Table des matières

HTML5 et la balise Canvas

avertissement

La fiche suppose que l'on travaille avec HTML5. Le navigateur utilisé doit être récent : Firefox >3, Chrome >4, IE > 9, Opera > 9 et Safari >3. On utilise JavaScript : suivant en cela une bonne habitude, on suppose disponible **jQuery**, qui masque les différences entre les implémentations du DOM. Ce framework s'avère particulièrement efficace pour ce qui est de la gestion des événements, de la sollicitation des feuilles de style ou de l'utilisation de formulaires et d'Ajax.

Le schéma de page web de base est donc le suivant :

```
<!DOCTYPE html>
<html><!-- fichier can00_xxx.html -->
<head>
    <meta charset="utf-8" />
    <title>essais de canvas</title>
    <style>
        #id_canvas {
            border : 2px black solid ;
            margin : 20px;
        }
    </style>
    <script src="./js/jquery-2.1.0.min.js"></script>
    <script>
        initialisation = function () {
            alert ('ok tout va bien !');
        }
        $(document).ready (initialisation) ;
    </script>
</head>

<body>
    <div>
        <canvas id="id_canvas" width = "500" height = "400">
            problème de création : vérifier votre navigateur
        </canvas>
    </div>
</body>
</html>
```

Comme il s'agit ici de pages d'illustration, les feuilles de style sont incluses dans la page, tout comme les scripts JavaScript. Les images sont dans le répertoire img.

La structure de site suit le modèle suivant :

```
├── canvas
│   ├── can00.html
│   ├── can01.html
│   ├── img
│   │   ├── arc.png
│   │   ├── arc.svg
│   │   └── tuxnpdc.png
│   ├── css
│   └── js
│       └── jquery-2.1.0.min.js
```

1. la balise canvas

1.1. syntaxe et attributs

déclaration : `<canvas>`

balise de fermeture : `<canvas>`

attributs :

* les quatre attributs classiques des éléments affichables : **`id, class, style, title`**

* **`width, height`** : dimensions du **`canvas`** en pixels. ex **`width="500".`** Par défaut les valeurs sont 100x200 px. Attention à cette question de dimension ! Elle ne se confond pas avec la dimension d'affichage qui elle est donnée par la feuille de style CSS ; la dimension d'affichage se confond avec la dimension du canvas s'il n'y a pas redéfinition dans la feuille de style, exactement comme une image sur fichier bitmap. La question relève des CSS. **Fixer les attributs est nécessaire si on désire autre chose que la dimension par défaut**.

On rappelle que pour ramener la fenêtre du navigateur à 100 % il suffit de faire **Ctrl-0** (contrôle + zéro). C'est important pour la mise au point des scripts, de fonctionner sans zoom.

Les attributs de style classiques s'appliquent : **`border, background-color, margin`** etc.

display : **`inline`** par défaut

Comme tous les éléments usuels, le **`canvas`** est «transparent» sauf si on lui impose une couleur de fond opaque.

1.2. le contexte

page prête

L'exécution du fichier HTML insère le canvas dans le DOM comme un nœud de celui-ci. On peut donc le récupérer par une des méthodes classiques selon que l'on utilise **jQuery** ou pas. Mais auparavant, il convient de s'assurer que la page est chargée. Pour mémoire, le procédé ancien est l'utilisation du **`onload`** :

```
window.onload = function () { ... }
```

On ne revient pas ici sur les inconvénients de cette démarche et c'est **jQuery** qui apporte la solution :

```
$(document).ready (function () { ... }) ;
```

récupération du nœud du DOM

On oublie le procédé direct par **`getElementById()`**, pour s'en remettre à **jQuery** :

```
monCanvas = $("#id_canvas").get(0)
```
ou
```
monCanvas = $("#id_canvas")[0]
```

On rappelle que la méthode **`jquery(`*`sélecteur`*`)`** (alias : **`$(`*`sélecteur`*`)`**) retourne un objet **JavaScript** de type **array** (tableau). Il est recommandé d'utiliser la méthode **`get()`** pour récupérer le nœud.

le contexte

Les méthodes qui permettent de travailler avec l'objet **`canvas`** n'appartiennent pas au nœud **`canvas`** lui-même mais à l'un de ses attributs : son contexte, retourné par la méthode **`getContext("2d")`**. Plus précisément, l'objet contexte retourné est assimilable à un objet qui comporte un calque de dessin et les méthodes ou attributs pour dessiner dessus.

L'argument **`"2d"`** est obligatoire, même si pour l'instant c'est la seule implémenté de façon utilisable.

On va donc trouver obigatoirement une ligne de la forme :

```
monContexte : monCanvas.getContext("2d")
```

le code proposé

```
<!-- fichier can01_xxx.html -->
        initialisation = function () {
            var monCanvas, monContexte ;
            monCanvas = $("#id_canvas").get(0) ;
            monContexte = monCanvas.getContext("2d") ;
            alert ('ok tout va bien !') ;
            /* voir (monContexte);*/
        }
```

note : les variables **`monCanvas`** et **`monContexte`** sont locales. On peut évidemment aussi travailler avec des variables globales.

1.3. analyse de context

```
<!-- fichier can01_xxx.html -->
        voir = function (noeud) {
            var arbre = "", tableau ;
            for (var p in noeud) {
                arbre = arbre + p + "    --->   " + typeof noeud[p] +";;";
            }
            tableau = arbre.split(";;").sort() ;
            alert (tableau.join("\n"));
        }
```

Le contexte est un objet **JavaScript** ; on peut donc en faire l'analyse structurelle, en l'occurrence, chercher ses attributs et leur type. On aurait pu faire la même chose avec le nœud **`canvas`**, mais c'est d'un maigre intérêt, sauf à y trouver la fonction **`getContext()`** comme attribut. Pour le détail de la méthode, voir les textes déjà rédigés sur **JavaScript**. On rappelle que les attributs d'un objet peuvent être vus comme des chaînes (la variable **`p`**), la valeur de l'attribut étant **`noeud[p]`**. La méthode de chaîne **`split()`** transforme une chaîne en tableau qui peut alors être trié (**`sort()`**) ; et **`join()`** transforme le tableau en chaîne.

Pour appeler la fonction **`voir`**, insérer l'appel dans la fonction initialisation :

```
<!-- fichier can01_xxx.html -->
        initialisation = function () {
            var monCanvas, monContexte ;
            monCanvas = $("#id_canvas").get(0) ;
            monContexte = monCanvas.getContext("2d") ;
            voir (monContexte) ;
            alert ('ok tout va bien !') ;
            /* voir (monContexte);*/
        }
```

Comme dans la suite les fonctions définies pour las tracés sont appelées selon la même démarche, on ne la rappellera pas. Pour la clarté de l'exposé, la chaîne a été transformée en tableau dans LibreOffice :

arc ---> function	moveTo ---> function
arcTo ---> function	mozCurrentTransform ---> object
beginPath ---> function	mozCurrentTransformInverse ---> object
bezierCurveTo ---> function	mozDash ---> object
canvas ---> object	mozDashOffset ---> number
clearRect ---> function	mozFillRule ---> string
clip ---> function	mozImageSmoothingEnabled ---> boolean
closePath ---> function	mozTextStyle ---> string
createImageData ---> function	putImageData ---> function
createLinearGradient ---> function	quadraticCurveTo ---> function
createPattern ---> function	rect ---> function
createRadialGradient ---> function	restore ---> function
drawImage ---> function	rotate ---> function
fill ---> function	save ---> function
fillRect ---> function	scale ---> function
fillStyle ---> string	setTransform ---> function
fillText ---> function	shadowBlur ---> number
font ---> string	shadowColor ---> string
getImageData ---> function	shadowOffsetX ---> number
globalAlpha ---> number	shadowOffsetY ---> number
globalCompositeOperation ---> string	stroke ---> function
isPointInPath ---> function	strokeRect ---> function

isPointInStroke ---> function	strokeStyle ---> string
lineCap ---> string	strokeText ---> function
lineJoin ---> string	textAlign ---> string
lineTo ---> function	textBaseline ---> string
lineWidth ---> number	transform ---> function
measureText ---> function	translate ---> function
miterLimit ---> number	

On peut remarquer une zone d'objets en **moz...** Ils sont spécifiques à Firefox ; on ne les retrouve pas des Chrome par exemple. Par contre beaucoup des noms sont assez évocateurs : **arc, rect, rotate, line, stroke** (trait), **Point, font, fill, shadow** (ombre), **Path** (chemin), **text**...

