

# Piloter canvas avec KineticJS

Par bestmomo



[www.openclassrooms.com](http://www.openclassrooms.com)

*Licence Creative Commons 6 2.0  
Dernière mise à jour le 14/11/2012*

## Sommaire

Sommaire .....	2
Partager .....	2
Piloter canvas avec KineticJS .....	4
Partie 1 : Dessiner .....	5
Premières formes .....	5
Mise en route .....	5
Présentation .....	5
Installation .....	5
Structure de base .....	6
Les classes .....	8
Formes prédéfinies .....	10
Rectangles .....	10
Cercles et ellipses .....	14
Polygones .....	16
Polygones réguliers .....	18
Texte .....	20
Écrire avec style .....	21
Contour du texte .....	21
Style du texte .....	22
Paragraphes .....	23
Alignement .....	24
Hauteur de ligne .....	24
Boîtes .....	25
Suivre un chemin SVG .....	27
Tracés et images .....	28
Lignes .....	29
Dessiner une ligne .....	29
Jointures .....	29
Extrémités de lignes .....	30
Interruptions de lignes .....	31
Shape : une classe à tout faire .....	32
Arcs .....	32
Courbes .....	35
Combinaisons .....	37
Images .....	39
Exemple de base .....	39
Charger plusieurs images .....	40
Utilisation d'un plugin pour le chargement des images .....	41
Style .....	43
Traits et remplissage .....	43
Traits .....	43
Remplissage uni .....	44
Dégradé linéaire .....	45
Dégradé radial .....	49
Opacité et ombre .....	51
Opacité .....	51
Ombre .....	53
Manipulations .....	55
Modifications .....	55
Positionnement .....	55
Rotation .....	56
Dimension .....	59
Groupements .....	62
Principe de base .....	62
Exemple de création d'objets .....	63
TP .....	65
Un paysage .....	65
Une vague .....	67
Encore le paysage .....	68
Fleurs .....	71
Partie 2 : Animer .....	73
Interactions .....	74
Événements .....	74
Mise en place .....	74
Empilement d'événements .....	76
Supprimer un événement .....	77
Suspendre un événement .....	78
Position du curseur .....	80
Déetecter des pixels .....	82
Glisser-déposer .....	85
Activation simple .....	85
Un exemple avec des lignes .....	86
Activation pour un groupe .....	87
Événements du glisser-déposer .....	88

Contraintes de déplacement .....	89
Contrainte linéaire .....	89
Limite de déplacement .....	91
TP .....	94
Un petit jeu .....	94
Un tool-tip .....	97
<b>Un exemple d'application .....</b>	<b>99</b>
Présentation .....	100
Aspect de l'interface .....	100
Les menus .....	100
Les barres de boutons .....	100
La barre inférieure .....	101
Il sait faire quoi ? .....	101
Principe de fonctionnement .....	101
Le dessin .....	102
Etape 1 : fixation du point initial .....	103
Etape 2 : positionnement sur le point final .....	103
Etape 3 : dessin définitif .....	105
Sélection et édition .....	106
Sélection automatique de la forme qu'on vient de dessiner .....	106
Cas de l'ellipse .....	108
Sélection d'une forme avec un clic .....	109
Désactivation de la sélection par touche d'échappement .....	110
Édition .....	110
Apparence et enregistrement .....	112
Contrôle de l'apparence .....	112
Ordre de superposition des formes .....	114
Enregistrement en image ou en JSON .....	115
<b>Animations .....</b>	<b>116</b>
Les transitions .....	117
Une transition simple pour commencer .....	117
Les effets de mouvement .....	118
Fonction de rappel .....	124
Stopper et reprendre une animation .....	125
TP .....	127
Les sprites .....	132
Le principe des sprites .....	133
Animation d'un sprite .....	133
Où on améliore notre script .....	135
Plusieurs animations .....	137
TP .....	139
Les animations .....	141
Quand on retrouve les transitions .....	141
Déplacement sur une ellipse .....	142
Mettre en mouvement plusieurs formes .....	145
TP .....	151



## Piloter canvas avec KineticJS

Par [bestmomo](#)

Mise à jour : [14/11/2012](#)

Difficulté : Facile



**KineticJS** est un framework **Javascript** qui permet d'utiliser plus facilement l'API **Canvas** du **HTML5**. Il étend les possibilités de **Canvas** : groupes, calques, événements... Il permet également de réaliser des animations très fluides par un système de buffers.

Pour suivre ce tutoriel, vous avez besoin de connaître **Javascript**. Si ce n'est pas le cas, commencez par vous initier à ce langage, par exemple avec [ce tutoriel](#). Je ne donne aucune indication quant au fonctionnement de ce langage dans ce cours qui est axé uniquement sur l'utilisation de **KineticJS**.

## Partie 1 : Dessiner

Dans cette première partie nous allons apprendre à dessiner avec **KineticJS**.

### Premières formes

Dans ce premier chapitre nous allons voir comment démarrer avec **KineticJS** et dessiner des formes prédéfinies.

#### Mise en route

##### Présentation

**Canvas** est une nouvelle fonctionnalité introduite avec le **HTML5**. Pour être honnête, c'est **Apple** qui l'a inventée et elle a été intégrée dans les [spécifications du W3C](#) et celles du [WhatWG](#).

Il existe de nombreuses librairies **Javascript** pour piloter **Canvas** :

- EaselJS
- jCanvascript
- jCanvas
- Artisan JS
- oCanvas
- jcotton
- Fabric
- canvasxpress
- gury
- bHive
- ...

Il me semble que **KineticJS**, créée et maintenue par Eric Drowell, est l'une des plus efficaces et elle est d'autre part bien documentée.

Le [site](#) est plutôt bien fait et comporte de nombreuses démonstrations. Il y a avait un forum dédié que j'aimais bien, mais il a disparu au profit de [stackoverflow](#). Les [tutoriels](#) sont bien organisés et couvrent presque toutes les fonctionnalités, on peut toutefois regretter le fait qu'ils soient souvent en retard sur les options proposées.

Vous n'avez pas forcément besoin de savoir manipuler les commandes de base de l'API **Canvas** pour utiliser **KineticJS** mais il est évident que vous utiliserez cette librairie de façon beaucoup plus efficace si vous avez des notions concernant cette API.

### Installation

Pour utiliser **KineticJS** vous devez d'abord télécharger la librairie sur [le site](#) pour obtenir à coup sûr la dernière version et la placer dans un répertoire de votre site. Au moment où j'écris ce tutoriel, la dernière version est la 4.0.4.

Le lien ouvre directement le fichier dans votre navigateur :

```
/**  
 * KineticJS JavaScript Library v4.0.4  
 * http://www.kineticjs.com/  
 * Copyright 2012, Eric Rowell  
 * Licensed under the MIT or GPL Version 2 licenses.  
 * Date: Oct 19 2012  
 *  
 * Copyright (C) 2011 - 2012 by Eric Rowell
```

Il suffit de faire un "Enregistrer sous..." pour placer le fichier dans un répertoire ("js" ou "librairies" ou autre). Pour ce tutoriel je le place dans un répertoire "js". Il suffit ensuite de le référencer sur les pages web :

Code : HTML

```
<script src="js/kinetic-v4.0.4.min.js"></script>
```

J'utilise la syntaxe **HTML5** pour tout ce tutoriel, c'est pour cette raison que je n'ai pas précisé le type de script dans la balise `<script>` puisque **JavaScript** est le type par défaut.

Il peut également être utile de récupérer le fichier source avec le code en clair et les commentaires pour utiliser la librairie de façon plus efficace. Etant donné le décalage entre la mise à jour de la documentation et les fonctionnalités proposées c'est même presque indispensable. Le mieux est sans doute d'aller voir sur la [page de GitHub consacrée au projet](#).

## Structure de base

Pour tous les exemples de ce tutoriel, nous allons avoir besoin d'une structure **HTML** de base. Ça va être tout simple parce que ça se limite à créer un conteneur avec un identifiant :

