)** : méthode virtuelle à redéfinir afin de proposer un fond personnalisé.
- x **drawForeground(painter)** : méthode virtuelle à redéfinir afin de proposer un premier plan personnalisé (au dessus de tous les autres éléments).
- x **ensureVisible(rectangle)** : fait en sorte que la zone sélectionnée soit visible et agit automatiquement sur les ascenseurs en conséquence.
- x **itemAt(x, y) - itemAt(point)** : renvoie le vecteur (la forme) à la position spécifié.
- x **items()** : renvoie la liste des formes sur le calque.
- x **xxxEvent(événement spécifique)** : gestion événementielle suivant le « xxx » proposé (voir la liste des méthodes dans l'assistant).
- x **resetTransform()** : Annule toutes les transformations déjà effectuées sur la vue.
- x **rotate(angle)** : tourne le calque suivant l'angle proposé suivant l'axe des Z dans le sens trigonométrique.
- x **scale(x, y)** : changement d'échelle suivant l'axe des X et suivant l'axe des Y.

| Constante                         | Description   |
|-----------------------------------|---|
| QPainter::Antialiasing            | Mise en place de l'antialiasing sur l'ensemble des dessins.                                   |
| QPainter::TextAntialiasing        | Même chose mais pour le texte.  |
| QPainter::SmoothPixmapTransform   | L'algorithme d'échantillonnage des images doit être très précise au détriment de la rapidité. |
| QPainter::HighQualityAntialiasing | Grande précision de l'antialiasing en OpenGL.   |
| QPainter::NonCosmeticDefaultPen   | Possibilité de faire des tracés extrêmement fin qui s'approche du 0 pixel.                    |

- x **setScene(calque)** : choisi le calque à visualiser. Un seul calque est visualisable.
- x **setTransform(transformation)** : propose une transformation globale : déplacement, échelle, rotation, perspective, cisaillement ou partielle. Nous reverrons ultérieurement comment réaliser de telles transformations.
- x **shear(x, y)** : cisaillement suivant l'axe des X et l'axe des Y sur l'ensemble du calque.
- x **translate(x, y)** : déplacement relative en x et en y de l'ensemble du calque.

## x LES FORMES VECTORIELLES

Nous découvrirons le calque dans le chapitre suivant. Je vous propose de voir maintenant comment fabriquer des formes élémentaires vectorielles.

*Plutôt que de faire du tracé sur une surface graphique tel que nous l'avons fait au travers de la classe **QPainter**, nous privilégions maintenant le tracé de chaque forme élémentaire par un objet de type **QGraphicsItem**.*

L'intérêt de cette démarche, nous l'avons vu en introduction, c'est qu'il est alors facilement possible de faire des transformations (translation, échelle, rotation, cisaillement), ou de proposer des filtres (Ombre portée, flou gaussien) sur chacun de ces objets.

Il est également possible de regrouper ces éléments pour former une structure plus complexe suivant deux axes :

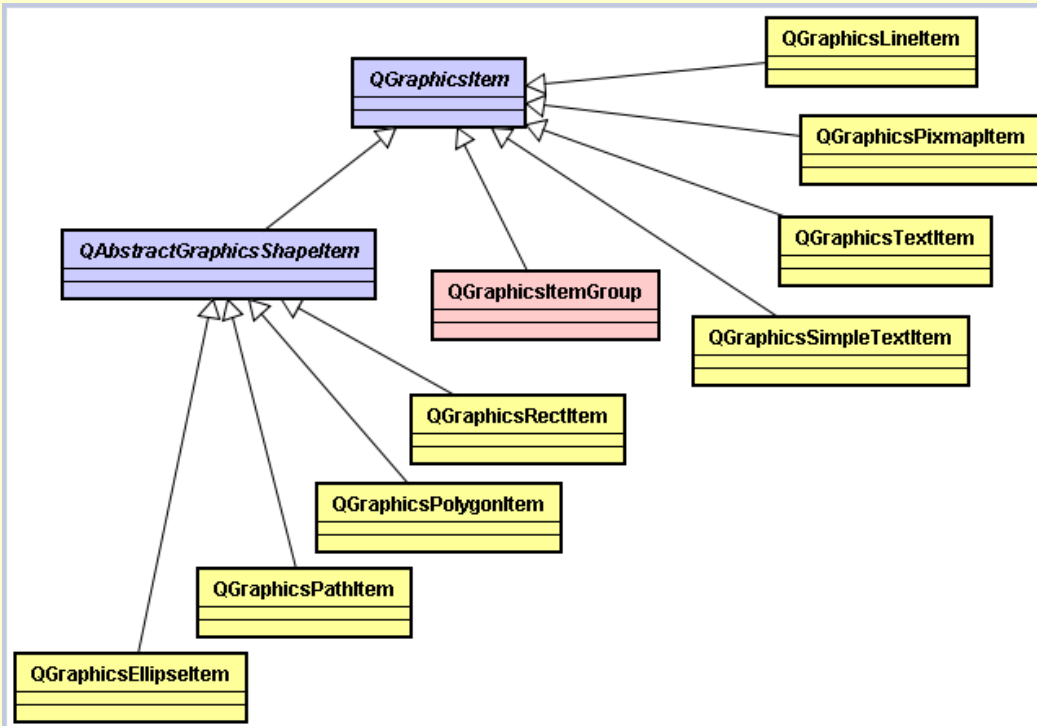
- x **Soit nous proposons une filiation entre éléments graphiques. Dans ce cas, un objet graphique complexe est alors composé d'un ensemble d'objet graphique élémentaire. La filiation se note soit au moment de la phase de construction de l'objet élémentaire, soit en faisant explicitement appel à la méthode **setParentItem()** de la classe **QGraphicsItem**.**
- x **Soit nous regroupons des objets élémentaires entre eux dont l'ensemble forme une structure cohérente. Nous devons alors créer un objet de la classe **QGraphicsItemGroup** qui dispose des méthodes **addToGroup()** et **removeFromGroup()** pour respectivement ajouter et supprimer un nouvel élément graphique à la structure globale.**

*Quelque soit l'ossature que nous choisissons, il peut être intéressant de regrouper des formes élémentaires entre elles afin d'effectuer en seule fois et de manière cohérente un ensemble de transformations linéaires et de réaliser ensuite quelques effets particuliers.*

*Par exemple, pour fabriquer ma fleur dans mon projet, je fabrique d'abord un pétale qui est un objet élémentaire graphique. J'effectue ensuite une rotation calculée sur ce pétale afin d'obtenir tous les autres qui deviennent également des objets graphiques élémentaires. Une fois tous ces pétales construits, ils sont ensuite affiliés à la fleur. L'ensemble de la fleur est ensuite inclinée avec en plus un filtre sur la totalité, de type ombre portée.*

La classe **QGraphicsItem** est en réalité une classe abstraite. Lorsque vous devez fabriquer des formes, vous devez le faire au travers de l'une de ces classe filles dont voici l'arborescence :





Dans l'ordre d'affichage proposé par la deuxième figure ci-contre, de gauche à droite et de haut en bas :

#### **QGraphicsRectItem**

Comme son nom l'indique, permet de tracer un vecteur sous forme rectangulaire.

#### **QGraphicsEllipseItem**

Comme son nom l'indique, permet de tracer un vecteur sous forme elliptique. Par contre, au moment de la construction, vous devez préciser la zone rectangulaire qui va circonscrire l'ellipse.

#### **QGraphicsPolygonItem**

Comme son nom l'indique, permet de tracer un vecteur sous forme de polygone. Pour sa construction, vous proposez alors une suite de point qui vont correspondre au sommet de chaque côté.

#### **QGraphicsLineItem**

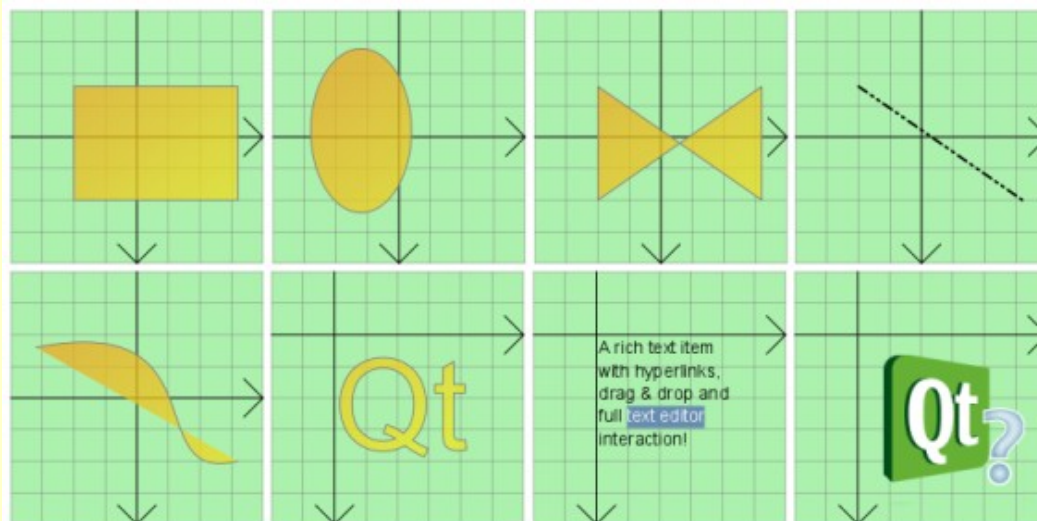
Comme son nom l'indique, permet de tracer un vecteur sous forme d'une simple ligne.