2. premiers tracés

2.1. coordonnées

Le système de coordonnées est le même que dans tous **les systèmes graphiques bitmap** (Python/Tkinter, HTML, Gimp, mais pas Inkscape qui est vectoriel). L'unité est le pixel (référence à l'écran), et elle n'est pas précisée dans les mesures comme toujours en HTML. L'origine est en haut et à gauche. La zone de travail est rectangulaire. La numérotation des pixels commence à 0.

2.2. tracé de ligne

méthodes

* **beginPath()** : nouveau tracé ;

* **closePath()** : fin de tracé ;

* **moveTo()** : tracé, déplacement plume levée au point donné en argument ;

* **lineTo()** : tracé, déplacement plume baissée du point actuel au point passé en argument ;

* **stroke()** : concrétisation de tracé de ligne uniquement ;

* **fill()** : concrétisation de remplissage ; la forme pleine est obtenue en complétant le tracé ;

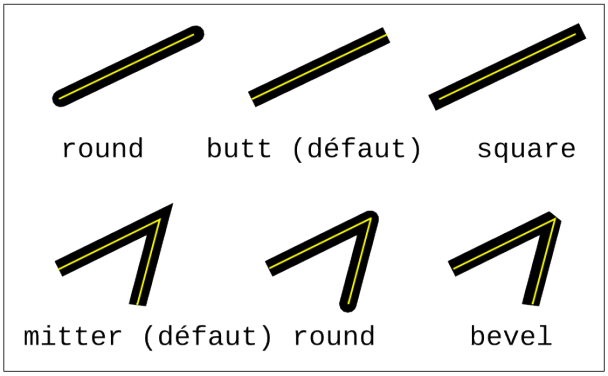
note : le tracé d'un trait épais est à cheval sur le ligne théorique ; lors d'un remplissage, la moitié de la ligne est cachée si on remplit après avoir tracé ; dans le cas contraire, la partie remplie est amputée d'autant. Lors d'un tracé de figure croisée, il peut y avoir un décrochage de contour.

attributs

* **lineWidth** : valeur numérique (pas d'unité explicite) ; largeur de tracé ;

* **lineJoin** : **mitter, round**, **bevel**, pour coins allongé pour faire un angle, arrondis, biseauté ;

* **lineCap** : **round, butt, square** selon que l'extrémité d'une ligne est arrondi, «au carré», dépassant.

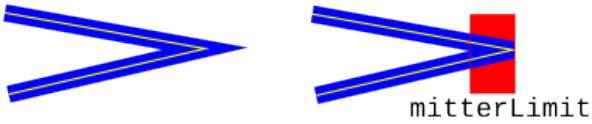


* **fillStyle** : chaîne, couleur de remplissage ; mêmes codes qu'en CSS. Attention, bien coller **rgb et parenthèse**, comme toujours en CSS. L'échelle des composants est 0-255 ; **rgba** implique un canal alpha ; l'échelle de ce canal est 0-1, pour aller de la transparence à l'opacité.

* **strokeStyle** : couleur de trait.

* **mitterLimit** : lorsque l'on est dans le mode **mitter**, il se peut que le raccord soit beaucoup trop grand ; imaginer un angle très aigu sur un rait épais ; il convient alors de retenir localement un mode comme **bevel**.

L'attribut **mitterLimit** indique pour quels raccord le changement doit être opéré. La valeur est l'épaisseur maximum de l'onglet.



* **save()** et **restore()** : ces méthodes permettent de sauvegarder les attributs qui précèdent et de les réutiliser ultérieurement.

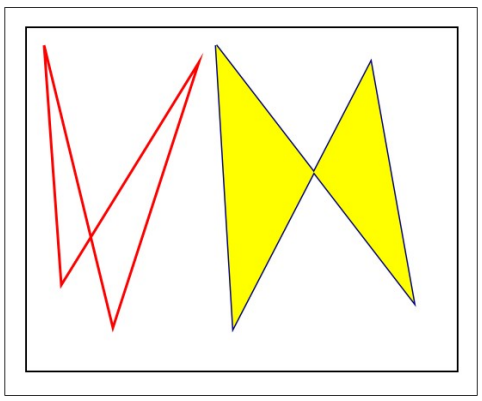
* **isPointInPath(x,y)** : retourne true si le point passé en paramètre appartient à un path ouvert.

* **isPointInStroke (x, y)** : le point passé en paramètre appartient à un tracé.

* **clip()** : une fois défini un contour (path, rectangle ou autre), la commande **clip()** conduit réduire l'affichage de la commande suivante au contour précédemment affiché.

2.3. exemple

```
/* fichier can02xxx.html */
       traceLigne  = function (contexte) {
       /* traceLigne (monContexte) */
           contexte.beginPath() ;          /* ouverture du tracé     */
           contexte.strokeStyle = "red" ;  /* couleur du tracé       */
           contexte.lineWidth = 3 ;        /* épaisseurr du tracé    */
           contexte.moveTo (20, 20) ;      /* point de départ        */
           contexte.lineTo (40, 300) ;     /* ligne                  */
           contexte.lineTo (200, 40) ;
           contexte.lineTo (100, 350) ;
           contexte.lineTo (20, 20) ;
           contexte.stroke() ;             /* concrétisation du trait */
           contexte.closePath() ;          /* tracé terminé           */
       }
       remplitLigne  = function (contexte) {
       /* remplitigne (monContexte) */
           contexte.beginPath() ;          /* ouverture du tracé     */
           contexte.strokeStyle = "#000080" ;
           contexte.fillStyle = "rgb(255, 255, 0)" ;
           contexte.lineWidth = 3 ;
           contexte.moveTo (220, 20) ;
           contexte.lineTo (240, 350) ;
           contexte.lineTo (400, 40) ;
           contexte.lineTo (450, 320) ;;
           contexte.lineTo (220, 20) ;
           contexte.stroke() ;
           contexte.fill() ;               /* remplissage             */
           contexte.closePath() ;          /* tracé terminé           */
       }
```



3. formes utiles

3.1. rectangle

Les méthodes sont :

* **`fillRect()`** : affiche un rectangle plein ;

* **`clearRect()`** : efface le rectangle ;

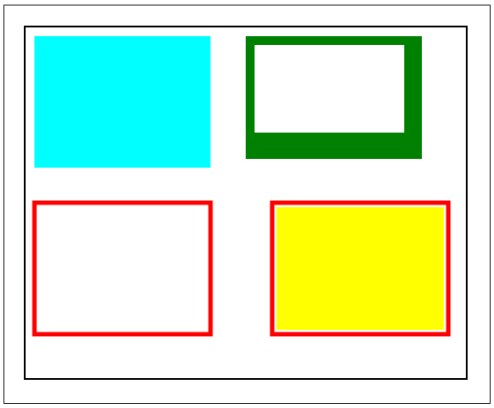
* **`strokeRect()`** : trace un contour rectangulaire ;

* **`rect()`** : définit un rectangle, que l'on peut tracer et remplir par la suite ; sert également pour **`clip()`**.

On n'est pas contraint de remplir un rectangle tracé par **`fill()`**. On peut également tracer un rectangle plein à l'intérieur d'un tracé antérieur. Quant on définit le rectangle/trait, les dimensions correspondent à un rectangle théorique ; le trait est à cheval sur le dessin théorique.