### Code : HTML

```
<body>
  <div id="kinetic"></div>
</body>
```

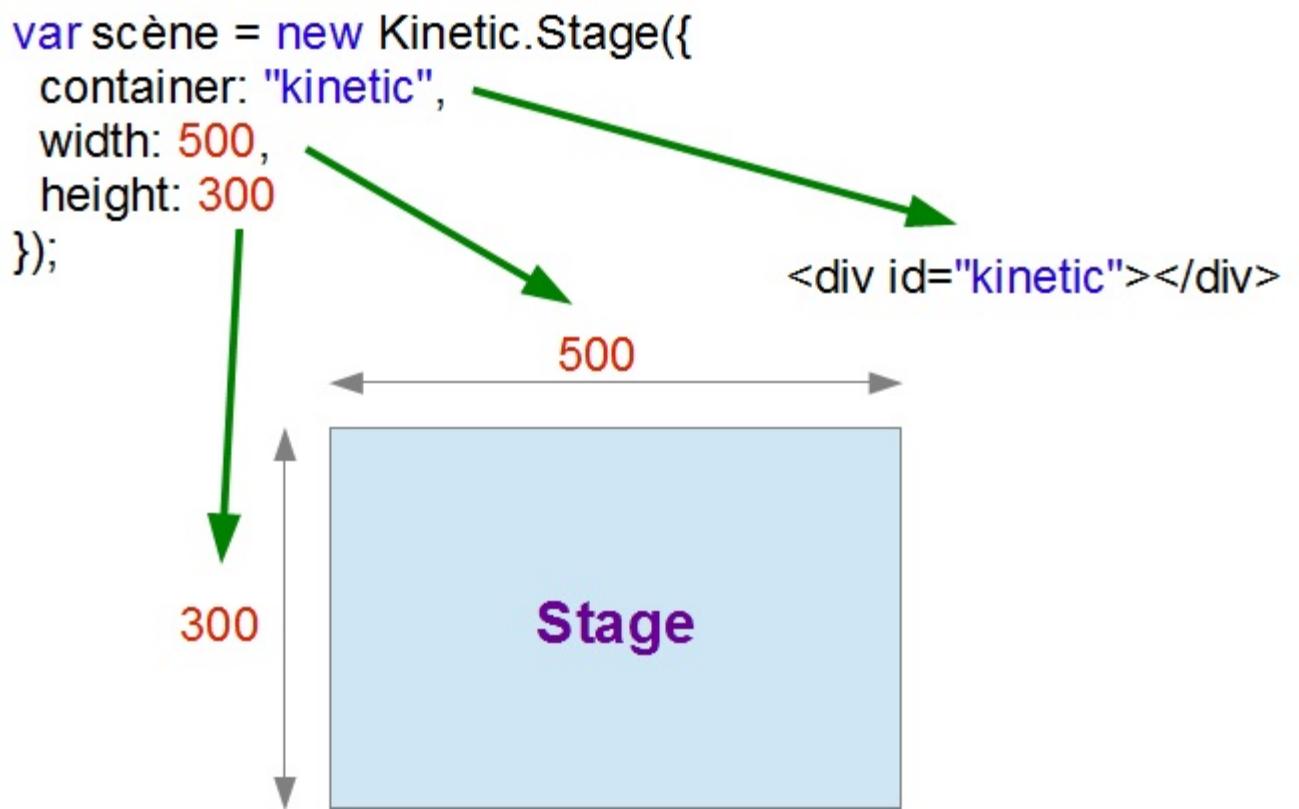
Il faut ensuite une structure **JavaScript** de base :

### Code : JavaScript

```
window.onload = function() {
  var scène = new Kinetic.Stage({
    container: "kinetic",
    width: 600,
    height: 400
  });

  var calque = new Kinetic.Layer();
  // Ici on dessine sur le calque !
  scène.add(calque);
};
```

**Kinetic** est l'espace de nom qui permet d'isoler tout ce que nous allons faire avec cette librairie. La première action est de créer une scène (**Stage**) pour contenir nos créations. Le constructeur de cet objet attend 3 paramètres. Le premier est l'identifiant du contenant **HTML**, dans notre cas cet identifiant est "kinetic". Les deux paramètres suivants définissent la largeur et la hauteur de la scène.



Il faut ensuite définir au moins un calque pour dessiner :

**Code : JavaScript**

```
var calque = new Kinetic.Layer();
```

Et ajouter ensuite ce calque dans la scène :

**Code : JavaScript**

```
scène.add(calque);
```

Il est aussi judicieux de repérer les limites de la scène en créant une bordure avec une touche de style :

**Code : CSS**

```
canvas {border-style:solid;}
```



Mais où se trouve la balise canvas ?

On ne met pas cette balise de façon explicite dans le code **HTML** mais elle est générée automatiquement par la librairie.

Reprendre tous ces éléments pour obtenir la structure de base de tous nos exemples :

**Code : HTML**

```
<!doctype html>  
<html>  
<head>
```

```
<meta charset="utf-8">
<script src="js/kinetic-v4.0.4.min.js"></script>
<style>
  canvas {border-style:solid;}
</style>
<script>
  window.onload = function() {
    var scène = new Kinetic.Stage({
      container: "kinetic",
      width: 500,
      height: 300
    });
    var calque = new Kinetic.Layer();
    // ...
    scène.add(calque);
  };
</script>
</head>
<body>
  <div id="kinetic"></div>
</body>
</html>
```

Tester !

Maintenant nous avons tout ce qui est nécessaire pour commencer à dessiner 😊

Pour ceux qui ont la curiosité de voir le code généré :

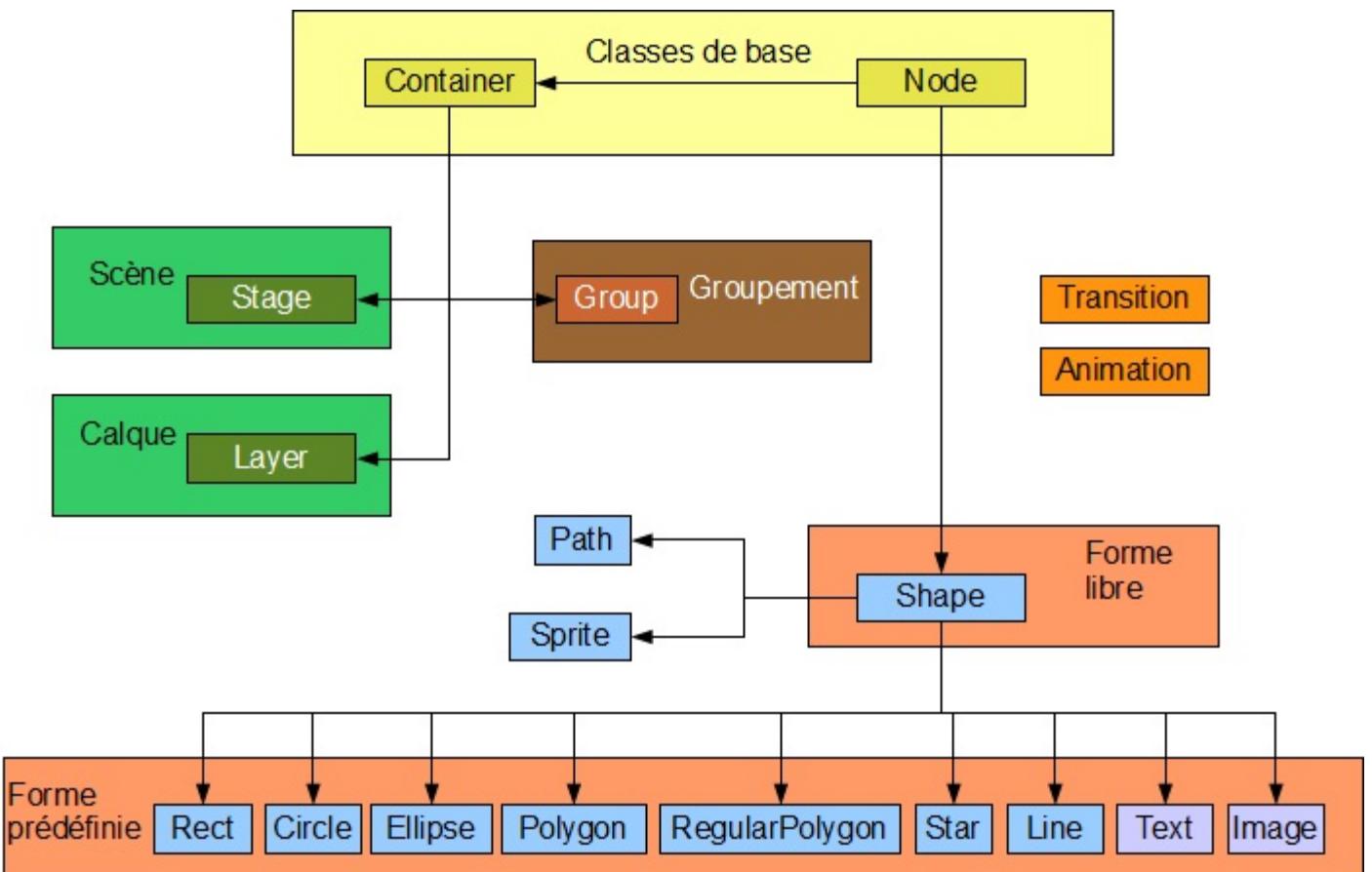
#### Code : HTML

 `<body>
 <div id="kinetic">
 <div style="position: relative; display: inline-block; width: 500px; height: 300px;" class="kineticjs-content">
 <canvas style="position: absolute; display: none; " class="kineticjs-buffer-layer" width="500" height="300"></canvas>
 <canvas style="position: absolute; display: none; " class="kineticjs-path-layer" width="500" height="300"></canvas>
 <canvas style="position: absolute; " width="500" height="300"></canvas>
 </div>
 </div>
</body>`

On se rend compte que trois balises **canvas** sont créées dont deux sont masquées.

## Les classes

**KineticJS** est organisé en classes hiérarchisées. Voici une illustration de sa structure :



Ce diagramme est issu de mes investigations dans le fichier de **KineticJS** étant donné que la documentation a un peu de retard.

Voici une description sommaire de ces classes :

Classe	Description
<b>Node</b>	C'est la classe principale qui offre les propriétés et méthodes de base pour créer des objets graphiques
<b>Container</b>	Cette classe offre des fonctionnalités pour contenir des Nodes
<b>Group</b>	Permet de regrouper des objets graphiques pour leur appliquer des actions
<b>Stage</b>	Permet de créer une scène pour dessiner et animier
<b>Layer</b>	Pour créer des calques indépendants
<b>Shape</b>	Classe qui permet de dessiner
<b>Path</b>	Pour dessiner en SVG
<b>Sprite</b>	Pour gérer des images regroupées sur une seule image
<b>Rect</b>	Pour dessiner des rectangles
<b>Circle</b>	Pour dessiner des cercles
<b>Ellipse</b>	Pour dessiner des ellipses et donc aussi des cercles
<b>Polygone</b>	Pour dessiner des polygones quelconques
<b>Star</b>	Pour dessiner des étoiles
<b>Line</b>	Pour dessiner des lignes en gérant les liaisons et les extrémités
<b>Text</b>	Pour dessiner du texte
<b>Image</b>	Pour afficher des images
<b>Transition</b>	Une classe isolée pour faire des transitions de position, taille, rotation, etc...