#### **QGraphicsPathItem**

Ce vecteur est particulier. Nous pouvons effectivement proposer une suite de tracés très sophistiqués dans une même entité. Nous passons par cette classe notamment lorsque nous devons mettre en place des courbes de Bézières.

#### **QGraphicsSimpleTextItem**

Comme son nom l'indique, permet de tracer un texte sous forme vectorielle. Dans ce cas, le texte est traité comme toutes les autres formes.



#### **QGraphicsTextItem**

Cette fois-ci, il s'agit d'un texte à par entière qui n'est plus considéré comme un vecteur, mais dont nous pouvons proposer des paragraphes, la mise en italique, des liens, etc.

#### **QGraphicsPixmapItem**

Là aussi, nous ne travaillons plus sous forme de vecteur. Cet élément représente tout simplement un Bitmap.

**x En réalité, vous n'êtes pas obligé de créer explicitement un de ces objets. Comme nous le découvrirons ultérieurement, l'objet scène **QGraphicsScene** dispose de méthodes comme **addRect()**, **addEllipse()**, etc. qui créent automatiquement pour vous chacun de ces objets, ce qui facilite grandement la tâche.**

Voici ci-dessous quelques méthodes de la classe **QGraphicsItem**, qui peuvent s'avérer utiles pour permettre la création de votre forme avec tous les réglages possibles :

**x `childItems()` : Donne la liste de toutes les formes qui compose cette forme complexe (éventuellement).**

**x `ensureVisible(rectangle)` : fait en sorte que la forme soit entièrement visible et agit automatiquement sur les ascenseurs de la vue en conséquence.**

**x `graphicsEffect()` : s'ils existent, renvoie les effets proposés sur cette forme.**

**x `group()` : retourne le groupe qui intègre cette forme.**



x **xxxEvent(événement spécifique)** : gestion événementielle suivant le « xxx » proposé (voir la liste des méthodes dans l'assistant).

x **hide()** : Cache momentanément la forme de la vue. Toutes les formes, dès leurs constructions, sont visibles par défaut.

x **isAncestorOf(enfant)** : indique si la forme passé en argument est un enfant de la forme globale (parente).

x **isObscured(zone) - isObscuredBy(autre forme)** : précise si la forme est cachée par d'autres entités.

x **isSelected() - setSelected(validation)** : indique si la forme a été sélectionnée. Il est aussi possible d'imposer que la forme soit sélectionnée.

x **isVisible() - setVisible(validation)** : indique si la forme est visible actuellement et peut la rendre visible ou pas.

x **moveBy(dx, dy)** : déplacement relatif de la forme suivant les axes des X et des Y.

x **opacity()** : retourne une valeur réelle comprise entre 0 (transparent) et 1 (opaque) qui informe sur le niveau d'opacité du composant.

x **parentItem()** : renvoie le parent de la forme (si il existe).

x **pos()** : renvoie la position, localisée sous forme de point, de la forme.

x **resetTransform()** : Annule toutes les transformations déjà effectuées sur la forme.

x **rotation()** : précise la rotation, sous forme d'une valeur réelle, déjà effectuée sur la forme actuelle.

x **scale()** : précise le changement d'échelle, sous forme d'une valeur réelle, déjà effectuée sur la forme actuelle.

x **scene()** : renvoie le calque sur lequel est tracé la forme.

x **setFlag(à prendre en compte)** : méthode très importante qui permet de proposer des comportement automatiques. Par exemple, si vous désirez que la forme soit automatiquement entourée d'une zone rectangulaire en pointillée lorsque la forme est sélectionnée, il faut activer la constante **ItemIsSelectable**. Voici quelques constantes importantes :

| Constante  | Description   |
|--|---|
| <b>QGraphicsItem::ItemIsMovable</b>              | Permet le déplacement de la forme à l'aide de la souris. Actif par défaut                             |
| <b>QGraphicsItem::ItemIsSelectable</b>           | Autorise la sélection de la forme en visualisant la zone de sélection par un rectangle en pointillé.  |
| <b>QGraphicsItem::ItemIsFocusable</b>            | La forme peut prendre le focus qui peut donc être accessible avec la touche de tabulation du clavier. |
| <b>QGraphicsItem::ItemIgnoresTransformations</b> | Permet d'empêcher les transformations proposées par un calque sur cette forme.                        |
| <b>QGraphicsItem::ItemIgnoresParentOpacity</b>   | Empêche la propagation du niveau d'opacité proposé par la forme parente.                              |

x **setGraphicsEffect(effet)** : propose un effet sur la forme : flou gaussien, ombre portée, transparence et colorisation suivant l'arrière plan.

x **setGroup(groupe)** : associe la forme à un groupe.

x **setOpacity(niveau d'opacité)** : propose un niveau de transparence sur la forme.

x **setParentItem(parent)** : indique quel est le parent conteneur de la forme.

x **setPos(nouvelle position)** : déplace la forme à l'endroit stipulée.

x **setRotation(angle)** : tourne la forme suivant l'axe des Z avec un angle exprimé en degré.

x **setScale(échelle)** : effectue un zoom sur la forme.

x **setToolTip(aide)** : propose une petite bulle d'aide lorsque le curseur de la souris passe au dessus de la forme.

x **setTransform(transformation)** : propose une transformation globale : déplacement, échelle, rotation, perspective, cisaillement ou partielle.

x **setZValue(niveau)** : les formes peuvent se retrouver les unes au dessus des autres. Cette méthode permet de changer l'ordre d'empilement suivant l'axe des Z. Il suffit de proposer une valeur entière qui donnera la priorité. La valeur par défaut est 0.

x **show()** : rend visible de nouveau la forme sur la vue.

## x LA SCÈNE – LE CALQUE

L'élément le plus important lorsque nous mettons en œuvre cette architecture est certainement la scène représentée par la classe **QGraphicsScene**. Cette scène correspond à un calque sur lequel nous plaçons nos différentes formes.

La scène ne peut être affichée directement. Vous êtes obligé de passer pour cela par un **QGraphicsView**.

Dans la pratique, nous travaillons essentiellement qu'avec la scène, même lorsque nous créons de nouvelles formes. En effet, cette classe possède des méthodes très sophistiquées qui permettent de tracer la forme désirée, en créant en même temps l'objet correspondant dans le type adapté.

Par exemple la méthode **addRect()** fabrique automatiquement un objet de type **QGraphicsRectItem**.

La plupart du temps, l'idéal est de créer une nouvelle classe qui hérite de **QGraphicsScene** afin de proposer





vos propres fonctionnalités, puisque pratiquement tous les traitements s'effectuent à ce niveau là. Voici la liste de toutes méthodes de cette classe :

- x **addEllipse(rectangle)** : fabrique et trace une ellipse dans le rectangle de délimitation, de type **QGraphicsEllipseItem** sur le calque et au dessus des autres formes. Sans aucune précision de localisation, le tracé s'effectue à partir de l'origine.
- x **addRect(rectangle)** : fabrique et trace un rectangle, de type **QGraphicsRectItem** sur le calque et au dessus des autres formes. Sans aucune précision de localisation, le tracé s'effectue à partir de l'origine.
- x **addItem(forme)** : ajoute une forme déjà construite qui peut être également une forme complexe composée d'autres formes ou un groupe de formes.
- x **addLine(ligne)** : fabrique et trace une ligne, de type **QGraphicsLineItem** sur le calque et au dessus des autres formes. Sans aucune précision de localisation, le tracé s'effectue à partir de l'origine.
- x **addPath(courbes)** : fabrique et trace une forme complexe souvent avec un ensemble de courbes de Béziérs, de type **QGraphicsPathItem** sur le calque et au dessus des autres formes. Sans aucune précision de localisation, le tracé s'effectue à partir de l'origine.
- x **addPixmap(image)** : fabrique et trace une image, de type **QGraphicsPixmapItem** sur le calque et au dessus des autres formes. Sans aucune précision de localisation, le tracé s'effectue à partir de l'origine.
- x **addPolygon(polygone)** : fabrique et trace un polygone, de type **QGraphicsPolygonItem** sur le calque et au dessus des autres formes. Sans aucune précision de localisation, le tracé s'effectue à partir de l'origine.
- x **addSimpleText(texte)** : fabrique et trace un texte sous forme vectorielle, de type **QGraphicsSimpleTextItem** sur le calque et au dessus des autres formes. Sans aucune précision de localisation, le tracé s'effectue à partir de l'origine.
- x **addText(texte)** : fabrique et trace un texte en prenant en compte la police de caractères, les styles, etc., de type **QGraphicsTextItem** sur le calque et au dessus des autres formes. Sans aucune précision de localisation, le tracé s'effectue à partir de l'origine.
- x **addWidget(composant graphique)** : possibilité de rajouter n'importe quel composant graphique sur ce calque au dessus des autres formes. Sans aucune précision de localisation, le tracé s'effectue à partir de l'origine.
- x **clear()** : efface le contenu et rend le calque totalement vierge de tout tracé.
- x **clearSelection()** : désactive toutes les sélections.
- x **createItemGroup(liste des formes)** : crée un groupe de forme avec la liste proposée en argument.
- x **xxxEvent(événement spécifique)** : gestion événementielle suivant le « xxx » proposé (voir la liste des méthodes dans l'assistant).
- x **destroyItemGroup(groupe)** : détruit le groupe de formes.
- x **drawBackground(painter)** : méthode virtuelle à redéfinir afin de proposer un fond personnalisé.
- x **drawForeground(painter)** : méthode virtuelle à redéfinir afin de proposer un premier plan personnalisé (au dessus de tous les autres éléments).
- x **height()** : hauteur du calque.
- x **invalidate() - invalidate(rectangle)** : rafraîchit le tracé de l'ensemble ou de la portion du calque.
- x **itemAt(position)** : retourne la forme sous la position définie par l'argument.
- x **items()** : déplacement relatif de la forme suivant les axes des X et des Y.
- x **removeItem(forme)** : enlève et détruit la forme du calque.
- x **selectedItems()** : donne la liste de l'ensemble des formes sélectionnées.
- x **.setSceneRect(rectangle)** : propose les dimensions du calque en définissant le point origine.
- x **setFont(police)** : Propose une nouvelle police pour le tracé de texte.
- x **setForegroundBrush(pinceau)** : propose un nouveau pinceau par défaut qui sera opérationnel pour les prochaines formes.
- x **setBackgroundBrush(pinceau)** : propose un nouveau fond pour le calque.
- x **width()** : largeur du calque.