La définition se fait sur le mode (x, y, w, h) : coordonnées d'un coin ; offsetx et offsety après le coin opposé (w et h peuvent être négatifs). Dans la majorité des cas, il s'agit du coin haut/gauche et offsetx, offsety sont positifs et égaux à la largeur et la hauteur du rectangle.

```
/* fichier can03xxx.html */
      /* traceRectangle (monContexte) */
       traceRectangle = function (ctx) {
           /* rectangle plein */
           ctx.fillStyle = "cyan" ;
           ctx.fillRect (10, 10, 200, 150) ;
           /* rectangle ligne */
           ctx.lineWidth = 5 ;
           ctx.strokeStyle = "red" ;
           ctx.strokeRect (10, 200, 200, 150) ;
           /* vider selon un rectangle */
           ctx.fillStyle = "green" ;
           ctx.fillRect (250, 10, 200, 140) ;
           ctx.clearRect (260, 20, 170, 100) ;
           /* bord à remplir */
           var ep = 5 ;
           ctx.lineWidth = ep ;
           ctx.strokeStyle = "red" ;
           ctx.strokeRect (280, 200, 200, 150) ;
           ctx.fillStyle = "yellow" ;
           ctx.fillRect (280+ep, 200+ep, 200-ep-ep, 150-ep-ep) ;
       }
```



3.2. arcTo

La méthode arc n'est pas très intuitive ; elle s'avère utile pour les raccordements arrondis dans les tracés, comme par exemple le tracé d'un rectangle à coins adoucis (deuxième exemple).

la syntaxe :

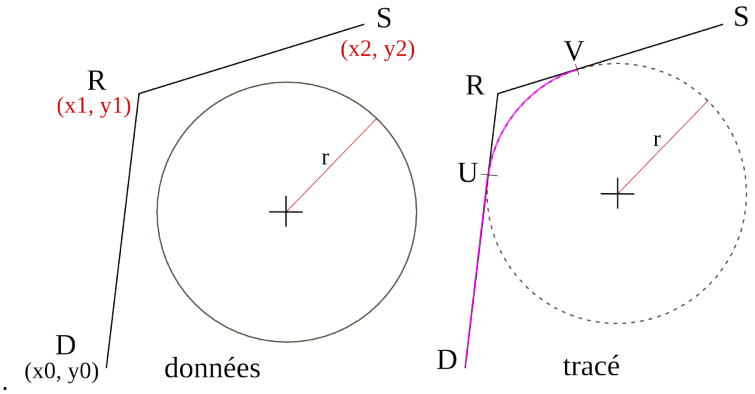
`arcTo (x1, y1, x2, t2, r)` avec (xi, y1), (x2, y2) comme coordonnées d'un point du **`canvas`** et un nombre positif, rayon du cercle de raccordement.

le fonctionnement

L'**arcTo** suppose un premier point D (x0, y0) qui sert de point de départ au tracé ; ce point est l'extrémité d'une ligne tracée par **lineTo** ou **arcTo**, ou obtenu par **moveTo**. Les deux points passés en argument sont des points référents. la paramètre **r** permet de définir un cercle, mais ne précise pas son emplacement.

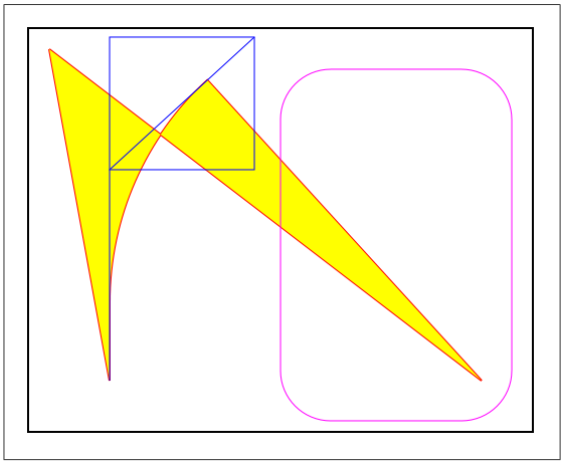
On joint le premier point référent R (x1, y1) à D (x0, y0) et S (x2, y2). On place alors le cercle de façon qu'il soit tangent à ces deux droites, respectivement aux points U et V.

Le tracé part de D, va en ligne droite en U, puis sur le cercle en V



illustration

```
/* can04xxx.html*/
        traceArc  = function (ctx) {
            /* traceLigne (moncontexte) */
            ctx.beginPath() ;          /* ouverture du tracé     */
            ctx.strokeStyle = "red" ;  /* couleur du tracé       */
            ctx.fillStyle = "yellow" ;
            ctx.lineWidth = 2 ;        /* épaisseurr du tracé    */
            ctx.moveTo (20, 20) ;      /* point de départ        */
            ctx.lineTo (80, 350) ;
            ctx.arcTo (80, 140, 224, 8, 300) ;
            ctx.lineTo (450,350) ;
            ctx.lineTo (20, 20) ;
            ctx.stroke() ;             /* concrétisation du trait */
            ctx.fill() ;
            ctx.closePath() ;          /* tracé terminé           */
            /*matérialisation des données référentes*/
            /* les points (par un rectangle) */
            ctx.lineWidth = 1 ;
            ctx.strokeStyle = "blue" ;
            ctx.strokeRect (80, 140, 144, -132) ;
            /* les lignes */
            ctx.beginPath() ;
            ctx.moveTo (80, 140) ;      /* point de départ        */
            ctx.lineTo (80, 350) ;
            ctx.moveTo (80, 140) ;      /* point de départ        */
            ctx.lineTo (224, 8) ;
            ctx.stroke() ;
            ctx.closePath() ;
        }
        traceCoinsRonds = function (ctx, x0, y0,offx, offy, r) {
            /* traceCoinsRonds (monContexte, 250, 40, 230, 350, 50) ; */
            ctx.strokeStyle = "magenta" ;
            ctx.beginPath() ;
            ctx.moveTo (x0, y0+r) ;
            ctx.arcTo (x0, y0+offy, x0+offx, y0+offy, r) ;
            ctx.arcTo (x0+offx, y0+offy, x0+offx, y0, r) ;
            ctx.arcTo (x0+offx, y0, x0, y0, r) ;
            ctx.arcTo (x0, y0, x0, y0+offy, r) ;
            ctx.stroke() ;
            ctx.closePath() ;
        }
```



3.3. cercles et arcs

La syntaxe

La méthode s'appelle arc et elle prend 6 arguments :

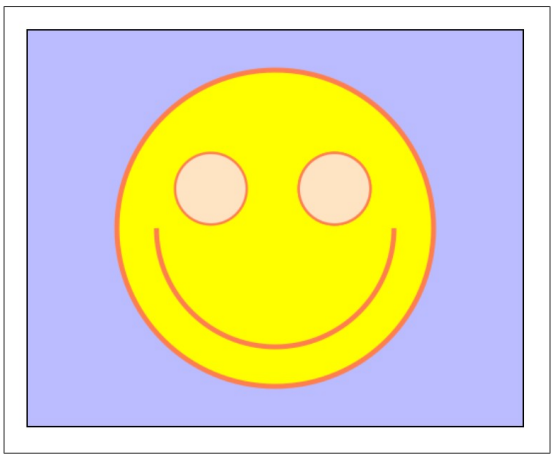
`arc (centreX, centreY, rayon, angleDébut, angleFin, sens)`

Si on veut un cercle au complet faire angleDébut = 0, angleFin = Math.PI*2 ;

Les angles sont en radians. Si **`sens`** a la valeur **`true`**, le cercle est décrit dans le sens des aiguilles d'une montre ; c'est la valeur par défaut, qui peut être omise. Les angles sont mesurés à partir de l'horizontale, de gauche à droite.

exemple

```
/* fichier can05xxx.html *//
        traceFigure = function (ctx) {
            /* traceFigure (moncontexte) */
            /* tracé visage */
            ctx.beginPath() ;
            ctx.strokeStyle = "coral" ;
            ctx.fillStyle = "yellow" ;
            ctx.lineWidth = 5 ;
            ctx.arc (250, 200, 160, 0, 2*Math.PI) ;

            ctx.fill() ;
            ctx.stroke() ;  /* attention : après fill() */
            ctx.closePath() ;
            /* les tracés yeux */

            ctx.strokeStyle = "coral" ;
            ctx.fillStyle = "bisque" ;

            ctx.beginPath() ;
            ctx.arc (185, 160, 35, 0, 2*Math.PI) ;
            ctx.stroke() ;
            ctx.fill() ;
            ctx.closePath() ;
            ctx.beginPath() ;
            ctx.arc (310, 160, 35, 0, 2*Math.PI) ;
            ctx.stroke() ;
            ctx.fill() ;
            ctx.closePath() ;
            /* tracé bouche*/
            ctx.beginPath() ;
            ctx.arc (250, 200, 120, 0, Math.PI) ;
            ctx.stroke() ;
            ctx.closePath() ;
        }
```