## Animation | Une classe isolée pour faire des animations

Il est à noter que le développeur de cette librairie est assez réactif et sort souvent des correctifs, quelques fois un peu rapidement d'ailleurs et je conseille de suivre l'évolution des versions en attendant quelque temps pour que les bugs soient corrigés 😊.

### Formes prédéfinies

#### Rectangles

##### *Notre premier dessin*

Pour commencer tranquillement on va dessiner un rectangle bleu avec la classe **Rect**. Le constructeur attend au minimum une largeur et une hauteur. Mais si on se contente de ça, on ne voit pas grand-chose à l'écran 😊. Voyons le code :

##### Code : JavaScript

```
window.onload = function() {
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

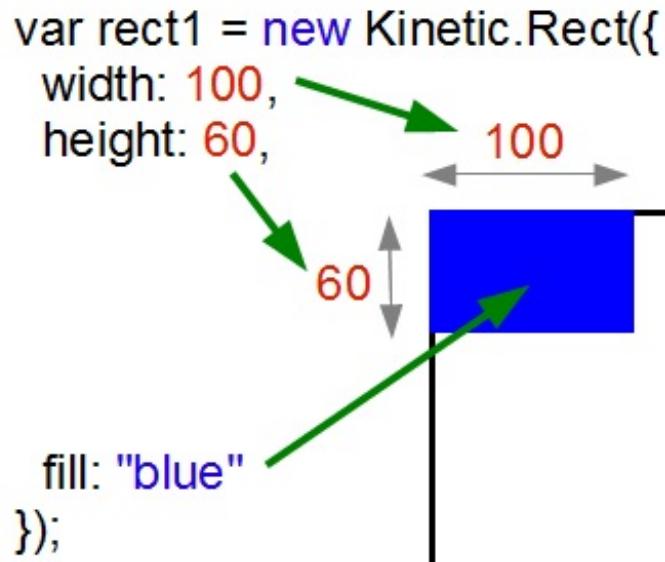
    var calque = new Kinetic.Layer();

    var rectangle = new Kinetic.Rect({
        width: 100,
        height: 60,
        fill: "blue"
    });

    calque.add(rectangle);

    scène.add(calque);
};
```

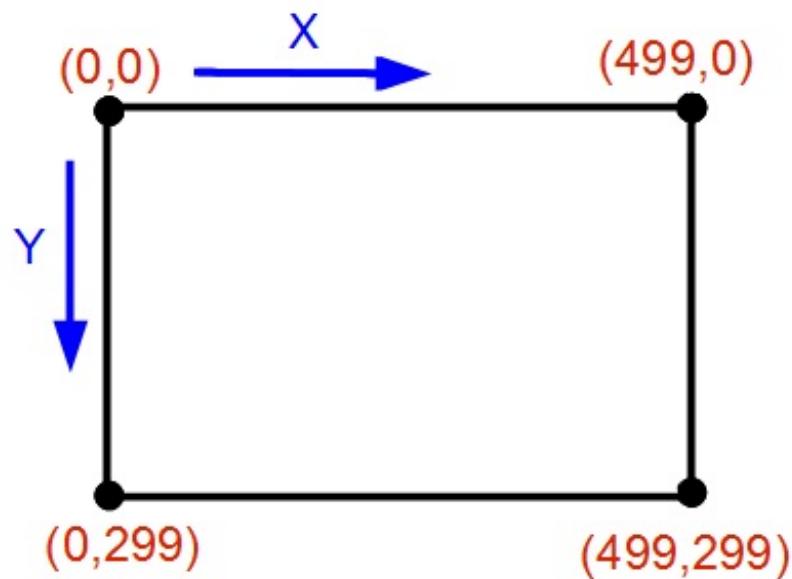
Tester !



C'est la propriété **fill** qui permet le remplissage. Ici je me suis contenté de la remplir d'un bleu uniforme, on verra qu'on peut également utiliser une image ou des dégradés. Ici j'ai écrit le nom de la couleur, on peut aussi utiliser le code hexadécimal comme pour les propriétés CSS. On peut également utiliser la fonction **rgb** comme nous le verrons plus loin.

### Positionnement

Vous avez sans doute aussi remarqué que le rectangle bleu vient se caler en haut et à gauche du cadre. Pour comprendre ce comportement, il faut déjà connaître le système de coordonnées utilisé. L'origine des axes est justement située en haut et à gauche :



L'axe des **X** s'étire horizontalement vers la droite et celui des **Y** verticalement vers le bas.

La référence de positionnement d'un rectangle est également située en haut à gauche. D'autre part les valeurs par défaut de x et y sont à 0. On comprend donc pourquoi notre rectangle bleu va se caler dans le coin haut gauche.

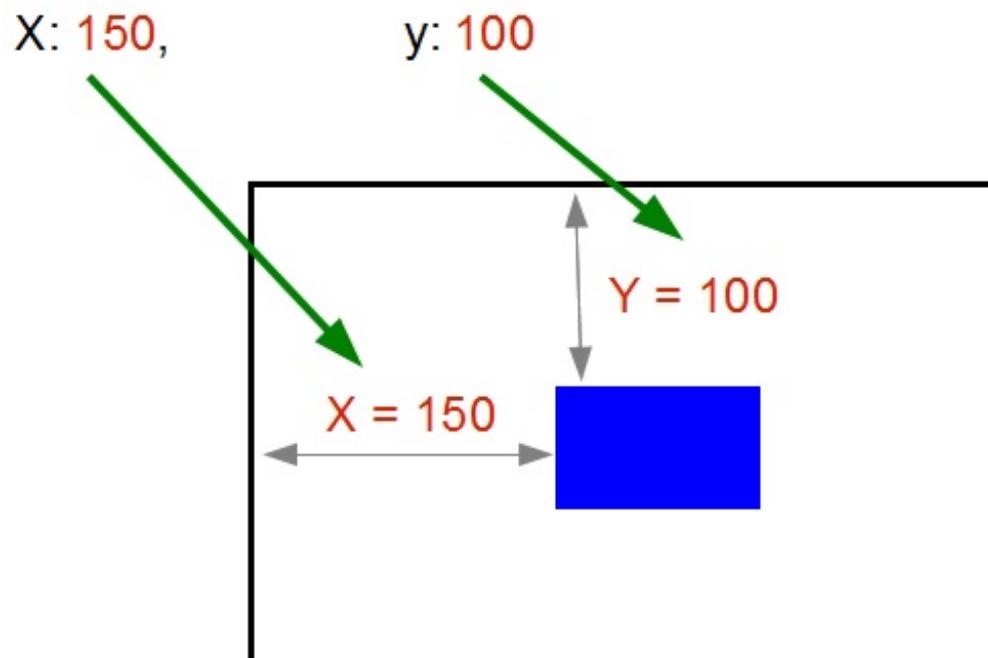
On va maintenant positionner différemment le rectangle en lui fixant des valeurs pour x et y :

#### Code : JavaScript

```
var rect1 = new Kinetic.Rect({  
    x: 150,  
    y: 100,  
    width: 100,  
    height: 60,  
    fill: "blue"  
});
```

Tester !

```
var rect1 = new Kinetic.Rect({
```



### La bordure

On peut aussi gérer la bordure du rectangle. Voici un exemple :

Code : JavaScript

```
window.onload = function() {
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

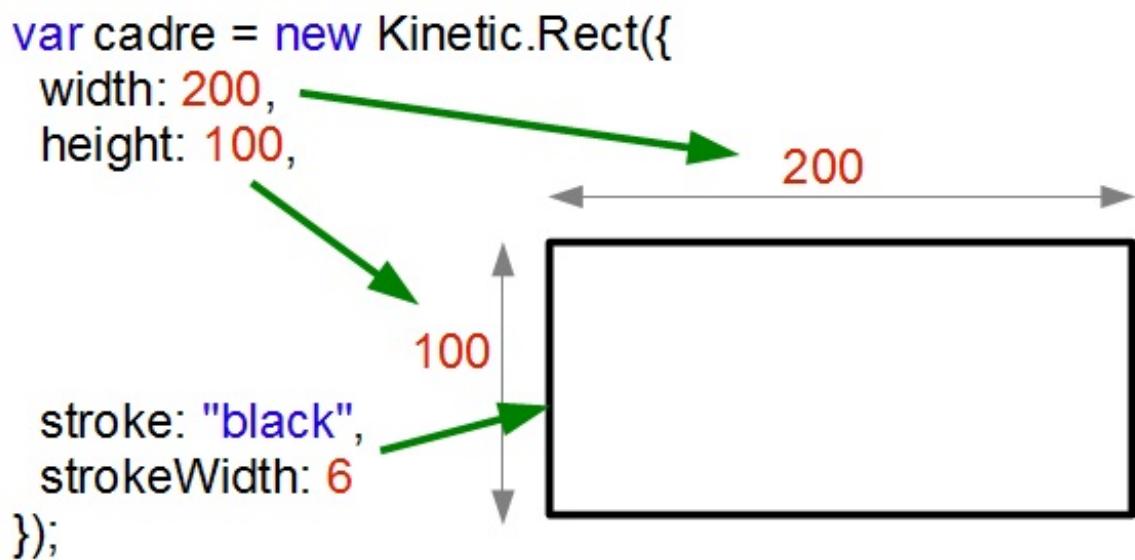
    var calque = new Kinetic.Layer();

    var rect1 = new Kinetic.Rect({
        x: 150,
        y: 100,
        width: 200,
        height: 100,
        stroke: "black",
        strokeWidth: 6
    });

    calque.add(rect1);

    scène.add(calque);
};
```

Tester !