## x FORME ÉLÉMENTAIRE AVEC TRACÉS COMPLEXES

Nous possédons des objets de type **QGraphicsItem** qui sont capables de tracer des formes très rudimentaires comme les rectangles, les ellipses, les polygones, etc. Nous pouvons avoir besoin de tracer des formes plus complexes tout en les considérant comme élémentaire comme c'est notamment le cas avec tous les tracés curvilignes. C'est la classe **QPainterPath** qui permet de réaliser ce type de tracé.

- x La classe **QPainterPath** permet d'implémenter des objets qui sont capables de construire un ensemble de tracés personnalisés à partir d'un point origine et en prenant en compte éventuellement d'autres tracés de type **QPainterPath**.
- x Une fois, qu'un objet vierge de type **QPainterPath** vient d'être créé, en ayant fixé le point d'origine, vous tracez la suite de lignes et/ou de courbes qui vous intéressent à l'aide des méthodes respectives **lineTo()**, **arcTo()**, **cubicTo()** et **quadTo()**.
- x A chaque fois que vous utilisez l'une de ces méthodes, vous partez de la position actuelle correspondant au dernier tracé réalisé, et vous passez en argument le point final qui va permettre de réaliser la ligne ou la courbe demandée.





x Ce point final devient le nouveau point origine pour le prochain tracé. Il est possible de connaître à tout moment la position actuelle à l'aide de la méthode `currentPosition()`.

x Généralement, pour fabriquer la forme qui vous intéresse, vous devez proposer une suite de lignes et de courbes qui se suivent dans votre tracé. Quelque fois, vous avez besoin de provoquer une rupture pour commencer un nouveau tracé un peu plus loin. La méthode `moveTo()` permet justement de changer la nouvelle position de référence sans qu'aucun tracé particulier ne soit réalisé.

x A la place de cette méthode `moveTo()`, vous pouvez également rompre une suite de tracés qui finalise une forme complexe. La méthode `closeSubpath()` clôture cette forme en proposant la nouvelle position de référence sur le point origine. C'est la différence essentielle avec `moveTo()`.

x La classe `QPainterPath` permet également de rajouter des formes toutes faites au tracé déjà effectué, à l'aide des méthodes `addEllipse()`, `addPath()`, `addPolygon()`, `addRect()` et `addText()`.



```
QPainterPath path;
path.addRect(20, 20, 60, 60);

path.moveTo(0, 0);
path.cubicTo(99, 0, 50, 50, 99, 99);
path.cubicTo(0, 99, 50, 50, 0, 0);

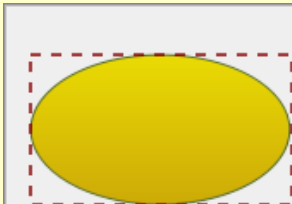
QPainter painter(this);
painter.fillRect(0, 0, 100, 100, Qt::white);
painter.setPen(QPen(QColor(79, 106, 25), 1, Qt::SolidLine,
                    Qt::FlatCap, Qt::MiterJoin));
painter.setBrush(QColor(122, 163, 39));

painter.drawPath(path);
```

x

x `QPainterPath()` - `QPainterPath(origine)` : fabrique une forme vide de tout tracé en fixant éventuellement un nouveau point de référence autre que le point origine.

x `addEllipse(rectangle)` : trace une ellipse dans le rectangle de délimitation et rajoute ce tracé aux autres déjà effectués. Le tracé est considéré comme clôturé, ce qui veut dire que la position courante est alors le point origine.



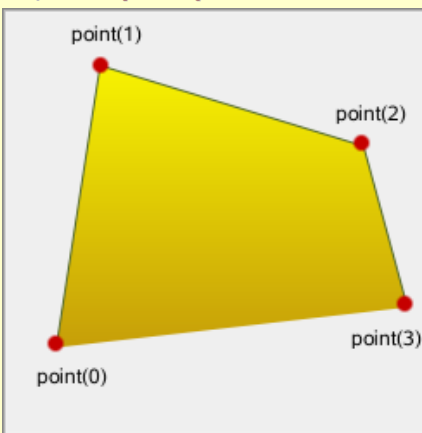
```
QLinearGradient myGradient;
QPen myPen;
QRectF boundingRectangle;

QPainterPath myPath;
myPath.addEllipse(boundingRectangle);

QPainter painter(this);
painter.setBrush(myGradient);
painter.setPen(myPen);
painter.drawPath(myPath);
```

x `addPath(tracés)` : Ajoute un tracé complexe aux autres tracés déjà effectués.

x `addPolygon(polygone)` : trace un polygone et rajoute ce tracé aux autres déjà effectués. La particularité d'un tracé de polygone, c'est que la position courante à l'issue de son tracé correspond au dernier point du polygone.



```
QLinearGradient myGradient;
QPen myPen;
QPolygonF myPolygon;

QPainterPath myPath;
myPath.addPolygon(myPolygon);

QPainter painter(this);
painter.setBrush(myGradient);
painter.setPen(myPen);
painter.drawPath(myPath);
```


x





x **addRect(rectangle)** : trace un rectangle et rajoute ce tracé aux autres déjà effectués. Le tracé est considéré comme clôturé, ce qui veut dire que la position courante est alors le point origine.

currentPosition()




```
QLinearGradient myGradient;
QPen myPen;
QRectF myRectangle;

QPainterPath myPath;
myPath.addRect(myRectangle);

QPainter painter(this);
painter.setBrush(myGradient);
painter.setPen(myPen);
painter.drawPath(myPath);
```

x **addRoundRect(rectangle, rayons)** : trace un rectangle à bords arrondis et rajoute ce tracé aux autres déjà effectués. Le tracé est considéré comme clôturé, ce qui veut dire que la position courante est alors le point origine.

x **addText(point de référence, police, texte)** : trace un texte et rajoute ce tracé aux autres déjà effectués. Le tracé est considéré comme clôturé, ce qui veut dire que la position courante est alors le point origine.




```
QLinearGradient myGradient;
QPen myPen;
QFont myFont;
QPointF baseline(x, y);

QPainterPath myPath;
myPath.addText(baseline, myFont, tr("Qt"));

QPainter painter(this);
painter.setBrush(myGradient);
painter.setPen(myPen);
painter.drawPath(myPath);
```

x **arcTo(rectangle, angle de départ, étendue)** : trace un arc dans le rectangle de délimitation à partir de l'angle proposé et suivant l'étendue spécifiée et rajoute ce tracé aux autres déjà effectués. Le tracé est considéré comme clôturé, ce qui veut dire que la position courante est alors le point origine.



```
QLinearGradient myGradient;
QPen myPen;

QPointF center, startPoint;

QPainterPath myPath;
myPath.moveTo(center);
myPath.arcTo(boundingRect, startPoint,
            sweepLength);

QPainter painter(this);
painter.setBrush(myGradient);
painter.setPen(myPen);
painter.drawPath(myPath);
```

x **boundingRect()** : retourne toute la zone rectangulaire prise par l'ensemble du tracé complexe.

x **closeSubPath()** : clôture le tracé en cours et propose du coup comme nouvelle position courante le point origine.

x **connectPath(tracé)** : connecte un nouveau tracé à ceux déjà effectués en rajoutant une ligne entre le dernier point de référence et le début du tracé proposé en argument.

x **contains(élément)** : indique si l'élément passé en argument se trouve bien à l'intérieur de la zone rectangulaire où se situe l'ensemble du tracé.

x **cubicTo(tangentes et point final)** : ajoute une courbe de Bézières entre la position courante et le point final proposé en argument en tenant compte des deux tangentes liées à ces deux points extrêmes. Après ce tracé, la position courante correspond au point final.







```

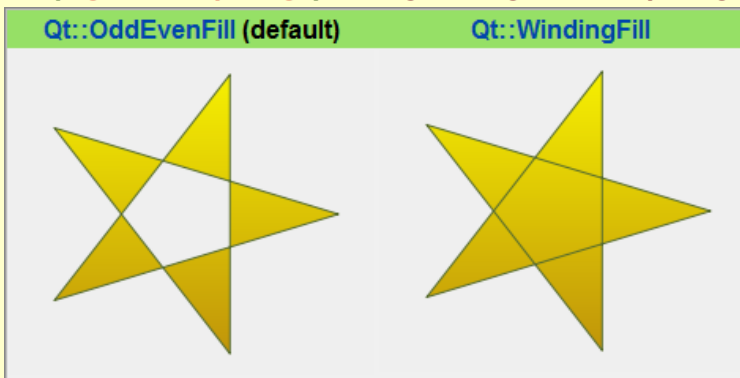
QLinearGradient myGradient;
QPen myPen;

QPainterPath myPath;
myPath.cubicTo(c1, c2, endPoint);

QPainter painter(this);
painter.setBrush(myGradient);
painter.setPen(myPen);
painter.drawPath(myPath);