3.4. courbe quadratique

Une courbe quadratique est définie par trois points : départ, arrivée, contrôle

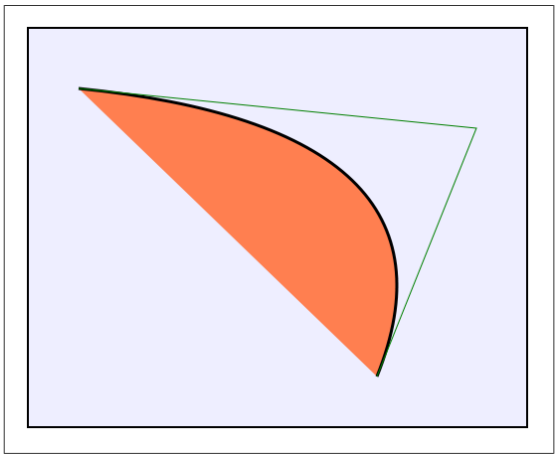
Le point de départ est un point existant.

syntaxe

`quadraticCurveTo (ctrlx, ctrly, finalx finnaly)`

exemple

```
/* fichier can06xxx.html */
       traceQuadra = function (ctx) {
           /* traceQuadra (moncontexte) */
           /* tracé courbe */
           ctx.beginPath() ;
           ctx.strokeStyle = "black" ;
           ctx.fillStyle = "coral" ;
           ctx.lineWidth = 3 ;
           ctx.moveTo (50,60) ;
           ctx.quadraticCurveTo (450, 100, 350, 350) ;
           ctx.fill() ;
           ctx.stroke() ;
           ctx.closePath() ;
           /* tracé tangentes de contrôle*/
           ctx.beginPath() ;
           ctx.strokeStyle = "green" ;
           ctx.lineWidth = 1 ;
           ctx.moveTo (50,60) ;
           ctx.lineTo (450, 100) ;
           ctx.lineTo (350, 350) ;
           ctx.stroke() ;
           ctx.closePath() ;
       }
```



Le tracé des lignes joignant les début et fin de la courbe au point de contrôle fait apparaître que ce sont les tangentes à la courbes. D'un point de vue mathématique, on a affaire à **un segment de la parabole** définie par les deux points extrêmes et les deux tangentes.

3.5. courbe de Bézier

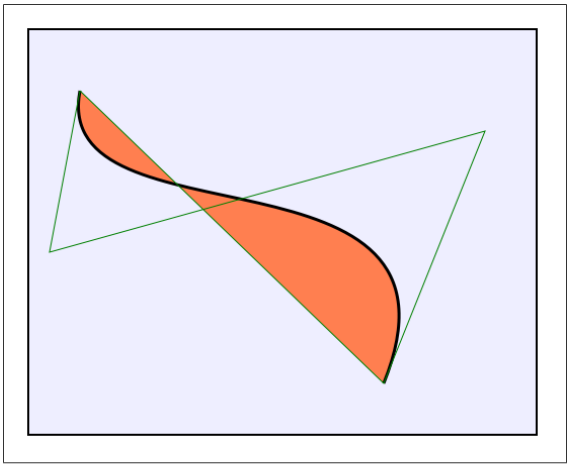
Une courbe de Bézier est caractérisée par ses points de contrôle et son point terminal. Son point initial est la position actuelle du tracé.

syntaxe pour une cubique (2 points de contrôle)

```
bezierCurveTo(ctrlx1, ctrly1, ctrex2, ctrly2, finalx, finaly)
```

exemple

```
/* fichier can07xxx.html */
       traceBezier = function (ctx) {
           /* traceBezier (moncontexte) */
           /* tracé courbe */
           ctx.beginPath() ;
           ctx.strokeStyle = "black" ;
           ctx.fillStyle = "coral" ;
           ctx.lineWidth = 3 ;
           ctx.moveTo (50,60) ;
           ctx.bezierCurveTo (20, 220, 450, 100, 350, 350) ;
           ctx.fill() ;
           ctx.stroke() ;
           ctx.closePath() ;

           /* tracé tangentes de contrôle*/
           ctx.beginPath() ;
           ctx.strokeStyle = "green" ;
           ctx.lineWidth = 1 ;
           ctx.moveTo (50,60) ;
           ctx.lineTo (20, 220) ;
           ctx.lineTo (450, 100) ;
           ctx.lineTo (350, 350) ;
           ctx.lineTo (50, 60) ;
           ctx.stroke() ;
           ctx.closePath() ;
       }
```



4. les textes

4.1. tracé de texte

Le tracé de texte est assez semblable au tracé de rectangle : on dispose de deux méthodes qui permettent de réaliser un texte plein ou un contour. Mais il faut définir les caractéristique du tracé : fonte utilisée, alignement horizontal (gauche, droite), alignement vertical (ligne de base). On dispose également d'une méthode de mesure d'encombrement, qui permet de connaître le rectangle encadrant, et autorise un positionnement en fonction de la taille de ce rectangle.

4.2. les attributs, les méthodes et leur syntaxe

méthodes

* **`fillText(texte, x, y)`** : x et y sont les coordonnées de la ligne de base ;

* **`strokeText (texte, x, y)`**

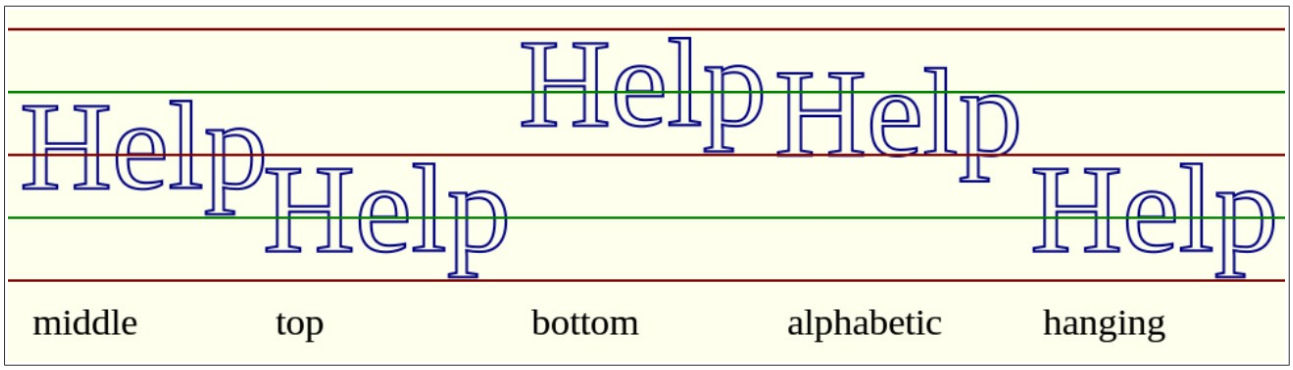
* **`measureText (texte)`** : retourne un objets JavaScript avec un attribut **`width.`** C'est actuellement le seul argument défini sur Firefox et Chrome.

attributs

* **`font`** : prend une chaîne de caractères, avec taille et nom de fonte ; ex **`"30pt Arial"`** :

* **`textAlign`** : prend les valeurs chaînes **`left`**, **`right, center`** ; **`left`** par défaut ;

* **`textBaseLine`** : prend les valeurs chaîne **`top, bottom, middle, alphabetic, hanging`** ; attention ; **`middle`** par défaut. Dans l'illustration qui suit, la fonte 100px ; les lignes sont espacées de 50px.