On voit que le trait est géré par la propriété **stroke**. Ici j'ai choisi la couleur noire (black) et une épaisseur de 6 pixels.

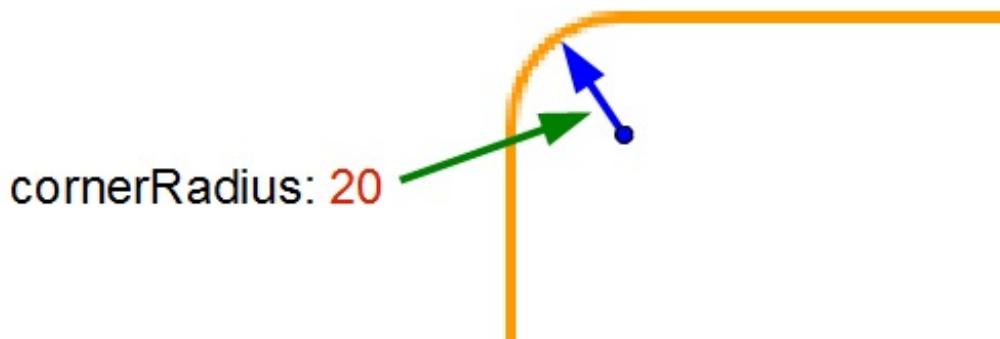
### Arrondir les angles

Les rectangles que nous avons dessinés jusqu'à présent ont des angles aigus. Il est possible de les arrondir avec la propriété **cornerRadius** qui fixe la valeur du rayon de courbure qui est par défaut à 0.

Code : JavaScript

```
var rect1 = new Kinetic.Rect ({
  x: 150,
  y: 100,
  width: 200,
  height: 100,
  stroke: "#F90",
  strokeWidth: 2,
  cornerRadius: 20
});
```

Tester !



## Cercles et ellipses

### Rayon

La classe **Circle** permet évidemment de dessiner des cercles. La dimension d'un cercle est fixée par la longueur de son rayon donnée par la propriété **radius**. Le trait est géré comme celui des rectangles. Voici un premier exemple :

#### Code : JavaScript

```
var cercle = new Kinetic.Circle({  
    radius: 80,  
    stroke: "darkgreen",  
    strokeWidth: 5  
});  
calque.add(cercle);
```

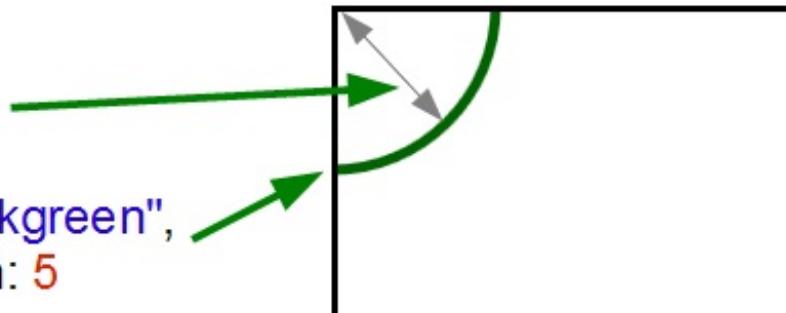
Tester !

```
var cercle = new Kinetic.Circle({
```

radius: 80,

stroke: "darkgreen",  
strokeWidth: 5

```
});
```



### Position

On s'attendait à voir un cercle et on en obtient qu'un morceau. Le point de référence est le centre du cercle qui vient tout naturellement se fixer à l'origine des coordonnées. Seule la partie tracée à l'intérieur de la scène apparaît. Pour avoir le cercle entier, nous devons définir la position du centre :

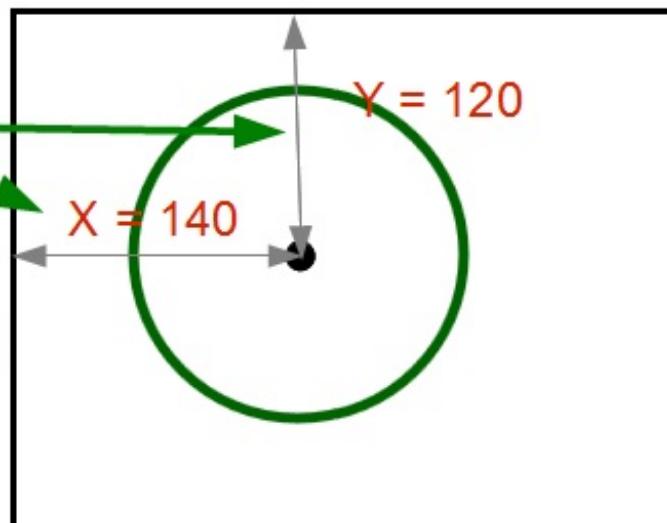
#### Code : JavaScript

```
var cercle = new Kinetic.Circle({  
    x: 140,  
    y: 120,  
    radius: 80,  
    stroke: "darkgreen",  
    strokeWidth: 5  
});  
calque.add(cercle);
```

Tester !

```
var cercle = new Kinetic.Circle({
```

x: 140,  
y: 120,  
radius: 80,  
stroke: "darkgreen",  
strokeWidth: 5  
});



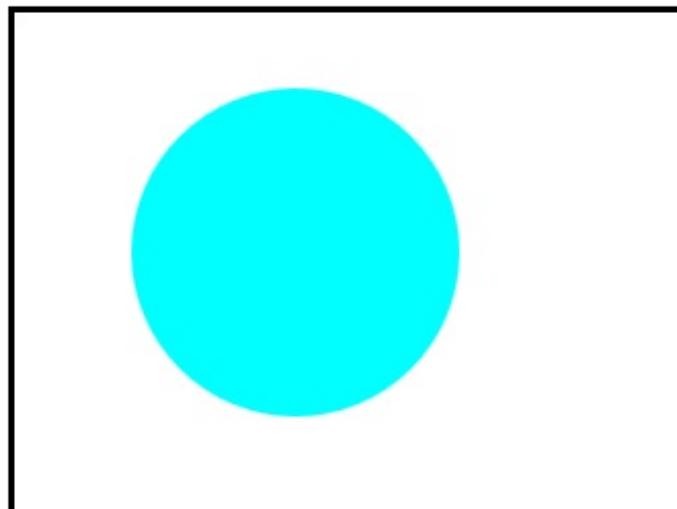
### Remplissage

La même propriété que le rectangle permet de remplir le cercle (ici avec une couleur cyan unie):

Code : JavaScript

```
var cercle = new Kinetic.Circle({  
    x: 140,  
    y: 120,  
    radius: 80,  
    fill: "#0ff"  
});
```

Tester !



### Une ellipse

Bon on va quand même dessiner une ellipse :

Code : JavaScript

```
var ellipse = new Kinetic.Ellipse({
```

```

    x: 140,
    y: 120,
    radius: [120, 80],
    stroke: "darkgreen",
    strokeWidth: 5
});

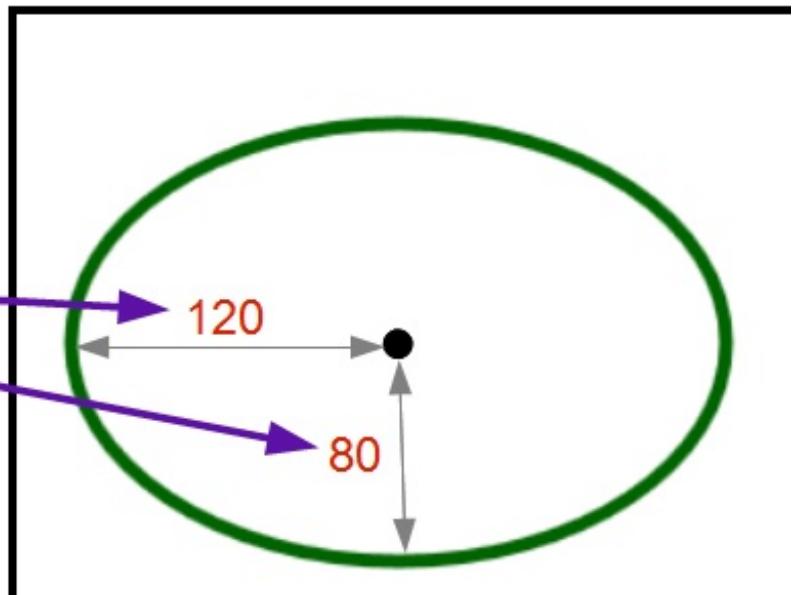
```

Tester !

```
var cercle = new Kinetic.Ellipse({
```

x: 140,  
y: 120,

Radius: [120,  
80]  
stroke: "darkgreen",  
strokeWidth: 5  
});



On voit qu'il suffit de définir la longueur du grand et du petit rayon dans un tableau. On peut aussi utiliser un objet avec les propriétés x et y. Donc ce code est équivalent :

**Code : JavaScript**

```

var ellipse = new Kinetic.Ellipse({
    x: 140,
    y: 120,
    radius: {x: 120, y: 80},
    stroke: "darkgreen",
    strokeWidth: 5
});

```

## Polygones

### Triangle

Pour tracer des polygones, il faut utiliser la classe **Polygon**. Elle permet de dessiner des polygones avec autant de côtés que l'on veut, il suffit de définir la position de chaque point. Voici par exemple un triangle :

**Code : JavaScript**

```

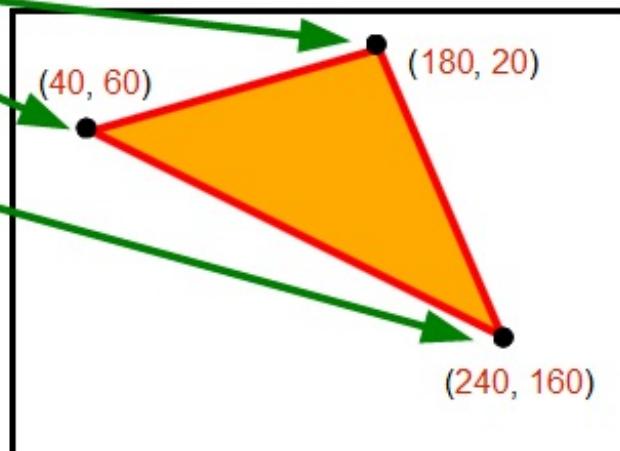
var triangle = new Kinetic.Polygon({
    points: [{x: 40, y: 60}, {x: 180, y: 20}, {x: 240, y: 160}],
    fill: "#fa0",
    stroke: "red",
    strokeWidth: 4
});

```

```
calque.add(triangle);
```

Tester !

```
var triangle = new Kinetic.Polygon({  
    points: [  
        {x:40,y:60},{x:180,y:20}  
        {x:240,y:160}],  
    fill: "#fa0",  
    stroke: "red",  
    strokeWidth: 4  
});
```



On voit que la définition du trait et du remplissage sont encore les mêmes.