```

- x **currentPosition()** : renvoie la position de référence actuelle.
- x **elementAt(index)** : renvoie le n<sup>ème</sup> tracé élémentaire.
- x **elementCount()** : comptabilise le nombre de tracés élémentaires déjà effectués.
- x **intersects(rectangle ou tracé)** : indique si une partie de l'élément proposé en argument se trouve à l'intérieur de la zone rectangulaire de l'ensemble du tracé complexe actuel.
- x **isEmpty()** : Existe-t-il le moindre tracé ?
- x **length()** : longueur « linéaire » totale du tracé.
- x **lineTo(point final)** : ajoute une ligne entre la position courante et le point final proposé en argument. Après ce tracé, la position courante correspond au point final.
- x **moveTo(point de référence)** : change le nouveau point de référence.
- x **quadTo(point intermédiaire et point final)** : ajoute une courbe de Béziérs quadratique entre la position courante et le point final proposé en argument en tenant compte du point intermédiaire. Après ce tracé, la position courante correspond au point final.
- x **setElementPositionAt(index, x, y)** : déplace la position du point origine de l'élément proposé en argument.
- x **setFillRule(règle de remplissage)** : change les règles de remplissage pour les formes complexes imbriquées.



- x **subtracted(tracé)** : retourne un tracé qui est une soustraction du tracé actuel avec celui passé en argument.
- x **toReversed()** : crée et renvoi un tracé dont l'ordre de tracer est inversé.
- x **translate(nouveau point)** : déplace l'ensemble du tracé relativement ou vers une nouvelle référence.
- x **translated(nouveau point)** : retourne une copie de l'ensemble du tracé vers une nouvelle référence relative ou absolue.
- x **united(tracé)** : retourne un tracé qui est une union entre le tracé actuel et celui passé en argument.

## x LES TRANSFORMATIONS

Les transformations sont souvent très utiles sur des objets graphiques. Effectivement, sur chacun d'eux, ou éventuellement sur l'ensemble du calque (au travers de la vue), vous pouvez proposer des déplacements, des changements d'échelle, des cisaillements, des rotations et bien d'autres fonctions affines. La classe **QTransform** implémente ces fonctionnalités là.

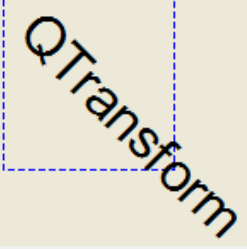
- x Elle possède des méthodes qui permettent de réaliser des transformations élémentaires, comme **scale()**, **rotate()**, **translate()** et **shear()** qui correspondent respectivement au changement d'échelle, à la rotation, au déplacement et au cisaillement.
- x Si nous passons par cette classe, nous pouvons proposer une suite de transformation élémentaire avec les méthodes que je viens de vous évoquer.
- x Mais, il est également possible d'effectuer des transformations en une seule fois au travers d'une matrice 3 x 3. Si nous





travaillons directement avec les matrices nous pouvons réaliser toutes les opérations habituelles sur les matrices, comme les additions, les soustractions et les multiplications.

- x Il n'est pas obligatoire de passer systématiquement par la classe **QTransform** pour réaliser les transformations désirées. Certaines sont effectivement déjà implémentées dans les différents objets de rendu graphique, comme par exemple la classe **QPainter**.

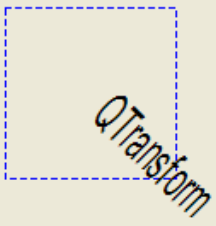


```
void SimpleTransformation::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.setPen(QPen(Qt::blue, 1, Qt::DashLine));
    painter.drawRect(0, 0, 100, 100);

    painter.rotate(45);

    painter.setFont(QFont("Helvetica", 24));
    painter.setPen(QPen(Qt::black, 1));
    painter.drawText(20, 10, "QTransform");
}
```

- x Dans le cas où plusieurs transformations sont nécessaires pour un même élément graphique, il est quand même préférable de passer par cette classe spécialisée **QTransform**, et de réaliser le traitement au moyen de la méthode **setTransform()** qui existe sur chacun des objets de rendu graphique, comme avec la classe **QPainter**.



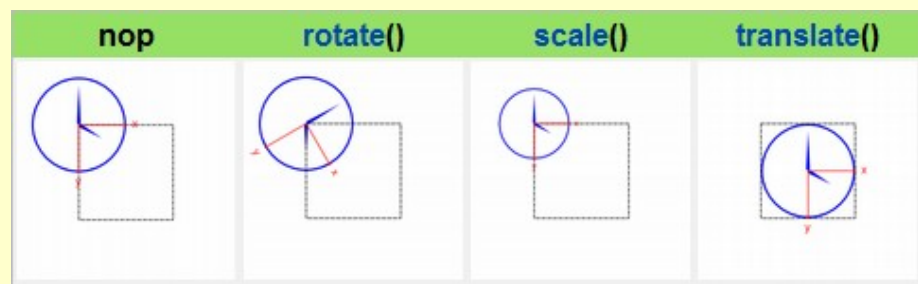
```
void CombinedTransformation::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.setPen(QPen(Qt::blue, 1, Qt::DashLine));
    painter.drawRect(0, 0, 100, 100);

    QTransform transform;
    transform.translate(50, 50);
    transform.rotate(45);
    transform.scale(0.5, 1.0);
    painter.setTransform(transform);

    painter.setFont(QFont("Helvetica", 24));
    painter.setPen(QPen(Qt::black, 1));
    painter.drawText(20, 10, "QTransform");
}
```

- x En réalité, un objet de type **QTransform** est de toute façon structurée sous forme de matrice 3 x 3.

|           |           |     |
|-----------|-----------|-----|
| m11       | m12       | m13 |
| m21       | m22       | m23 |
| m31<br>dx | m32<br>dy | m33 |



- x Les éléments **m31(dx)** et **m32(dy)** correspondent respectivement aux translations horizontale et verticale.
- x Les éléments **m11** et **m22** correspondent respectivement aux échelles horizontale et verticale.
- x Les éléments **m21** et **m12** correspondent respectivement aux cisaillements horizontal et vertical.
- x Enfin, les éléments **m13** et **m23** correspondent respectivement aux projections horizontale et verticale, avec **m33** qui joue le rôle de coefficient multiplicateur sur ces projections.
- x Grâce à cette matrice, nous pouvons donc réaliser un certain nombre d'opération de transformations élémentaires très simplement. La translation en fait partie. Ainsi, en réglant les valeurs de **dx** et de **dy**, vous changez les origines suivant l'axe des X et suivant l'axe des Y. Le changement d'échelle s'exécute au moyen de **m11** et **m22**. Par exemple, en proposant la valeur **2** pour **m11** et la valeur **1.5** pour **m22**, vous doublez la hauteur et vous amplifiez de 50% la largeur. Le cisaillement est contrôlé par **m12** et **m21**. La rotation est envisageable en agissant cette fois-ci sur plusieurs éléments, les facteurs de cisaillement avec les facteurs de changement d'échelle. Enfin, la perspective peut être mise en

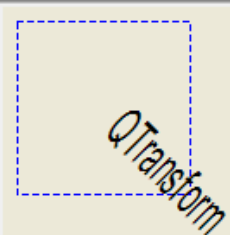


œuvre au moyen des facteurs de projection avec là aussi les facteurs de changement d'échelle.

x Comme premier exemple, pour transformer un point du plan vers un autre point, utilisez la formule suivante :

```
x' = m11*x + m21*y + dx
y' = m22*y + m12*x + dy
if (is not affine) {
    w' = m13*x + m23*y + m33
    x' /= w'
    y' /= w'
}
```

x Je vous propose également une combinaison de transformations qui utilise aussi le concept de matrice :



```
void BasicOperations::paintEvent(QPaintEvent *)
{
    double pi = 3.14;

    double a    = pi/180 * 45.0;
    double sina = sin(a);
    double cosa = cos(a);

    QTransform translationTransform(1, 0, 0, 1, 50.0, 50.0);
    QTransform rotationTransform(cosa, sina, -sina, cosa, 0, 0);
    QTransform scalingTransform(0.5, 0, 0, 1.0, 0, 0);

    QTransform transform;
    transform = scalingTransform * rotationTransform * translationTransform;

    QPainter painter(this);
    painter.setPen(QPen(Qt::blue, 1, Qt::DashLine));
    painter.drawRect(0, 0, 100, 100);

    painter.setTransform(transform);

    painter.setFont(QFont("Helvetica", 24));
    painter.setPen(QPen(Qt::black, 1));
    painter.drawText(20, 10, "QTransform");
}
```

x **QTransform()** : fabrique une matrice de transformation où tous les éléments de la matrice sont positionnés à 0 excepté pour **m11** et **m22** (valeurs d'échelle) et **m13** qui sont placés à 1.

x **QTransform(m11, m12, m13, m21, m22, m23, m31, m32, m33)** : construit la matrice de transformation en prenant en compte tous les éléments.

x **QTransform(m11, m12, m21, m22, dx, dy)** : construit la matrice de transformation en prenant en compte les éléments conventionnels.

x **m11() - m12() - m13() - m21() - m22() - m23() - m31() - m32() - m33() - dx() - dy()** : retourne un facteur de la matrice.

x **adjoint()** : retourne l'adjoint de la matrice.

x **determinant()** : retourne le déterminant de la matrice.

x **fromScale(x, y)** : méthode statique qui retourne une transformation en relation avec un changement d'échelle.

x **fromTranslate(x, y)** : méthode statique qui retourne une transformation en relation avec un déplacement relatif.

x **inverted()** : retourne la matrice inverse.

x **reset()** : réinitialise la matrice.

x **rotate(angle)** : propose une transformation de rotation avec un angle exprimé en degré.

x **scale(x, y)** : propose une transformation de changement d'échelle suivant l'axe des X et suivant l'axe des Y.

x **shear(x, y)** : propose une transformation de type cisaillement suivant l'axe des X et suivant l'axe des Y.

x **translate(x, y)** : propose une transformation de déplacement relatif suivant l'axe des X et suivant l'axe des Y.