4.3. exemple

```
/* fichier can08xxx.html */
        traceTexte = function (ctx) {
            /* traceTexte (moncontexte) */
            ctx.font = "50pt 'Liberation Serif', serif" ;
            ctx.textBaseline = "bottome" ;
            ctx.textAlign = "left" ;
            ctx.fillText ("AmiPoste MSE", 30, 70) ;
            ctx.strokeText ("AmiPoste MSE", 30, 150) ;
            var  rectTexte = ctx.measureText ("AmiPoste MSE") ;
            var X = (30+rectTexte.width) / 2 ;

            ctx.font = "80pt 'Liberation Serif', serif" ;
            ctx.strokeStyle = "blue" ;
            ctx.lineWidth = 2 ;
            ctx.fillStyle = "yellow" ;
            ctx.textBaseline ="top" ;
            ctx.textAlign = "center" ;
            ctx.fillText ("JEAN", X, 200) ;
            ctx.strokeText ("JEAN", X , 200) ;
         }
```

5. image

5.1. image dans un canvas

Une image, au sens de JavaScript est utilisable pour un affichage dans un **`canvas`**. Le problème est de disposer d'un objet image. Cette question ne relève pas de **`canvas`**, mais du DOM classique, celui des débuts de l'utilisation d'images sur le web et de leur manipulation en JavaScript. On rappelle donc des données étudiées et discutées par ailleurs (JavaScript et le DOM).

L'objet **`Image`** a les propriétés spécifiques des éléments **`img du HTML`** : **`src`**, pour la destination, **`complete`**, drapeau qui indique le chargement de l'image, **`width, height, fileSize`**. L'initialisation de l'objet **`Image`** se fait en affectant une **url** à sa propriété **`src`**.

Une variable image est créée par le constructeur **`Image()`** . Mais le chargement d'une image est asynchrone, et se produit dès que l'on a défini sa source. On commettrait une erreur en utilisant une image non entièrement constituée ; heureusement, la fin du chargement crée un événement **`load`** sur l'objet image. Il faut donc assujettir la poursuite du travail à la réalisation de cet événement. Classiquement on le fait par l'attribut **`onload`** ; mais la difficulté due à une gestion différente suivant les navigateurs conduit à utiliser **jQuery**.

La balise **`canvas`** accepte une grande variété de formats d'images : png, jpeg, gif, svg. Mais comme destination, on peut utiliser un élément image du DOM (défini dans une balise img) ou un autre **`canvas`** (c'est une manière de simuler les calques).

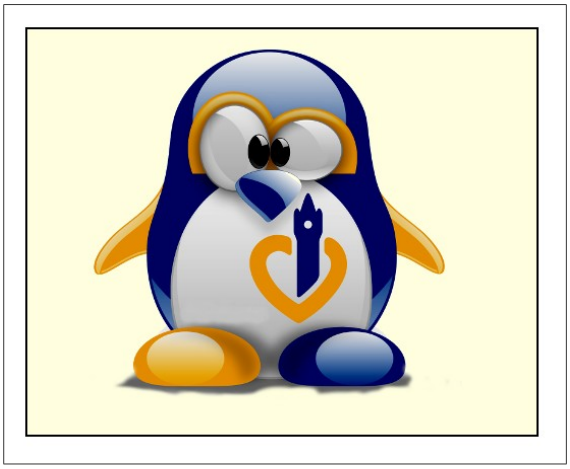
5.2. syntaxe

L'affichage d'image se fait par la méthode **`drawImage()`** :

`drawImage (image, x, y)` : x, y réfèrent le sommet gauche de l'image.

5.3. exemple

```
/* fichier can09xxx.html */
       traceImagePng = function (ctx) {
           var tuxnpdc = new Image() ;
           tuxnpdc.src = "./img/tuxnpdc.png" ;
           $(tuxnpdc).bind ("load", function (event) {
               ctx.drawImage (tuxnpdc, 50, 20) ;
           }) ;
       }
```



mise en garde : il faut faire attention si on utilise le gestionnaire d'événement **`onload`** ; en effet, s'il est implémenté sur tous les navigateurs, il n'est pas paramétré de la même façon avec IE et Firefox. **jQuery** peut être utilisé pour passer des paramètres utilisateur au gestionnaire. Dans le présent exposé, on passa par **jQuery** pour gérer l'événement.

6. les méthode de remplissage

6.1. rappels

On a vu dans la section relative au tracé que le style des traits pouvait être modifiée (**`lineWidth, lineCap, lineJoin`**). Les traits enclosent des surfaces qui ont plusieurs modes de remplissage ; on a déjà rencontré la valeur «couleur uniforme» pour **`fillStyle`**. Il en existe deux autres ; le «pattern/motif» et le dégradé.

6.2.la méthode createPattern()

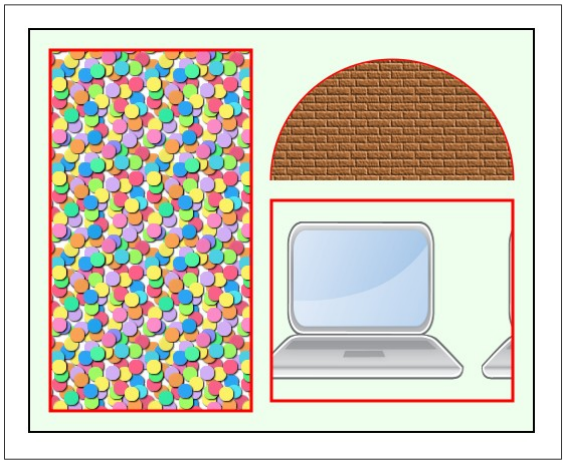
`La méthode` **`createPattern()`** `crée un motif de remplissage. La` méthode rappelle le **`background-image`** en CSS. Elle lui emprunte d'ailleurs ses attributs : **`no-repeat, repeat-x, repeat-y, repeat.`**

syntaxe

```
createPattern (image, "repeat")
```

Mais le fonctionnement est un peu particulier et déroutant. En effet, le motif appartient au canevas et pas à la courbe à remplir. Il n'y a pas grand problème avec **`repeat`** : le fond est pavé par le motif et il est «clippé» sur le fond, dans la surface limitée par le tracé,. Mais, dans les autres cas, sauf à avoir une figure calée en haut et à gauche, une bonne partie, voir la totalité du pavage peut échapper au morceau pavé.

exemple



```
/* fichier can10xxx.html */
        faireConfetis = function (ctx){
            var im = new Image() ;
```

```
            im.src = "./img/confetis.jpg";
            $(im).bind ("load",  function () {
                var confetis = ctx.createPattern (im, "repeat");
                ctx.fillStyle = confetis ;
                ctx.fillRect (20, 20, 200, 360) ;
                ctx.strokeRect (20, 20, 200, 360) ;
            });
        }
        faireBriques = function (ctx){
            var image ;
            im = new Image() ;
            im.src = "./img/briques.png";
            $(im).bind ("load", function () {
                var briques = ctx.createPattern (im, "repeat");
                ctx.fillStyle = briques;
                ctx.beginPath () ;
                ctx.arc (360, 150, 120, 0, Math.PI, true) ;
                ctx.stroke() ;
                ctx.fill() ;
                ctx.closePath() ;
            });
        }
        faireOrdi = function (ctx){
            var im ;
            im = new Image() ;
            im.src = "./img/laptop.png"

            $(im).bind ("load", function () {
                var ordinat = ctx.createPattern (im, "repeat");
                ctx.fillStyle = ordinat ;
                ctx.fillRect (240, 170, 240, 200) ;
                ctx.strokeRect (240, 170, 240, 200) ;
            });
        }

        faireBordure = function (ctx) {
            var ep = 3 ;
            ctx.lineWidth = ep ;
            ctx.strokeStyle = "red" ;
        }
```