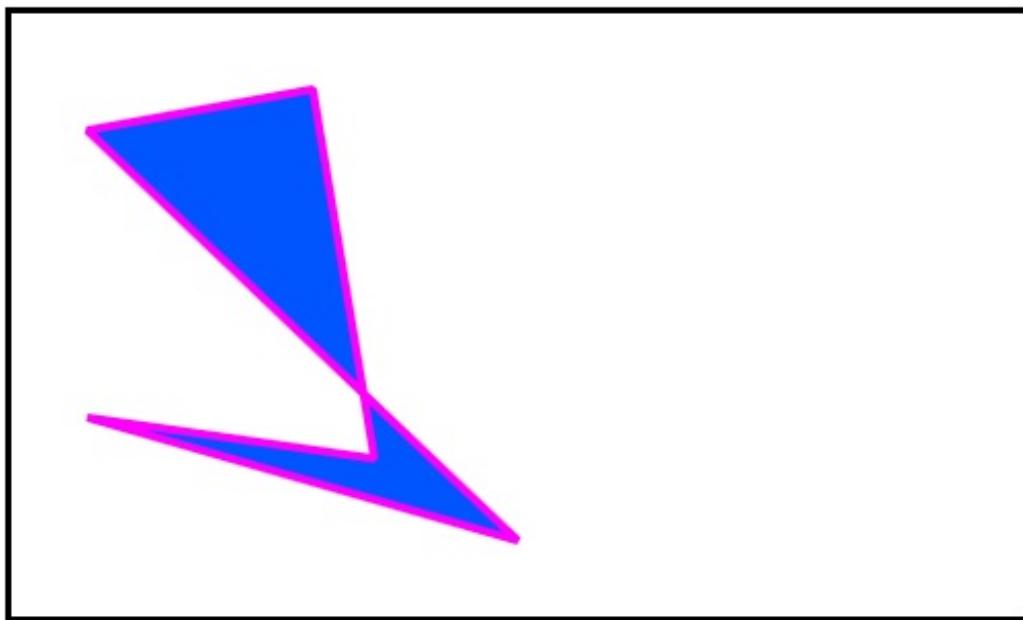
### Polygones variés

On est pas limités en nombre de points et les côtés peuvent se croiser, le remplissage se fait toujours de façon efficace :

#### Code : JavaScript

```
var polygone = new Kinetic.Polygon({  
    points :  
    [{x:40,y:60},{x:150,y:40},{x:180,y:220},{x:40,y:200},{x:250,y:260}],  
    stroke: "magenta",  
    strokeWidth: 4,  
    fill: "#05f"  
});  
calque.add(polygone);
```

Tester !



## Polygones réguliers

### *Carré*

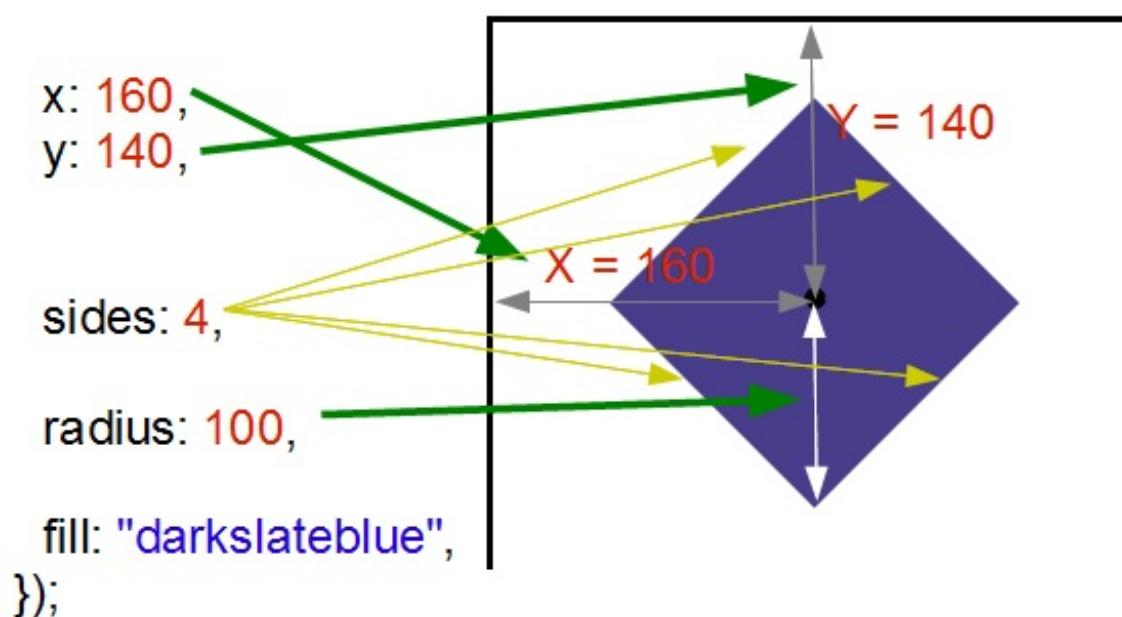
On peut dessiner des polygones réguliers avec la classe **RegularPolygon**. Il faut définir le nombre de côtés avec la propriété **sides**. Voici par exemple un carré :

#### Code : JavaScript

```
var carré = new Kinetic.RegularPolygon({  
    x: 160,  
    y: 140,  
    sides: 4,  
    radius: 100,  
    fill: "darkslateblue",  
});  
calque.add(carré);
```

Tester !

```
var carré = new Kinetic.RegularPolygon({
```



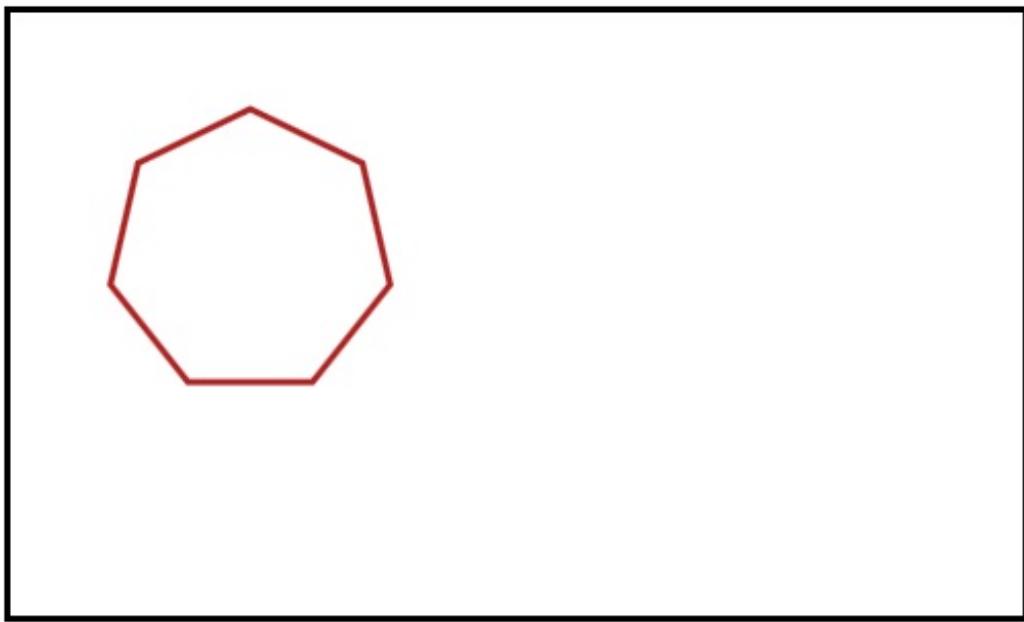
### Heptagone

On choisit le nombre de côtés comme on veut. Voici par exemple un heptagone :

Code : JavaScript

```
var heptagone = new Kinetic.RegularPolygon({
  x: 120,
  y: 120,
  sides: 7,
  radius: 70,
  stroke: "firebrick",
  strokeWidth: 3
});
```

Tester !



On voit que les traits sont encore gérés de la même manière.