## x LES EFFETS

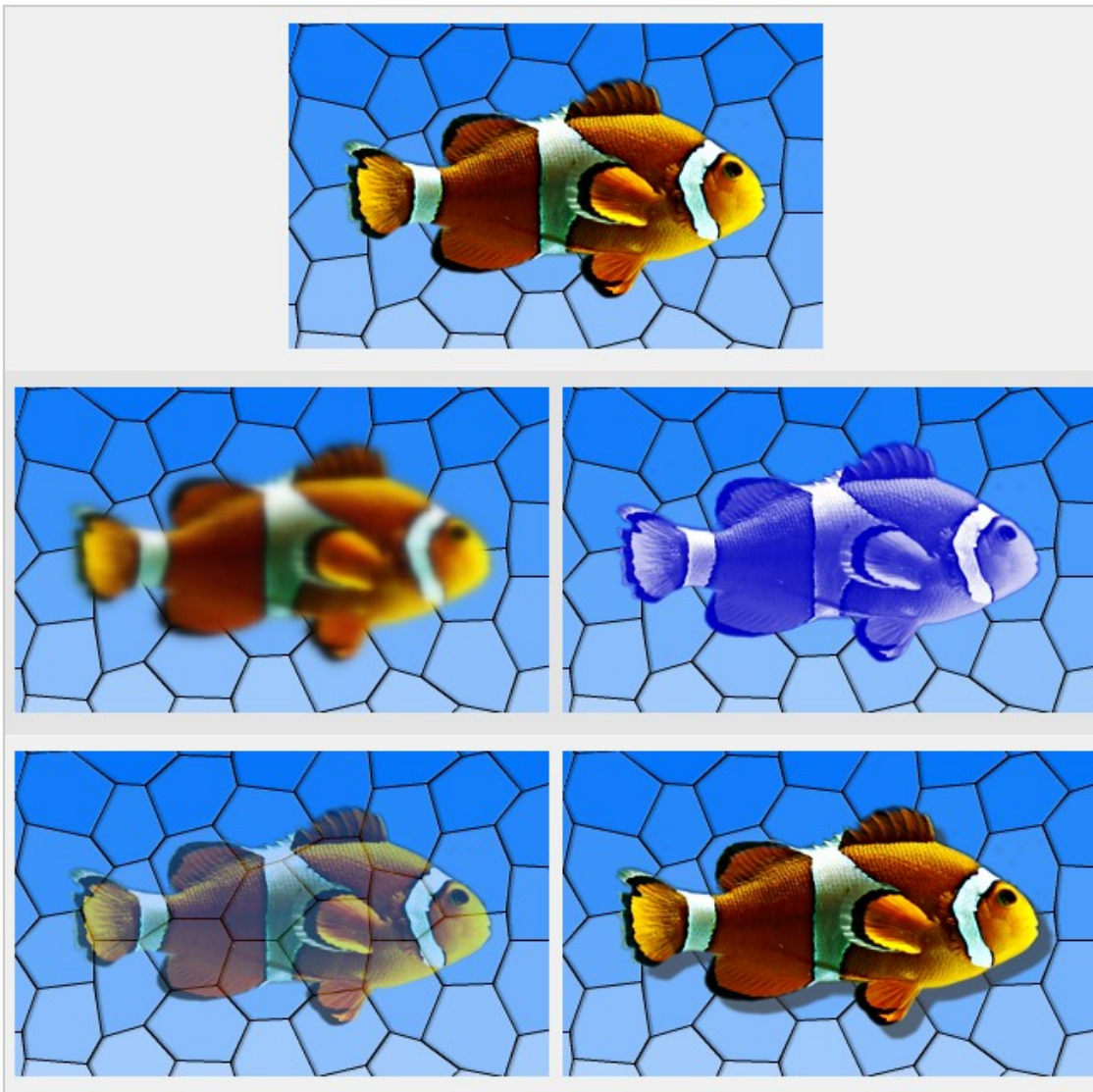
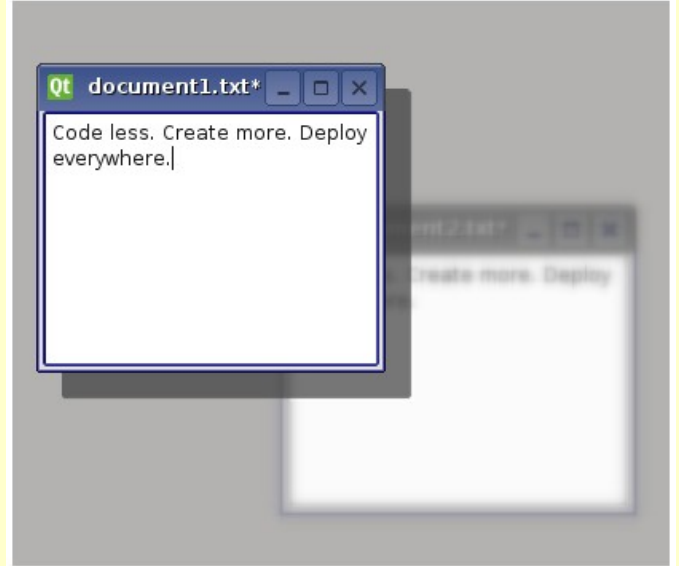
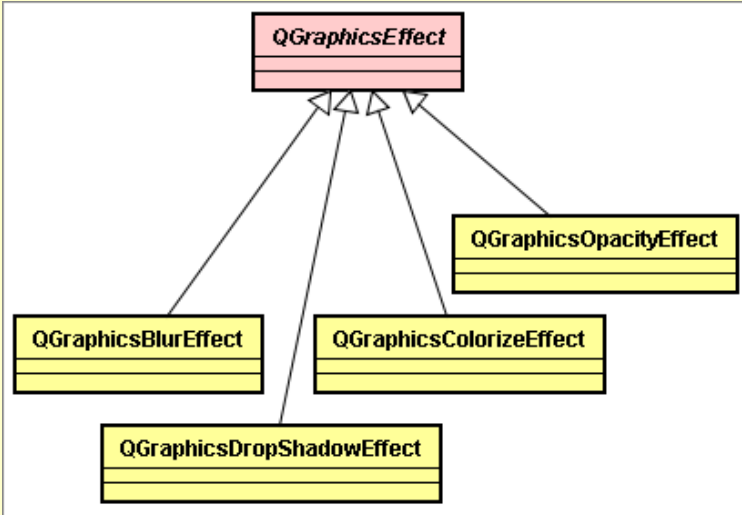
Nous pouvons donc à tout instant proposer des transformations sur chacun des éléments graphiques. Il est tout-à-fait envisageable de rajouter des effets supplémentaires, comme une ombre portée, un flou gaussien, une semi-transparence, etc.





Là aussi, vous disposez de classes spécialisées sur chacun des effets souhaités. Toutes ces classes héritent de la classe abstraite **QGraphicsEffect**. Pour mettre en œuvre un effet particulier, comme nous l'avons déjà vu, il suffit alors de faire appel à la méthode **setGraphicsEffect()** d'une de ces classes filles de **QGraphicsItem** dont voici leurs rôles :

- x le flou avec **QGraphicsBlurEffect**,
- x la colorisation avec **QGraphicsColorizeEffect**,
- x la transparence avec **QGraphicsOpacityEffect**
- x et enfin l'ombre portée avec **QGraphicsDropShadowEffect**.





x **QGraphicsBlurEffect** : Propose un flou gaussien. Cet effet est utilisé pour réduire les détails. Lorsque par exemple vous désirez faire en sorte qu'une partie de l'image soit moins intéressante, vous la rendez floue, pour porter l'attention sur d'autres éléments plus importants qui eux restent bien nets.

x Vous pouvez régler le rayon du flou au moyen de la méthode `setBlurRadius()` et son intensité au moyen de la méthode `setBlurHints()`.

x Par défaut, le rayon du flou est réglé à **5**.

x

x **QGraphicsColorizeEffect** : Propose de changer globalement la teinte de l'élément graphique.

x Vous pouvez sélectionner la couleur souhaité au moyen de la méthode `setColor()`.

x Par défaut, la couleur proposée est le bleu clair (`QColor(0, 0, 192)`).

x

x **QGraphicsOpacityEffect** : Propose une semi transparence sur l'élément graphique.

x Ce niveau de transparence peut être réglé au moyen de la méthode `setOpacity()`.

x Par défaut, l'opacité est réglée à **0,7**.