6.3. dégradés

les méthodes

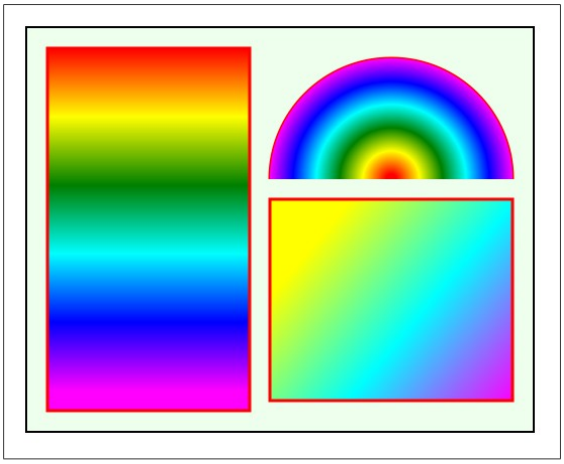
* **`createLinearGradient()`** : un dégradé linéaire possède un axe ; c'est lui qui est donné en paramètre sous forme DX, SY, AX, AY (départ x, y et arrivée x, y)

* **`createRadialGradient()`** : le gradient radial est donné par deux cercles sous la forme : centredX, centreDY, Drayon, et centreAX, centreAY, Arayon.

l' attribut

L'attribut s'appelle **`addColorStop().`** Il peut être affecté autant de fois qu'on le veut ; l'objet «dégradé» empile les valeurs successives. L'attribut a deux paramètres : le premier est flottant entre 0 et 1 ; 0 correspond à la valeur de départ et 1 à celle d'arrivée.

exemple



```
/* fichier can11xxx.html */
       faireGradient1= function (ctx){
           var grd = ctx.createLinearGradient (100,20, 100, 360);
           grd.addColorStop (0, "red") ;
           grd.addColorStop (0.2, "yellow") ;
           grd.addColorStop (0.4, "green") ;
           grd.addColorStop (0.6, "cyan") ;
           grd.addColorStop (0.8, "blue") ;
           grd.addColorStop (1, "magenta") ;
           ctx.fillStyle = grd ;
           ctx.fillRect (20, 20, 200, 360) ;
           ctx.strokeRect (20, 20, 200, 360) ;
       }
       faireGradient2 = function (ctx){
           var grd = ctx.createRadialGradient (360, 150, 5, 360, 150, 120);
           grd.addColorStop (0, "red") ;
           grd.addColorStop (0.2, "yellow") ;
           grd.addColorStop (0.4, "green") ;
           grd.addColorStop (0.6, "cyan") ;
           grd.addColorStop (0.8, "blue") ;
           grd.addColorStop (1, "magenta") ;
           ctx.fillStyle = grd ;
           ctx.beginPath () ;
           ctx.arc (360, 150, 120, 0, Math.PI, true) ;
           ctx.stroke() ;
           ctx.fill() ;
           ctx.closePath() ;
       };

       faireGradient3 = function (ctx){
           var grd = ctx.createLinearGradient (240, 170, 480, 370);
           grd.addColorStop (0.2, "yellow") ;
           grd.addColorStop (0.6, "cyan") ;
           grd.addColorStop (1, "magenta") ;
           ctx.fillStyle = grd ;
           ctx.fillRect (240, 170, 240, 200) ;
           ctx.strokeRect (240, 170, 240, 200) ;
       };
```

6.4. interaction entre deux images

Dans les logiciels d'édition (Gimp, Paint Shop Pro etc) qui manipulent les calques, on connaît l'interaction entre calques (opérations logiques, arithmétiques, masques etc.). L'objet **`canvas`** n'a pas de calques. À un niveau de l'élaboration d'un dessin, on a l'état actuel des pixels qui est la **destination** de ce qui va suivre immédiatement. La première commande qui suit définit le dessin **source**. L'interaction se fait entre ces deux dessins. Elle est commandée par un appel de la méthode **`globalCompositionOperation()`**

syntaxe

`globalCompositionOperation(`*`chaîne paramètre`*`)`

paramètres

* le nouveau contenu (source s'impose :

source-over : la source s'impose

source-in : la source s'impose où il y avait des pixels de dessin destination qui disparaît

source-out : la source s'affiche en dehors des pixels de dessin destination qui disparaît

source-atop : la source s'impose où il y avait des pixels de dessin destination

* l'ancien contenu (destination) s'impose :

destination-over : la source est placée en arrière-plan

destination-in : la source est affiché uniquement en remplacement des pixels de la destination

destination-out : n'affiche pas la source et qui cache la destination

destination-atop : n'affiche dans la source que ce qui est dans destination, le reste disparaissant

* les deux éléments sont équivalents :

lighter : prend le plus "clair" des deux (éventuellement transparent) comme un **et** logique

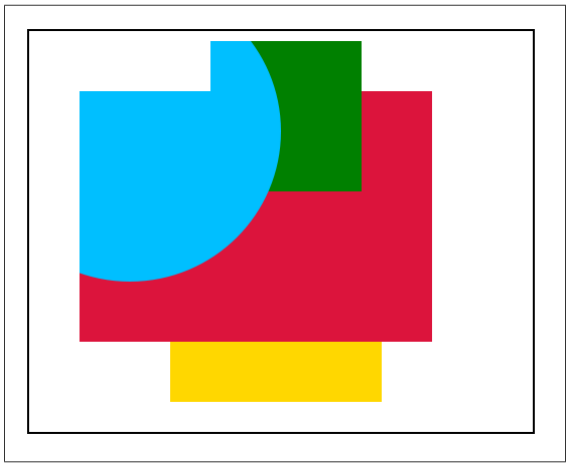
darker : prend le plus sombre

copy : ne garde que le source

xor : un xor est effectué pixel par pixel entre source et destination.

exemple

```
/* fichier can012xxx.html */
       faireComposition = function (ctx) {
           ctx.fillStyle = "crimson" ;
           ctx.fillRect (50,60,350,250) ;
           ctx.fillStyle = "green" ;
           ctx.fillRect (180,10,150,150) ;

           ctx.globalCompositeOperation = "source-atop" ;
           ctx.fillStyle = "deepskyblue" ;
           ctx.arc (100,100,150, 0, 2*Math.PI) ;
           ctx.fill() ;

           ctx.globalCompositeOperation = "destination-over" ;
           ctx.fillStyle = "gold" ;
           ctx.fillRect (140,140,210,230) ;
       }
```



6.5. transparences

Les couleurs ont la possibilité d'avoir une composante alpha. C'est le cas également des fichiers png ou svg qui admettent la transparence avec canal alpha. On peut imposer un canal alpha pour tous les tracés à venir.

syntaxe

`globalAlpha = valeur entre 0 et 1`

exemple

```
/* fichier can013xxx.html */
       faireCanvas = function (ctx) {
           ctx.strokeStyle =  "#0000aa" ;
           ctx.lineWidth =5 ;
           ctx.strokeRect (52, 52, 296, 196) ;

           ctx.globalAlpha = 0.5 ;
           ctx.fillStyle = "rgb(255,0,0)" ;
           ctx.fillRect(60, 100, 180, 120) ;

           ctx.fillStyle = "rgb(0,255,0)" ;
           ctx.fillRect(130,60, 160, 130) ;

           ctx.fillStyle = "rgb(0, 0, 255)" ;
           ctx.fillRect(170, 120, 160, 120) ;
       }
```

6.6. ombres

Il existe une panoplie d'attribut pour l'ombrage :

* **`shadowOffsetX`** : décalage en X (en pixels)

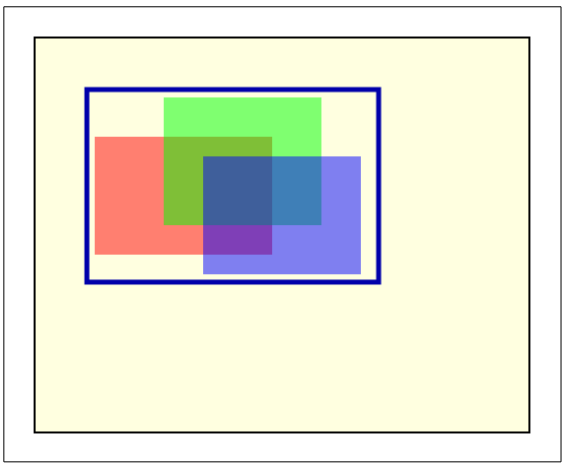
* **`shadowOffsetY`** :décalage en Y (en pixels)