### Étoile

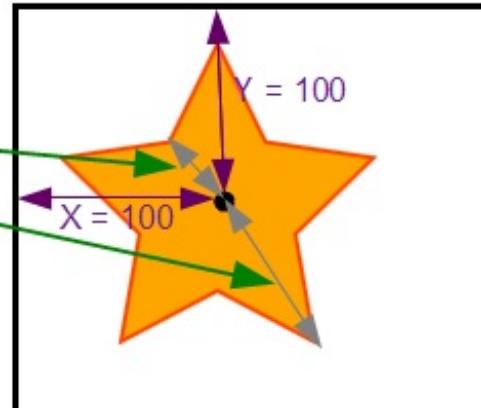
Il y a également une classe pour dessiner des étoiles, c'est la classe **Star**. Voici une étoile avec 5 branches :

Code : JavaScript

```
var étoile = new Kinetic.Star({
    x: 100,
    y: 100,
    numPoints: 5,
    innerRadius: 40,
    outerRadius: 80,
    fill: "orange",
    stroke: "orangered",
    strokeWidth: 2
});
calque.add(étoile);
```

Tester !

```
var étoile = new Kinetic.Star({
    x: 100,
    y: 100,
    numPoints: 5,
    innerRadius: 40,
    outerRadius: 80,
    fill: "orange",
    stroke: "orangered",
    strokeWidth: 2
});
```



## Texte

### Écrire avec style

On peut considérer le texte comme un ensemble de formes prédéfinies. Il y a une classe **Text** qui permet de réaliser de belles choses. Voici un premier exemple :

Code : JavaScript

```
var text = new Kinetic.Text({  
    x: 40,  
    y: 30,  
    text: "Je sais écrire",  
    fontSize: 26,  
    fontFamily: "Garamond",  
    textFill: "#c16"  
});  
calque.add(text);
```

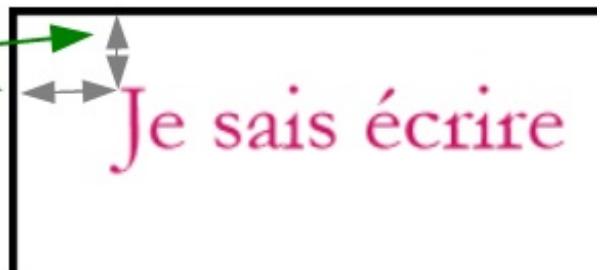
Tester !

```
var text = new Kinetic.Text({
```

X: 40,  
Y: 30,

text: "Je sais écrire",  
fontSize: 26,  
fontFamily: "Garamond",  
textFill: "#c16"

```
});
```



## Contour du texte

Dans l'exemple précédent, nous avons rempli chaque caractère avec une couleur unie avec la propriété **textFill**. Il est aussi possible de tracer uniquement le contour avec la propriété **textStroke** :

Code : JavaScript

```
var text = new Kinetic.Text({  
    x: 40,  
    y: 30,  
    text: "Je sais écrire",  
    fontSize: 36,  
    fontFamily: "Georgia",  
    textStroke: "blue",  
});  
calque.add(text);
```

Tester !



# Je sais écrire

On peut régler l'épaisseur du contour avec la propriété **textStrokeWidth** dont la valeur par défaut est 2 :

#### Code : JavaScript

```
var text = new Kinetic.Text({  
    x: 40,  
    y: 30,  
    text: "Je sais écrire",  
    fontSize: 36,  
    fontFamily: "Georgia",  
    textStroke: "blue",  
    textStrokeWidth: 1  
});  
calque.add(text);
```

Tester !



# Je sais écrire

## Style du texte

On peut définir le style du texte avec la propriété **fontStyle**. Elle peut prendre les valeurs **italic** ou **bold** :

#### Code : JavaScript

```
var text1 = new Kinetic.Text({  
    x: 40,  
    y: 30,  
    text: "Je suis en italique",  
    fontSize: 34,  
    fontFamily: "Georgia",  
    fontStyle: "italic",  
    textFill: "rgb(200,50,10)"  
});  
  
var text2 = new Kinetic.Text({  
    x: 40,  
    y: 80,  
    text: "Je suis en gras",  
    fontSize: 34,  
    fontFamily: "Georgia",  
    fontStyle: "bold",  
    textFill: "rgb(200,50,10)"  
});  
calque.add(text1);
```

```
calque.add(text2);
```

Tester !



*Je suis en italique*  
**Je suis en gras**

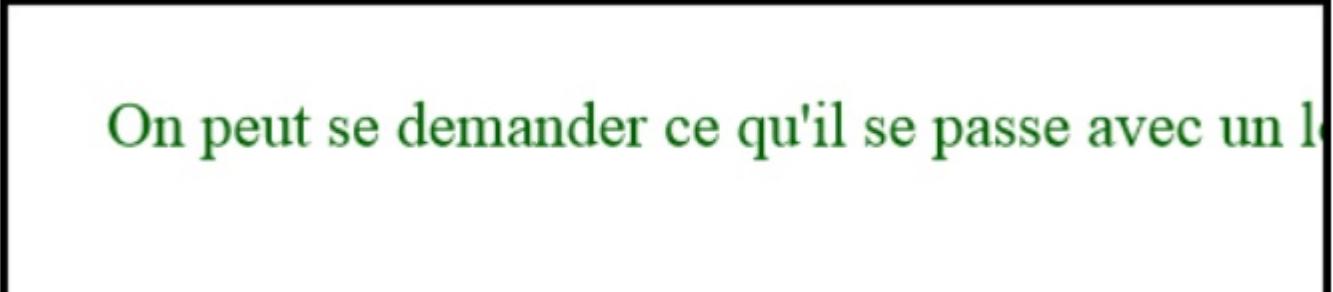
## Paragraphes

On peut se demander ce qu'il se passe si le texte à afficher est plus large que la scène. Voici un exemple :

Code : JavaScript

```
var text = new Kinetic.Text({  
    x: 40,  
    y: 40,  
    text: "On peut se demander ce qu'il se passe avec un long texte",  
    fontSize: 18,  
    fontFamily: "Fraktur",  
    textFill: "darkgreen",  
});  
calque.add(text);
```

Tester !



On peut se demander ce qu'il se passe avec un l'

On se rend compte qu'on perd le texte qui ne rentre pas dans le cadre. On peut arranger ça avec la propriété **width** qui limite la largeur du texte et provoque un renvoi à la ligne :

Code : JavaScript

```
var text = new Kinetic.Text({  
    x: 40,  
    y: 40,  
    text: "On peut se demander ce qu'il se passe avec un long texte",  
    fontSize: 18,  
    fontFamily: "Fraktur",  
    textFill: "darkgreen",  
    width: 200  
});  
calque.add(text);
```

```
width: 300  
});  
calque.add(text);
```

Tester !

On peut se demander ce qu'il se passe avec un long texte

## Alignement

On a vu qu'on peut créer des paragraphes, on peut aussi aligner le texte avec la propriété **align** qui est par défaut à **left**. Les autres valeurs sont **center** et **right**. voici le texte précédent centré :

Code : JavaScript

```
var text = new Kinetic.Text({  
    x: 40,  
    y: 40,  
    text: "On peut se demander ce qu'il se passe avec un long texte",  
    fontSize: 18,  
    fontFamily: "Fraktur",  
    textFill: "darkgreen",  
    width: 300,  
    align: "center"  
});  
calque.add(text);
```

Tester !

Voici un petit texte qui est très bien centré

## Hauteur de ligne

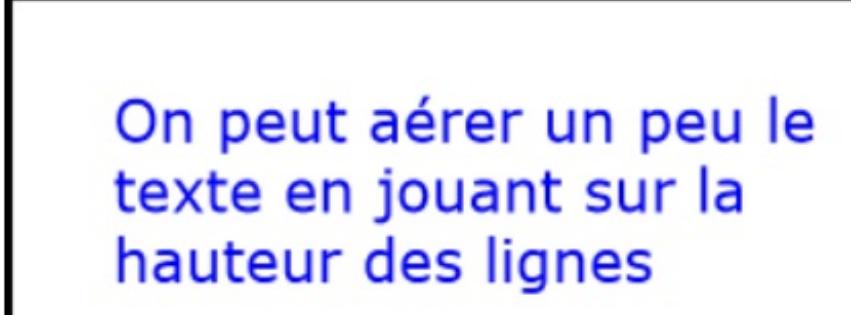
La hauteur des lignes est fixée par défaut à 1.2, on peut la modifier avec la propriété **lineHeight** :

Code : JavaScript

```
var text = new Kinetic.Text({  
    x: 40,  
    y: 40,  
    text: "On peut aérer un peu le texte en jouant sur la hauteur des  
    lignes",  
    fontSize: 16,
```

```
fontFamily: "Verdana",
textFill: "blue",
width: 300,
lineHeight: 1.6
});
calque.add(text);
```

Tester !



On peut aérer un peu le  
texte en jouant sur la  
hauteur des lignes

## Boîtes

On peut matérialiser visuellement la boîte qui contient le texte. Voici un exemple élémentaire :

Code : JavaScript

```
var text = new Kinetic.Text({
  x: 40,
  y: 40,
  text: "On peut visualiser la boîte qui contient le texte",
  fontSize: 20,
  fontFamily: "Verdana",
  textFill: "#3aa",
  width: 300,
  align: "center",
  stroke: '#177',
  strokeWidth: 3,
});
calque.add(text);
```

Tester !



On peut visualiser la  
boîte qui contient le  
texte

Il est possible de régler la distance entre le texte et la boîte avec la propriété **padding** :

Code : JavaScript

```
var text = new Kinetic.Text({
```

```
x: 40,  
y: 40,  
text: "On peut aussi régler la distance entre le texte et la  
bordure",  
fontSize: 16,  
fontFamily: "Verdana",  
textFill: "#3aa",  
width: 300,  
align: "center",  
stroke: '#177',  
strokeWidth: 3,  
padding: 10  
});  
calque.add(text);
```

Tester !