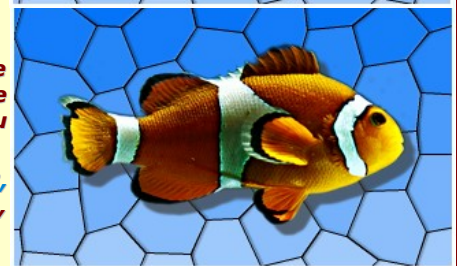
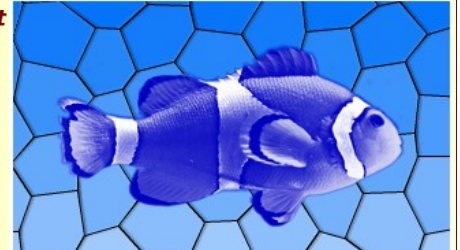
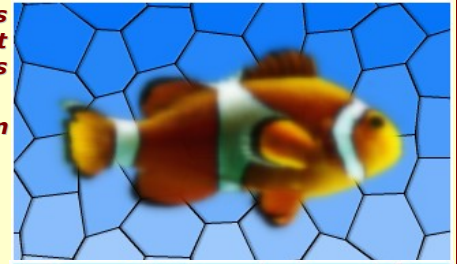
x

x **QGraphicsDropShadowEffect** : Propose une ombre portée sur l'élément graphique.

x Nous pouvons proposer la largeur et l'orientation de l'ombre au moyen de la méthode `setOffset()`. Nous pouvons également rajouter plus ou moins de flou sur cette ombre au moyen de la méthode `setBlurRadius()`. Enfin, nous pouvons choisir sa couleur au moyen de la méthode `setColor()`.

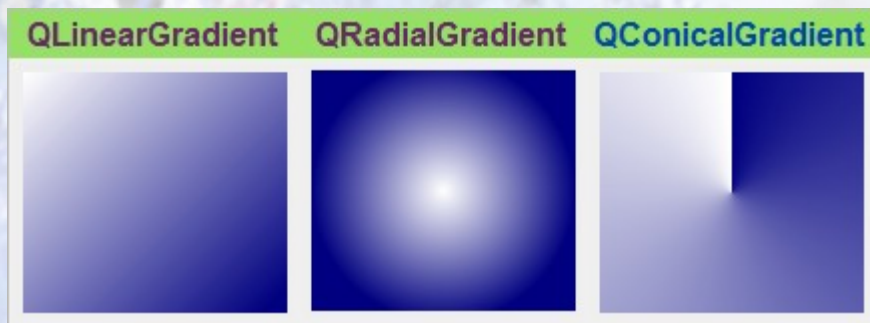
x Par défaut, l'ombre portée est semi-transparente de couleur gris foncé (`QColor(63, 63, 63, 180)`) avec un léger flou de rayon **1**, avec également une largeur de l'ombre, en bas à droite de **8** pixels.

x



## x RETOUR SUR 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.

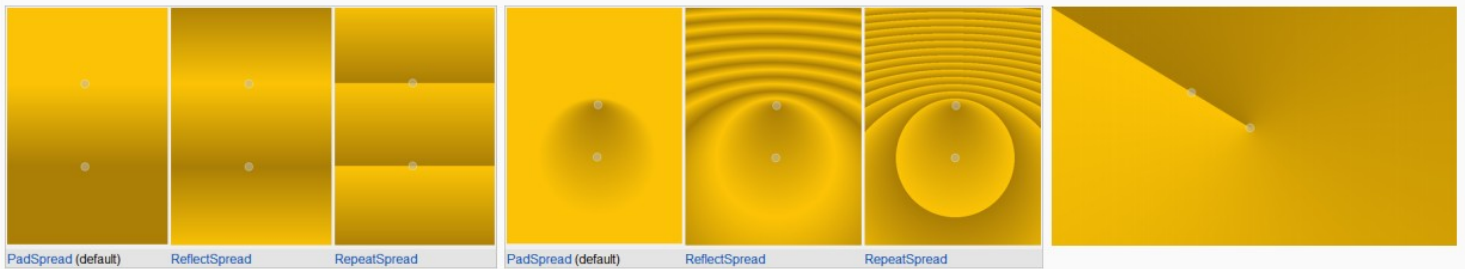
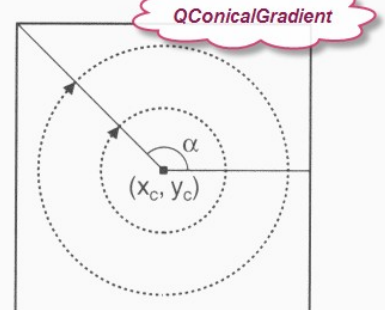
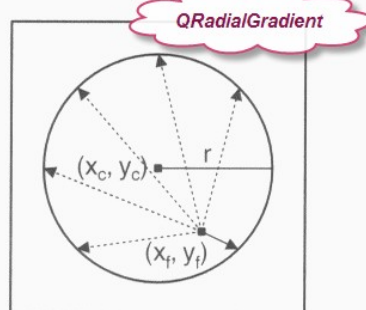
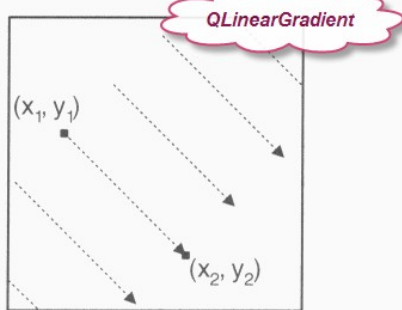


x Les dégradés linéaires sont définis par deux points de contrôle et par une série « d'arrêt de couleur » sur la ligne qui relie ces deux points. Attention, les positions sont indiquées comme des valeurs à virgule flottante entre 0 et 1, où 0 correspond au premier point de contrôle et 1 au second. Les couleurs situées entre les interruptions spécifiées sont interpolées. `QLinearGradient(x1, y1, x2, y2)`

x Les dégradés circulaires sont définis par un centre (x<sub>c</sub>, y<sub>c</sub>), un rayon r et une focale (x<sub>f</sub>, y<sub>f</sub>), 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(xc, yc, r, xf, yf)`



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



x **Les dégradés coniques sont définis par un centre  $(x_c, y_c)$ , un angle  $\alpha$ . Les couleurs se diffusent autour du point central comme la trajectoire de la petite aiguille d'une montre. **QConicalGradient** $(x_c, y_c, \alpha)$**

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 :

- x **QGradient::PadSpread** (par défaut)
- x **QGradient::ReflectSpread**
- x **QGradient::RepeatSpread**

## x RETOUR SUR LES SYSTÈMES DE COULEUR

Une couleur est définie avec une qualité de 24 ou 32 bits. Une couleur 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 (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 :

- x **QRgb rouge = qRgba(255, 0, 0, 255) ;**
- x **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 :

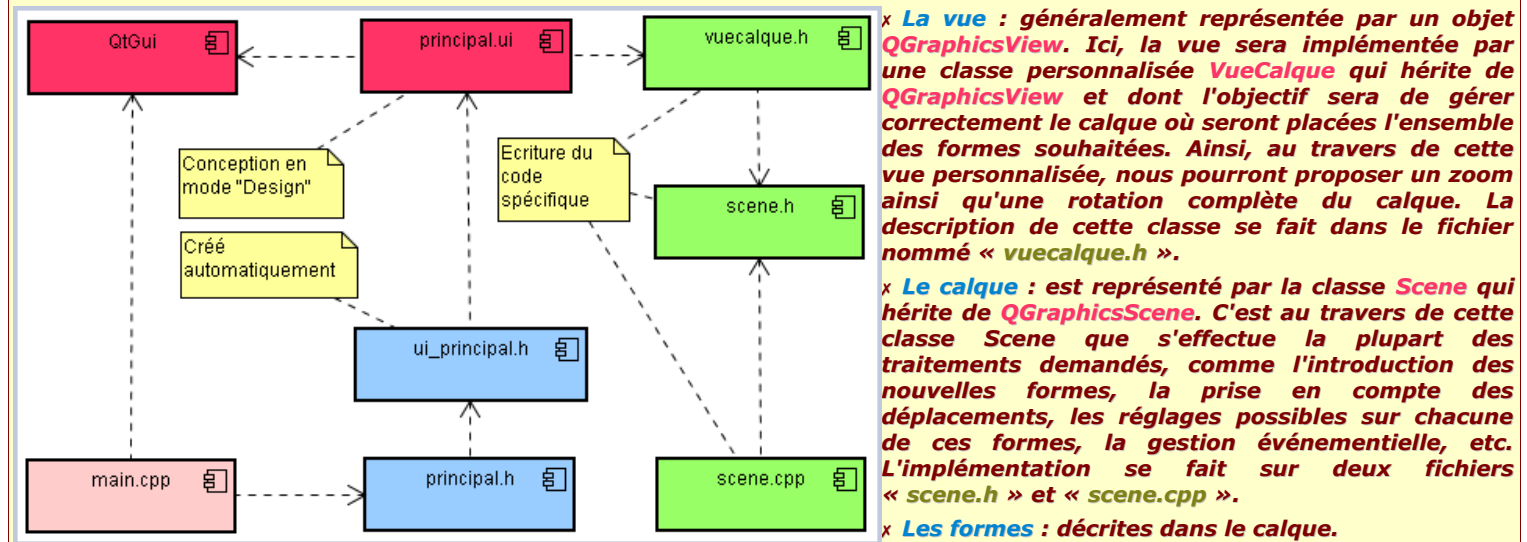
- x **QRgb rouge = 0xFFFF0000 ; // la premier FF correspond au canal alpha et le second FF à la composante rouge.**

**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.

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

## x IMPLÉMENTATION DU PROJET

Après avoir pris connaissance sur toutes les notions nécessaires à l'élaboration de graphismes 2D, je vous propose de les maîtriser au travers de notre projet d'étude. Je rappelle que pour élaborer correctement ces graphismes 2D, nous devons nous préoccuper de trois structures différentes :



x **La vue** : généralement représentée par un objet `QGraphicsView`. Ici, la vue sera implémentée par une classe personnalisée `VueCalque` qui hérite de `QGraphicsView` et dont l'objectif sera de gérer correctement le calque où seront placées l'ensemble des formes souhaitées. Ainsi, au travers de cette vue personnalisée, nous pourrions proposer un zoom ainsi qu'une rotation complète du calque. La description de cette classe se fait dans le fichier nommé « `vuecalque.h` ».

x **Le calque** : est représenté par la classe `Scene` qui hérite de `QGraphicsScene`. C'est au travers de cette classe `Scene` que s'effectue la plupart des traitements demandés, comme l'introduction des nouvelles formes, la prise en compte des déplacements, les réglages possibles sur chacune de ces formes, la gestion événementielle, etc. L'implémentation se fait sur deux fichiers « `scene.h` » et « `scene.cpp` ».

x **Les formes** : décrites dans le calque.