* **`shadowBlur`** : importance du floutage ; largeur de diffusion en pixel

* **`shadowColor`** : une couleur.

exemple

```
/* fichier can014xxx.html */
       faireCanvas = function (ctx) {
           ctx.shadowOffsetX = 4 ;
           ctx.shadowOffsetY = 4 ;
           ctx.shadowBlur = 5 ;
           ctx.shadowColor = "orange" ;

           ctx.strokeStyle =  "#0000aa" ;
           ctx.lineWidth = 5 ;
           ctx.strokeRect (50, 50, 400 , 300) ;

           ctx.font = "80pt 'Liberation Serif', serif" ;
           ctx.strokeStyle = "blue" ;
           ctx.lineWidth = 2 ;
           ctx.fillStyle = "yellow" ;
           ctx.textBaseline ="top" ;
           ctx.textAlign = "left" ;
           ctx.fillText ("Bonjour",75, 80) ;
           ctx.strokeText ("Bonjour",75, 80) ;
           ctx.fillText ("JEAN",110, 210) ;
           ctx.strokeText ("JEAN",110, 210) ;
       }
```



7. les transformations

Les transformations s'appliquent à l'ensemble de l'objet contexte actuel.

7.1. les transformations proposées

Chacune des méthodes suivantes crée un nouveau contexte de dessin, obtenu à partir du précédent par la transformation induite :

* **`scale(x, y)`** : change l'échelle

* **`rotate (angle)`** : applique une rotation

* **`translate (x, y)`** : opère une translation

* **`transform(), setTransform()`** : opère une transformation complexe.

Les transformations fonctionnent sur le principe de la pile ; ainsi si on compose deux transformations, c'est la seconde déclarée qui est exécutée en premier. Le script suivant :

```
ctx.translate (1024, 0) ;
ctx.rotate (Math.PI/2) ;
ctx.fillText ("D O M", 10, 80) ;
```

est à comprendre comme :

- dessiner le texte
- le tourner d'un quart de tour
- translater

Si on veut revenir à une commande avec le contexte initial ; il convient donc d'écrire la rotation inverse puis la translation inverse ; les transformations seront effectuées dans l'ordre inverse.

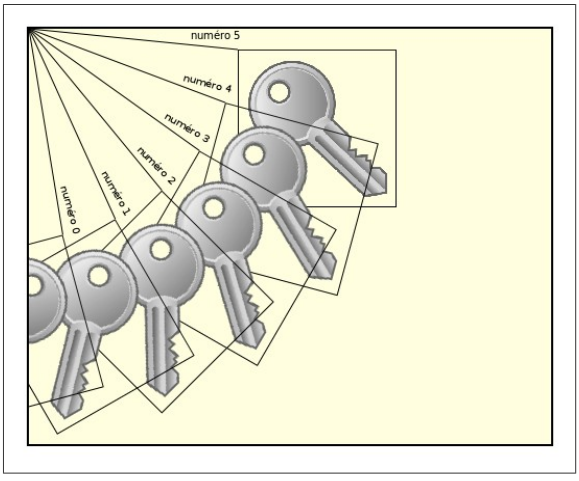
7.2. premier exemple

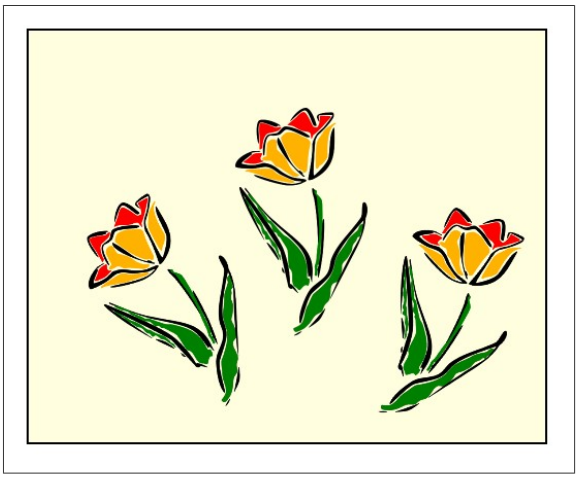
```
/* fichier can015axxx.html */
       fnFaire = function (ctx, imag) {
           ligne = function () { /* fonction incluse */
               ctx.lineWidth = 1 ;
               ctx.beginPath() ;
               ctx.moveTo (0, 0) ;
               ctx.lineTo (200, 20);
               ctx.stroke() ;
               ctx.closePath() ;
           }
           motif = function () { /* fonction incluse */
               ctx.font ="14pt" ;
               ctx.fillText ("numéro "+cpt,155, 10) ;
               ctx.drawImage (imag, 200, 20) ;
               ligne() ;
```

```
            }
            cpt = 5 ;
            motif() ;
            t = setInterval (function() {
                cpt -= 1 ;
                if (cpt == 0) clearInterval(t) ;
                ctx.rotate (Math.PI/12 ) ;
                motif()
                }, 1000) ;
        }

        faireCanvas = function (ctx) {
            var imag = new Image() ;
            imag.src = "./img/clef.png" ;
            $(imag).bind ("load", function(event) {fnFaire(ctx, imag);}) ;
        }
```

7.3. deuxième exemple

```
/* fichier can015bxxx.html */
        fnFaire = function (ctx, imag) {
            ctx.translate (400,150) ;
            ctx.rotate(Math.PI / 6) ;
            ctx.scale(0.5,0.5) ;
            ctx.drawImage(imag, 0,0) ;

            ctx.rotate (Math.PI / -6);
            ctx.translate (-400,-150) ;
            ctx.drawImage(imag, 0,0) ;

            ctx.rotate (Math.PI / -6);
            ctx.translate (-420, 50) ;
            ctx.drawImage(imag, 0,0) ;
        }
        faireCanvas = function (ctx) {
            var imag = new Image() ;
            imag.src = "./img/tulipe.svg" ;
            $(imag).bind ("load", function(event) {fnFaire(ctx, imag);}) ;
        }
```





7.4. transform et setTransform

Il y a 6 arguments, caractérisant chacun une transformation. On **dans l'ordre** :

* **`échelleX`** : échelle horizontale ; flottant

* **`cisaillementX`** : cisaillement horizontal ; flottant

* **`cisaillementY`** : cisaillement vertical ; fottant

* **`échelleY`**:échelle verticale

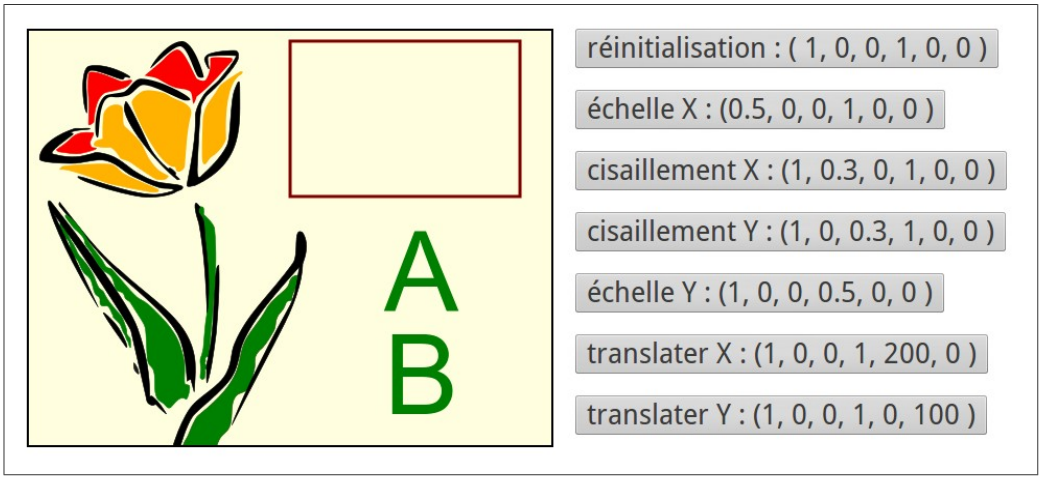
* **`translationX`** : déplacement horizontal en pixels