Il est aussi plus joli d'arrondir les coins avec la propriété **cornerRadius** :

Code : JavaScript

```
var text = new Kinetic.Text({  
x: 40,  
y: 40,  
text: "C'est plus joli avec des arrondis",  
fontSize: 16,  
fontFamily: "Verdana",  
textFill: "magenta",  
width: 300,  
align: "center",  
stroke: "blueviolet",  
strokeWidth: 3,  
padding: 10,  
cornerRadius: 20  
});  
calque.add(text);
```

Tester !



Il est possible de remplir la boîte avec une couleur de fond avec la propriété **fill** :

**Code : JavaScript**

```
var text = new Kinetic.Text({
  x: 40,
  y: 40,
  text: "On peut aussi avoir un fond coloré",
  fontSize: 16,
  fontFamily: "Verdana",
  textFill: "magenta",
  width: 300,
  align: "center",
  stroke: "blueviolet",
  strokeWidth: 3,
  padding: 10,
  cornerRadius: 20,
  fill: "lightgrey"
});
calque.add(text);
```

Tester !



## Suivre un chemin SVG

Vous avez la possibilité de faire en sorte qu'un texte suive un chemin au format SVG de la forme que vous voulez avec la classe **TextPath**. Voici un exemple simple :

**Code : JavaScript**

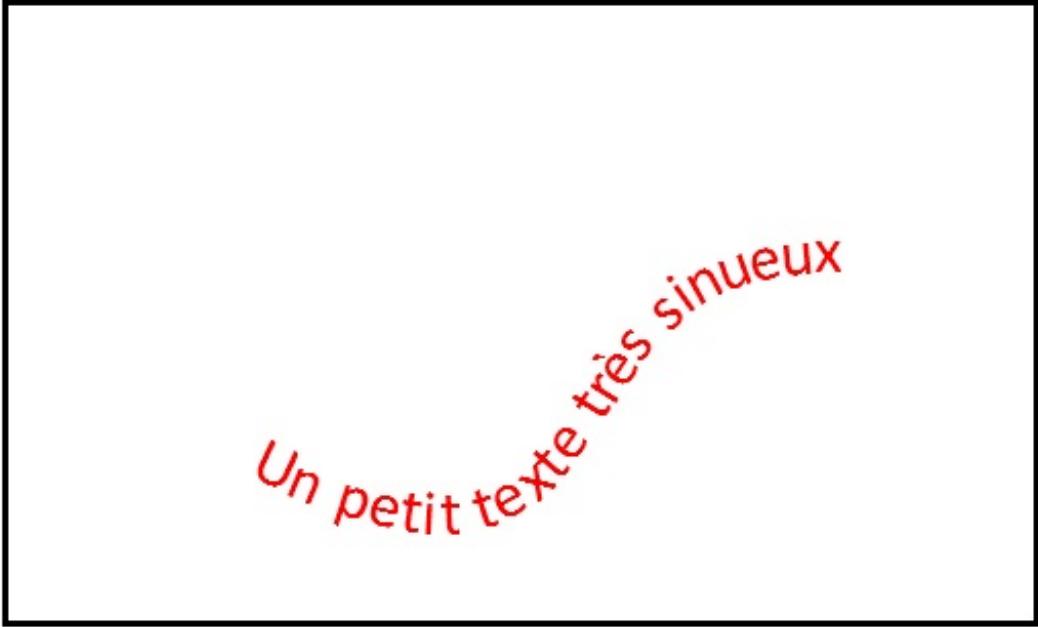
```
window.onload = function() {
  var scène = new Kinetic.Stage({
    container: "kinetic",
    width: 500,
    height: 300
  });

  var calque = new Kinetic.Layer();

  var textpath = new Kinetic.TextPath({
    textFill: 'red',
    fontSize: '12',
    text: "Un petit texte très sinueux",
    data: "M 60,110 C 160,160 120,60 200,60",
    scale: 2
  });

  calque.add(textpath);
  scène.add(calque);
};
```

Tester !



Un petit texte très sinueux

Si vous posez des questions sur la mystérieuse valeur "**M 60,110 C 160,160 120,60 200,60**", je vous conseille [cette lecture bénéfique](#) même si elle n'est pas vraiment très digeste 😊.

Maintenant que notre infrastructure est en place nous allons pouvoir commencer à dessiner.

## Tracés et images

Maintenant, nous allons voir comment dessiner en traçant des lignes, des courbes et également comment afficher des images.

### Lignes

#### Dessiner une ligne

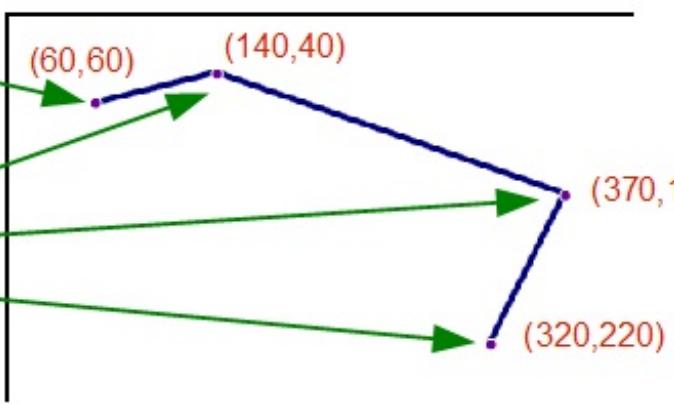
Voici un exemple :

Code : JavaScript

```
var ligne = new Kinetic.Line({
    points: [60, 60, 140, 40, 370, 120, 320, 220],
    stroke: "navy",
    strokeWidth: 4,
});
calque.add(ligne);
```

Tester !

```
var ligne = new Kinetic.Line({
    points: [
        60, 60,
        140, 40,
        370, 120,
        320, 220
    ],
    stroke: "navy",
    strokeWidth: 4,
});
```



### Jointures

Il y a 3 façons de raccorder les lignes qui sont définies avec la propriété **lineJoin**. Les valeurs sont :

- miter (valeur par défaut)
- round
- bevel

Pour visualiser la différence, voici 3 lignes identiques avec des jointures différentes :

Code : JavaScript

```
var ligne1 = new Kinetic.Line({
    points: [60, 80, 140, 40, 220, 80],
    stroke: "hotpink",
    strokeWidth: 30,
    lineJoin: "miter"
});
```

```

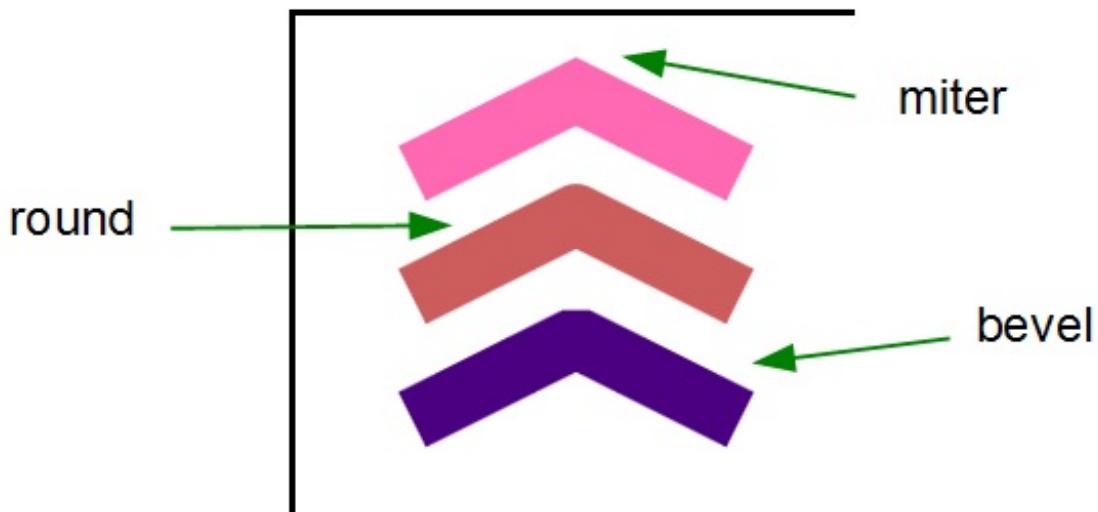
var ligne1 = new Kinetic.Line({
  points: [60, 140, 140, 100, 220, 140],
  stroke: "indianred",
  strokeWidth: 30,
  lineJoin: "round"
});

var ligne2 = new Kinetic.Line({
  points: [60, 200, 140, 160, 220, 200],
  stroke: "indigo",
  strokeWidth: 30,
  lineJoin: "bevel"
});

calque.add(ligne1);
calque.add(ligne2);
calque.add(ligne3);

```

Tester !