La capture d'écran montre l'interface Qt Creator avec l'éditeur de formes et l'explorateur de classes.

**Éditeur de formes :** Affiche une palette d'objets (cône, fleur, pyramide, lentille, voilette) et des réglages (forme et calque).

**Explorateur de classes :** Liste les classes et objets du projet :

| Objet          | Classe      |
|----------------|-------------|
| Principal      | QMainWindow |
| centralWidget  | QWidget     |
| toolbox        | QToolBox    |
| pageFormes     | QWidget     |
| boutonCone     | QPushButton |
| boutonFleur    | QPushButton |
| boutonLentille | QPushButton |
| boutonPyramide | QPushButton |
| boutonVoilette | QPushButton |
| pageReglages   | QWidget     |
| choixLargeur   | QSpinBox    |
| choixPetale    | QSpinBox    |
| cisaillementX  | QSlider     |
| cisaillementY  | QSlider     |
| labelLargeur   | QLabel      |
| labelNombre    | QLabel      |
| labelRotationY | QLabel      |
| labelX         | QLabel      |
| labelY         | QLabel      |
| rotationY      | QSlider     |
| page           | QWidget     |
| rotationDroite | QPushButton |
| rotationGauche | QPushButton |
| zoomMoins      | QPushButton |
| zoomPlus       | QPushButton |
| vue            | VueCalque   |

**Éditeur de Signaux Slots :**

| Émetteur       | Signal            | Receveur | Slot               |
|----------------|-------------------|----------|--------------------|
| boutonFleur    | clicked()         | vue      | ajoutFleur()       |
| boutonVoilette | clicked()         | vue      | ajoutVoilette()    |
| boutonPyramide | clicked()         | vue      | ajoutPyramide()    |
| boutonCone     | clicked()         | vue      | ajoutCone()        |
| boutonLentille | clicked()         | vue      | ajoutLoupe()       |
| choixLargeur   | valueChanged(int) | vue      | reglerLargeur(int) |
| choixPetale    | valueChanged(int) | vue      | nombrePetales(int) |
| cisaillementX  | valueChanged(int) | vue      | cisaillementX(int) |
| cisaillementY  | valueChanged(int) | vue      | cisaillementY(int) |
| rotationY      | valueChanged(int) | vue      | rotationAxeY(int)  |
| zoomMoins      | clicked()         | vue      | zoomMoins()        |
| zoomPlus       | clicked()         | vue      | zoomPlus()         |
| rotationDroite | clicked()         | vue      | rotationDroite()   |
| rotationGauche | clicked()         | vue      | rotationGauche()   |

**Propriétés de l'objet Principal :**

| Propriété              | Valeur                              |
|------------------------|-------------------------------------|
| objectName             | Principal                           |
| windowMo... (Modality) | NonModal                            |
| enabled                | <input checked="" type="checkbox"/> |
| geometry               | [(0, 0), 493 x 378]                 |
| sizePolicy             | [Preferred, Preferred, 0, 0]        |
| minimumSize            | 0 x 0                               |
| maximumSize            | 16777215 x 16777215                 |
| sizeIncrement          | 0 x 0                               |
| baseSize               | 0 x 0                               |



```

scene.h
Scene
Ligne : 34, Col : 1
1  #ifndef SCENE_H
2  #define SCENE_H
3
4  #include <QGraphicsScene>
5  #include <QGraphicsItem>
6
7  class Scene : public QGraphicsScene
8  {
9  public:
10     Scene();
11 protected:
12     void mousePressEvent(QGraphicsSceneMouseEvent *event);
13     void mouseMoveEvent(QGraphicsSceneMouseEvent *event);
14 public:
15     void genererFleur(int dx, int dy, unsigned nombre, unsigned longueur);
16     void genererOmbre(QGraphicsItem *);
17     void genererFlou(QGraphicsItem *);
18     void genererRideau(int x, int y, int cote);
19     void genererLoupe(int x, int y, int rayon);
20     void genererPyramide(int x, int y, int cote);
21     void genererCone(int x, int y, int rayon);
22     void changeEchelle(double echelle)          { if (forme!=0) forme->setScale(echelle); }
23     void cisaillementX(double horizontal);
24     void cisaillementY(double vertical);
25     void rotationAxeY(int degre);
26 private:
27     void activerForme(QGraphicsItem *forme);
28 private:
29     int x, y;
30     QGraphicsItem *forme;
31 };
32
33 #endif // SCENE_H

```

```

vuecalque.h
<Selectionner un symbole>
Ligne : 1, Col : 1
1  #ifndef VUECALQUE_H
2  #define VUECALQUE_H
3
4  #include "scene.h"
5  #include <QGraphicsView>
6  #include <QWheelEvent>
7
8  class VueCalque : public QGraphicsView
9  {
10     Q_OBJECT
11 public:
12     VueCalque(QWidget *parent = 0) : QGraphicsView(parent), scene(new Scene), largeur(150), nombre(7) { setScene(scene); }
13 private slots:
14     void ajoutFleur()          { scene->genererFleur(0, 0, nombre, largeur*2); }
15     void ajoutPyramide()      { scene->genererPyramide(0, 0, largeur); }
16     void ajoutCone()         { scene->genererCone(0, 0, largeur/2); }
17     void ajoutVoilet()       { scene->genererRideau(0, 0, largeur); }
18     void ajoutLoupe()        { scene->genererLoupe(0, 0, largeur/2); }
19     void reglerLargeur(int largeur) { this->largeur = largeur; scene->changeEchelle(largeur/150.0); }
20     void nombrePetales(int nombre) { this->nombre = nombre; }
21     void cisaillementX(int horizontal) { scene->cisaillementX(horizontal/100.0); }
22     void cisaillementY(int vertical) { scene->cisaillementY(vertical/100.0); }
23     void rotationAxeY(int degre) { scene->rotationAxeY(degre); }
24     void zoomMoins()          { scale(.8, .8); }
25     void zoomPlus()           { scale(1.2, 1.2); }
26     void rotationGauche()     { rotate(-10); }
27     void rotationDroite()     { rotate(10); }
28 protected:
29     void wheelEvent(QWheelEvent *event)
30     {
31         if (event->delta()>0) zoomPlus();
32         else zoomMoins();
33     }
34 private:
35     Scene *scene;
36     int largeur;
37     int nombre;
38 };
39
40 #endif // VUECALQUE_H

```



```

#include "scene.h"
#include <math.h>
#include <QtGui>

Scene::Scene()
{
    forme = 0;
    setBackgroundBrush(Qt::red);
    setSceneRect(-200, -200, 400, 400);
}

void Scene::genererOmbre(QGraphicsItem *element)
{
    QGraphicsDropShadowEffect *ombre = new QGraphicsDropShadowEffect;
    ombre->setBlurRadius(10);
    ombre->setOffset(5, 3);
    element->setGraphicsEffect(ombre);
}

void Scene::genererFlou(QGraphicsItem *element)
{
    QGraphicsBlurEffect *effet = new QGraphicsBlurEffect;
    effet->setBlurRadius(2);
    element->setGraphicsEffect(effet);
}

void Scene::genererFleur(int dx, int dy, unsigned nombre, unsigned longueur)
{
    double x = longueur * cos(M_PI/nombre) + dx;
    double y = longueur * sin(M_PI/nombre) + dy;

    QGraphicsPathItem *fleur = new QGraphicsPathItem();
    addItem(fleur);

    QPainterPath petale;
    petale.moveTo(0, 0);
    petale.cubicTo(0, -y, x, -y, 0, 0);

    QPen crayon(Qt::darkGreen, 0.1, Qt::SolidLine);
    QLinearGradient pinceau(0, 0, x, y);
    pinceau.setColorAt(0.0, Qt::yellow);
    pinceau.setColorAt(0.8, Qt::darkGreen);

    addPath(petale, crayon, pinceau)->setParentItem(fleur);

    for (unsigned i=1; i<nombre; i++) {
        QGraphicsPathItem *element = addPath(petale, crayon, pinceau);
        element->setRotation(i*360/nombre);
        element->setParentItem(fleur);
    }
    genererOmbre(fleur);

    QTransform axes;
    axes.rotate(75, Qt::XAxis);
    fleur->setTransform(axes);
    fleur->setFlag(QGraphicsItem::ItemIsSelectable, true);
    activerForme(fleur);
}

void Scene::genererRideau(int x, int y, int cote)
{
    QLinearGradient degrade(0, 0, 0, 20);
    degrade.setSpread(QGradient::RepeatSpread);
    QPen crayon(QColor(220, 220, 220), 10, Qt::SolidLine, Qt::RoundCap, Qt::MiterJoin);
    QGraphicsRectItem *rideau = addRect(x, y, cote, cote, crayon, degrade);
    rideau->setFlag(QGraphicsItem::ItemIsSelectable, true);
    genererOmbre(rideau);
    activerForme(rideau);
}

void Scene::genererLoupe(int x, int y, int rayon)
{
    QRadialGradient degrade(x+rayon, y+rayon, rayon, x+3 * rayon / 5, y+3 * rayon / 5);
    degrade.setColorAt(0.0, QColor(255, 255, 255, 191));
    degrade.setColorAt(0.2, QColor(255, 255, 127, 191));
    degrade.setColorAt(0.9, QColor(150, 150, 200, 63));
    degrade.setColorAt(0.95, QColor(0, 0, 0, 127));
    degrade.setColorAt(1, QColor(0, 0, 0, 0));
    QGraphicsEllipseItem *loupe = addEllipse(x, y, 2*rayon, 2*rayon, Qt::NoPen, degrade);
    loupe->setFlag(QGraphicsItem::ItemIsSelectable, true);
    activerForme(loupe);
}