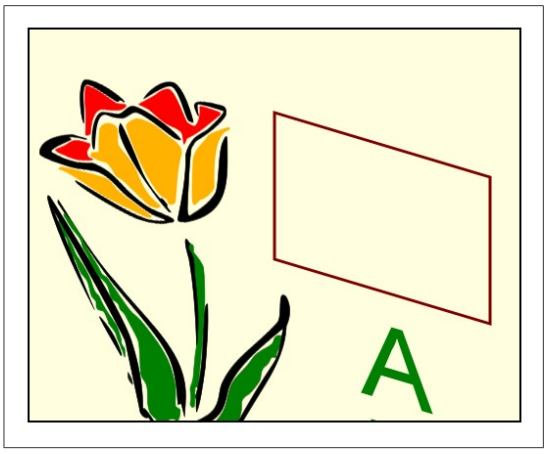
* **`translationY`** : déplacement vertical en pixels

Il existe deux commandes : **`transform()`** qui part du contexte actuel et applique la transformation décrite dans les arguments. La commande **`setTransform()`** applique la transformation sur le contexte réinitialisé.

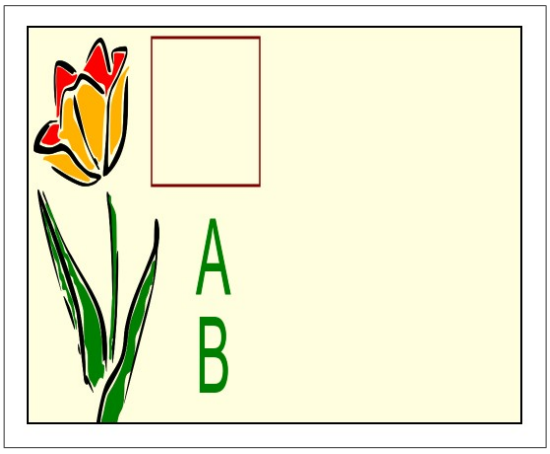
exercice

Les démonstrations sont réalisés à partir d'une réinitialisation du contexte à chaque démo. Il n'y a pas ici de transformations cumulées.





cisaillement horizontal



échelle horizontale

```
/* fichier can015cxxx.html */
       faireCanvas = function (ctx) {
           var fnFaire = function () {
                   ctx.drawImage (imag, 10, 10) ;
                   ctx.strokeStyle = "#800000" ;
                   ctx.lineWidth = 3 ;
                   ctx.strokeRect (250, 10, 220, 150) ;
                   ctx.fillStyle = "green" ;
                   ctx.font = "80pt Arial" ;
                   ctx.fillText ("A", 340, 270) ;
                   ctx.fillText ("B", 340, 370) ;
               }
           var imag = new Image() ;
           imag.src = "./img/tulipe.svg" ;
           $(imag).bind ("load", function(event) {fnFaire();}) ;
       }

       initialisation = function () {
           var action = function (id,a,b,c,d,e,f) {
               $(id).bind ("click", function (event) {
               contexte.setTransform(1,0,0,1,0,0) ;
               contexte.clearRect (0,0,500,400) ;
```

```
            contexte.setTransform(a,b,c,d,e,f) ;
            faireCanvas (contexte) ;
            });
        }
        var contexte = $("#id_canvas").get(0).getContext("2d") ;
        faireCanvas (contexte) ;
        action ("#id_init", 1,0,0,1,0,0) ;
        action ("#id_echelleX", 0.5,0,0,1,0,0) ;
        action ("#id_cisaillementX", 1,0.3,0,1,0,0) ;
        action ("#id_cisaillementY", 1,0,0.3,1,0,0) ;
        action ("#id_echelleY", 1,0,0,0.5,0,0) ;
        action ("#id_translaterX", 1,0,0,1,200,0) ;
        action ("#id_translaterY", 1,0,0,1,0,100) ;
    }
```

8. travail au niveau des pixels

8.1. L'objet ImageData

Le **`canvas`** propose trois méthodes pour traiter les pixels de l'image :

* **`createImageData(w, h)`** : crée un objet (nous le nommerons **imgData** ; son type est **`ImageData`**), représentatif d'un bitmap de largeur w et hauteur h. Cet objet comporte un tableau **`data`** à une seule dimension. Chaque item de **`data`** est un octet, caractérisant le niveau d'une primitive : rouge, vert, bleu et alpha. Un pixel est donc représenté par 4 éléments de **`data`**.

Ainsi, le premier pixel est **`imgData.data[0], imgData.data[1], imgData.data[2] imgData.data[3`**].

La longueur du tableau est **`imgData.data.length`** et vaut **(w** x **h)**

On dispose aussi de la syntaxe : **`createImageData(imageData)`** où **`imagedata`** est une donnée image déjà existante.

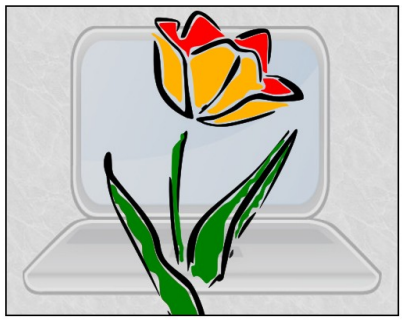
* **`getImageData(x, y, w, h)`** : retourne un objet **imgData. Voir l'encadré pour les questions de sécurité**

* **`putImageData(imgData, x, y)`** : place les données d'un **imgData** en **(x, y)** sur le **`canvas`**.

8.2. un exemple

Le cahier des charges de l'exemple est le suivant : un fond d'image est donné par un motif image (le *laptop* grisé) ; l'image de la tulipe est affichée en symétrie verticale ; le bouton d'inversion permet d'inverser les couleurs du canvas, sans toucher au canal alpha.

toggle inverser



toggle inverser

le script

```
 /* fichier can015cxxx.html */
faireCanvas = function (ctx) {
            var fnFaire = function(event) {
                /* inversion de l'image du canvas */
                var donne = ctx.getImageData(0,0,500,400) ;
                for (var index=0 ; index < donne.data.length; index++) {
                    if ((index +1) % 4 ) { /* on laisse le canal alpha */
                        donne.data[index] = 255 - donne.data[index] ;
                    }
```

```
            } ;/* boucle for */
            ctx.putImageData (donne, 0,0) ;
        } ; /* fin de fonction locale */

        var imag = new Image() ;
        var fond = new Image() ;
        fond.src ="./img/computer.jpeg" ;
        $(fond).bind ("load", function(event) {
            motif = ctx.createPattern (fond, "no-repeat");
            ctx.fillStyle = motif ;
            ctx.fillRect (0,0,500,400) ;
            imag.src = "./img/tulipe.svg" ;
            $(imag).bind ("load", function(event) {
                ctx.transform (-1,0,0,1,500,0) ; /* retourner l'image */
                    ctx.drawImage (imag, 120,5) ;
                $("#id_bt").bind("click", fnFaire) ;
            })
        });
    }
```

question de sécurité avec getImageData

Il se peut que le test échoue avec certains paramétrages des navigateurs. Cela est dû au fait que les normes de sécurité demandent que l'on ne puisse pas travailler sur une image provenant d'un autre domaine que celui où se trouve le script. C'est en général le cas lorsque l'image est dans le même site que la page contenant le script ; sinon l'origine des images est considérée comme étrangère au domaine (***cross-origin data).***

Mais si on travaille dans le système de fichiers, il n'y a pas de domaine déclaré, contrairement à ce qui se passe si on utilise un serveur local (**`localhost`**). Certains navigateurs interprètent alors ceci comme une pollution dans les fichiers. **C'est le cas de Chrome.** Il faut donc déclarer au navigateur qu'il ne doit pas tenir compte de ce type d'erreur.

Concrètement, au lieu de lancer **Chrome** sans paramètre, il faut utiliser la ligne de commande et lancer :

```
>>>chromium-browser --disable-web-security
```

O.K.