```

```

void Scene::genererCone(int x, int y, int rayon)
{
    QConicalGradient degrade(x+rayon, y+rayon, 135);
    degrade.setColorAt(0.0, QColor(255, 255, 255, 191));
    degrade.setColorAt(0.5, QColor(0, 0, 0, 100));
    degrade.setColorAt(1, QColor(255, 255, 255, 191));
    QPen crayon(QColor(127, 127, 127, 150), 1.5);
    QGraphicsEllipseItem *cone = addEllipse(x, y, 2*rayon, 2*rayon, crayon, degrade);
    cone->setFlag(QGraphicsItem::ItemIsSelectable, true);
    genererFlou(cone);
    activerForme(cone);
}

void Scene::genererPyramide(int x, int y, int cote)
{
    QConicalGradient degrade(x+cote/2, y+cote/2, 135);
    degrade.setColorAt(0.0, QColor(255, 255, 255, 220));
    degrade.setColorAt(0.249, QColor(230, 230, 230, 180));
    degrade.setColorAt(0.251, QColor(200, 200, 200, 170));
    degrade.setColorAt(0.499, QColor(170, 170, 170, 135));
    degrade.setColorAt(0.501, QColor(140, 140, 140, 100));
    degrade.setColorAt(0.749, QColor(100, 100, 100, 63));
    degrade.setColorAt(0.751, QColor(63, 63, 63, 63));
    degrade.setColorAt(1, QColor(0, 0, 0, 63));
    QPen crayon(QColor(127, 127, 127, 150), 1.5);
    QGraphicsRectItem *pyramide = addRect(x, y, cote, cote, crayon, degrade);
    pyramide->setFlag(QGraphicsItem::ItemIsSelectable, true);
    genererFlou(pyramide);
    activerForme(pyramide);
}

void Scene::mousePressEvent(QGraphicsSceneMouseEvent *event)
{
    if (forme!=0) forme->setSelected(false);
    forme = itemAt(event->scenePos());
    if (forme == 0) return;
    if (forme->parentItem()!=0) forme = forme->parentItem();
    x = event->scenePos().x() - forme->x();
    y = event->scenePos().y() - forme->y();
    forme->setSelected(true);
}

void Scene::mouseMoveEvent(QGraphicsSceneMouseEvent *event)
{
    if (forme!=0) {
        int posX = event->scenePos().x() - x;
        int posY = event->scenePos().y() - y;
        forme->setPos(posX, posY);
    }
}

void Scene::cisaillementX(double horizontal)
{
    if (forme!=0) {
        QTransform mat = forme->transform();
        forme->setTransform(QTransform(mat.m11(), mat.m12(), horizontal, mat.m22(), 0, 0));
        // QTransform transformation;
        // transformation.shear(horizontal, forme->transform().m12());
        // forme->setTransform(transformation);
    }
}

void Scene::cisaillementY(double vertical)
{
    if (forme!=0) {
        QTransform mat = forme->transform();
        forme->setTransform(QTransform(mat.m11(), vertical, mat.m21(), mat.m22(), 0, 0));
    }
}

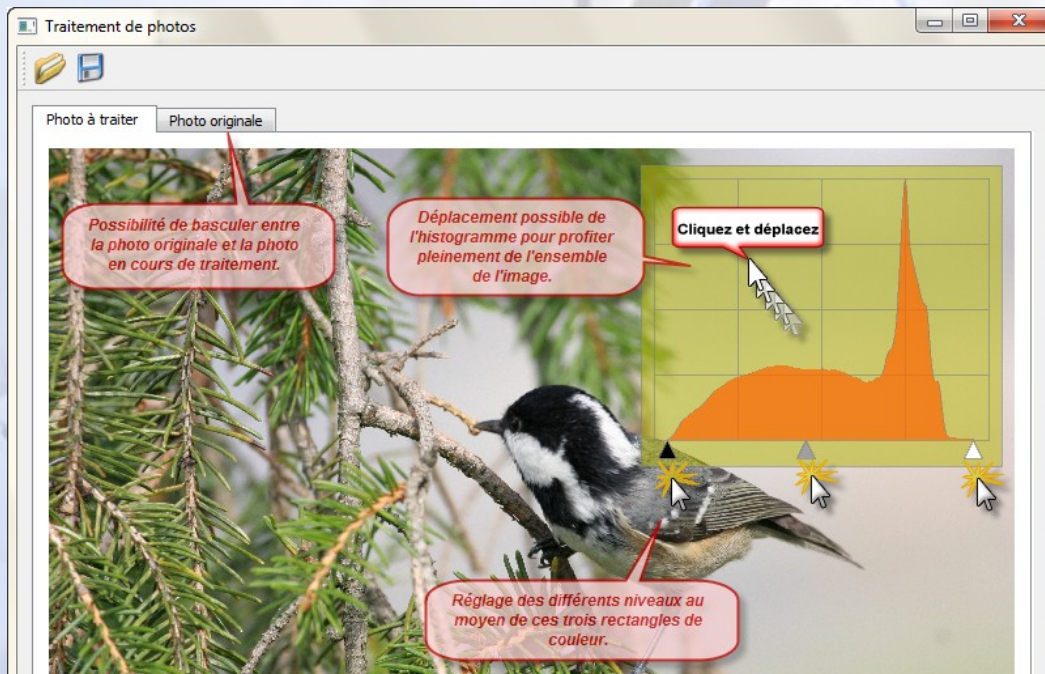
void Scene::rotationAxeY(int degre)
{
    if (forme!=0) {
        QTransform transformation;
        transformation.rotate(degre, Qt::YAxis);
        transformation.shear(forme->transform().m21(), forme->transform().m12());
        forme->setTransform(transformation);
    }
}

void Scene::activerForme(QGraphicsItem *forme)
{
    this->forme = forme;
    clearSelection();
    forme->setSelected(true);
}

```

## x TRAVAUX PRATIQUES EN AUTONOMIE

Afin de maîtriser plus complètement ces notions sur les dessins vectoriels, je vous propose de retravailler sur le logiciel de traitement d'image. Pour cela, nous changeons l'ergonomie afin que l'histogramme devienne un élément interactif qui nous permet de régler directement les différents niveaux, comme nous le retrouvons systématiquement sur les logiciels du commerce.



| Émetteur          | Signal          | Receveur       | Slot                 |
|-------------------|-----------------|----------------|----------------------|
| photoOriginale    | envoi(QPixmap*) | photoTraite    | prendPhoto(QPixmap*) |
| actionOuvrir      | triggered()     | photoOriginale | ouvrirPhoto()        |
| actionEnregistrer | triggered()     | photoTraite    | enregistrerPhoto()   |

| Objet             | Classe      |
|-------------------|-------------|
| Principal         | QMainWindow |
| centralWidget     | QWidget     |
| onglets           | QTabWidget  |
| traitement        | QWidget     |
| photoTraite       | Vue         |
| originale         | QWidget     |
| photoOriginale    | Originale   |
| mainToolBar       | QToolBar    |
| actionOuvrir      | QAction     |
| actionEnregistrer | QAction     |

| Propriété  | Valeur                              |
|------------|-------------------------------------|
| QObject    |                                     |
| objectName | actionOuvrir                        |
| QAction    |                                     |
| checkable  | <input type="checkbox"/>            |
| checked    | <input type="checkbox"/>            |
| enabled    | <input checked="" type="checkbox"/> |
| icon       | ouvrir.png                          |
| text       | Ouvrir                              |
| iconText   | Ouvrir                              |

x Nous trouvons deux onglets sur notre logiciel. Le premier, partie cachée, représentant la photo originale, peut être réalisée de façon classique, c'est-à-dire en créant une sous-classe d'un composant **QWidget** et en redéfinissant simplement la méthode **paintEvent()**.

x Le deuxième onglet correspond à l'affichage de l'image traitée ainsi que la visualisation de la courbe des niveaux. Il est plus judicieux de concevoir cette partie là sous forme vectorielle (vue, calque, éléments graphiques) pour que l'histogramme soit, par exemple, plus facile à déplacer.

x Pour vous aidez, il serait également souhaitable de créer une nouvelle classe qui s'occupe uniquement de la gestion de chacun des triangles correspondant aux réglages des niveaux. Ces objets sont issus d'une classe qui hérite de **QGraphicsPolygonItem**.