## Extrémités de lignes

Il y a 3 façons de finir les lignes qui sont définies avec la propriété **lineCap**. Les valeurs sont :

- butt (valeur par défaut)
- round
- square

Pour visualiser la différence voilà 3 lignes identiques avec des extrémités différentes :

### Code : JavaScript

```

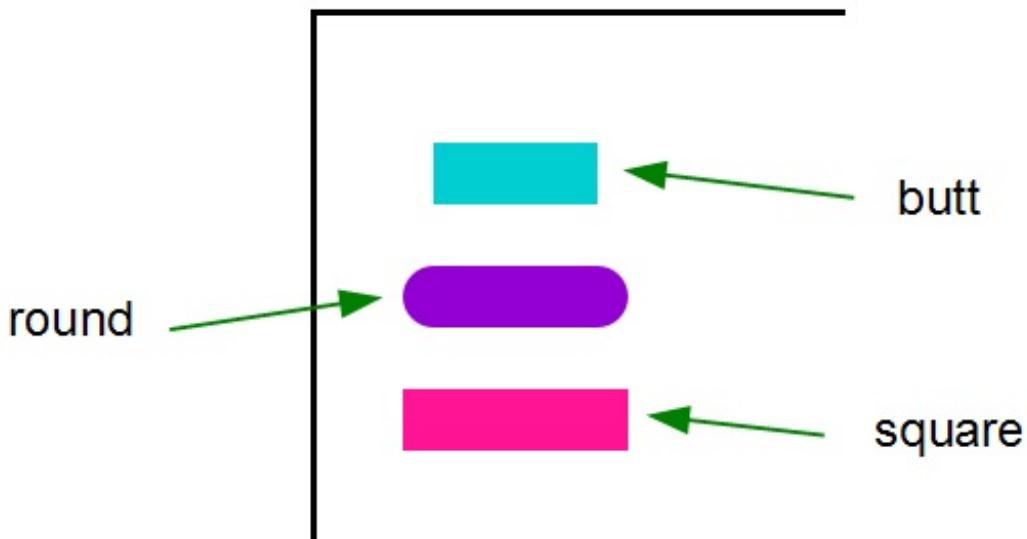
var ligne1 = new Kinetic.Line({
  points: [60, 80, 140, 80],
  stroke: "darkturquoise",
  strokeWidth: 30,
  lineCap: "butt"
});

var ligne2 = new Kinetic.Line({
  points: [60, 140, 140, 140],
  stroke: "darkviolet",
  strokeWidth: 30,
  lineCap: "round"
});

```

```
) ;  
  
var ligne3 = new Kinetic.Line({  
    points: [60, 200, 140, 200],  
    stroke: "deeppink",  
    strokeWidth: 30,  
    lineCap: "square"  
});  
  
calque.add(ligne1);  
calque.add(ligne2);  
calque.add(ligne3);
```

Tester !



## Interruptions de lignes

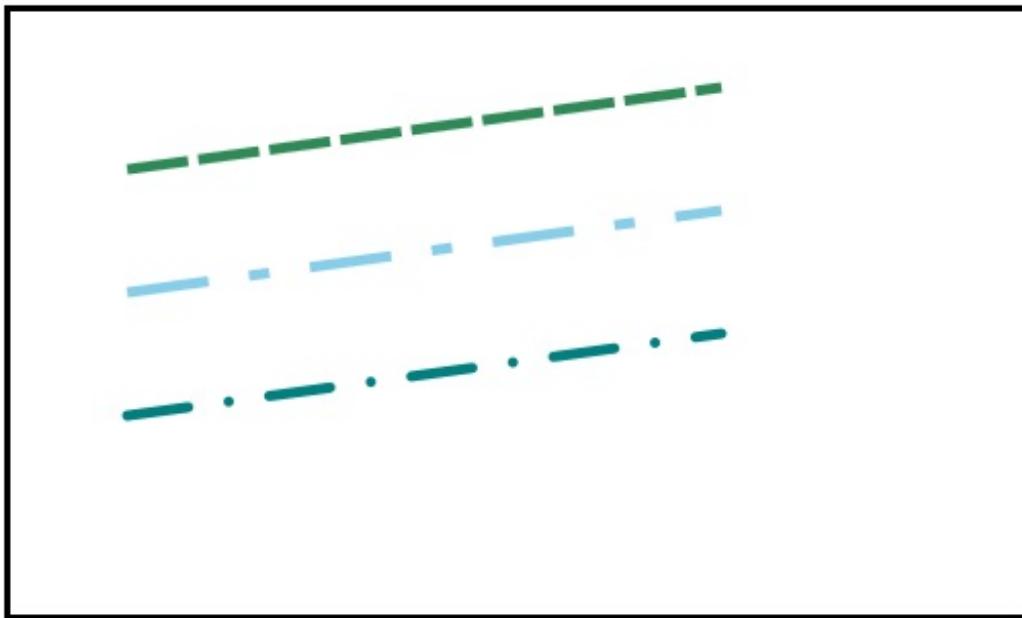
Nous avons vu jusque-là des lignes continues. Il est aussi possible de créer des interruptions pour obtenir des pointillés ou traits d'axe avec la propriété **dashArray**. Voici un exemple :

Code : JavaScript

```
var ligne1 = new Kinetic.Line({  
    points: [60, 80, 350, 40],  
    stroke: "seagreen",  
    strokeWidth: 5,  
    dashArray: [30, 5]  
});  
  
var ligne2 = new Kinetic.Line({  
    points: [60, 140, 350, 100],  
    stroke: "skyblue",  
    strokeWidth: 5,  
    dashArray: [40, 20, 10, 20]  
});  
  
var ligne3 = new Kinetic.Line({  
    points: [60, 200, 350, 160],  
    stroke: "teal",  
    strokeWidth: 5,  
    lineCap: "round",  
    dashArray: [30, 20, 0, 20]  
});
```

```
calque.add(ligne1);
calque.add(ligne2);
calque.add(ligne3);
```

Tester !



Il suffit de définir un tableau de valeurs pour la propriété **dashArray**. Les valeurs sont par couples de lignes visibles puis invisibles. Par exemple la ligne 1 ci-dessus a des zones visibles de 30 pixels suivies de zones invisibles de 5 pixels. Pour la ligne 2 on a deux couples de valeurs, ce qui permet d'obtenir un effet de trait d'axe. Pour la ligne 3 il a fallu définir un **lineCap** arrondi pour rendre les points visibles (valeurs à 0).

## Shape : une classe à tout faire

### Arcs

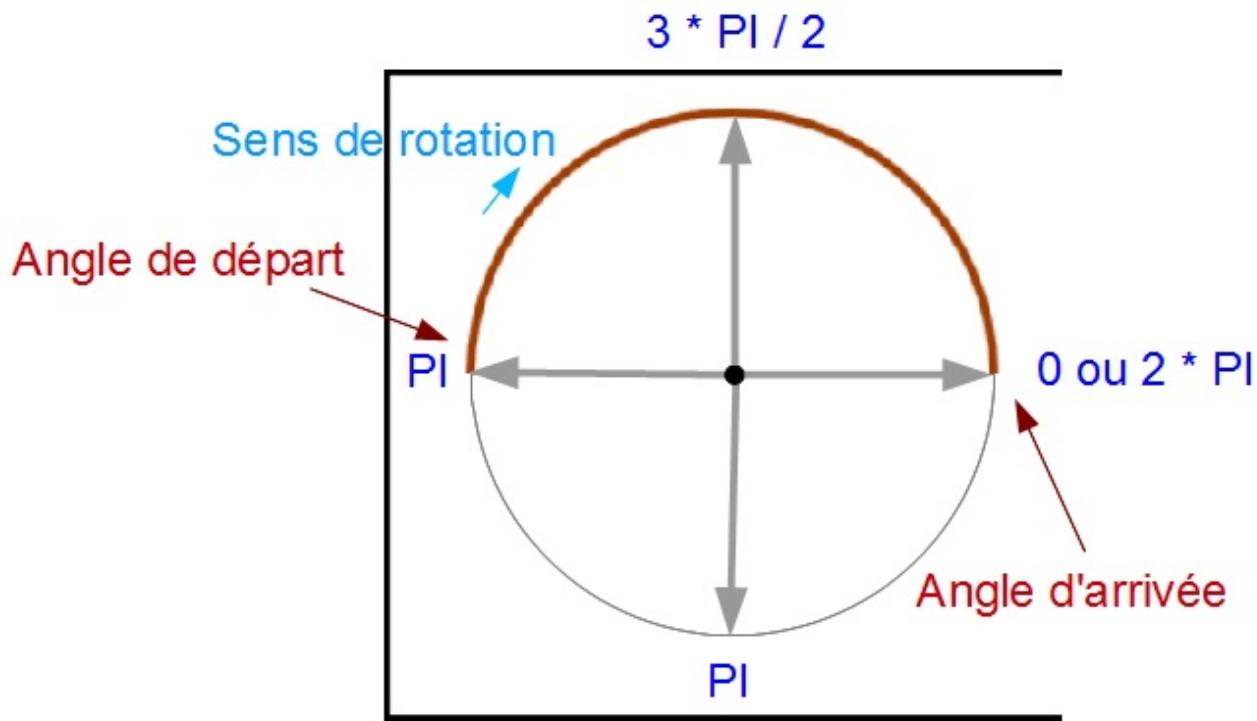
#### *arc()*

Au chapitre précédent nous avons vu comment créer des cercles en jouant sur le rayon, le style du trait et le remplissage. Maintenant, voyons comment dessiner des arcs de cercle, autrement dit des morceaux de cercle. La classe utilisée est maintenant **Shape** qui est une classe très générale qui permet d'utiliser les commandes génériques de l'API de **Canvas**. Cette classe attend comme propriété une fonction de dessin. **Canvas** propose deux méthodes pour dessiner des arcs de cercle, voici la première :

#### Code : JavaScript

```
var arc = new Kinetic.Shape({
  drawFunc: function(context) {
    context.arc(160, 140, 120, Math.PI, 2 * Math.PI);
    this.stroke(context);
  },
  stroke: "rgb(160,60,0)",
  strokeWidth: 4,
});
calque.add(arc);
```

Tester !



Pour le dessin d'un arc, la librairie utilise la méthode de l'API de **Canvas**. Le contexte est récupéré comme paramètre de la fonction de dessin :

**Code : JavaScript**

```
drawFunc: function(context)
```

Le contexte est ensuite utilisé pour appeler la méthode **arc** de **Canvas** qui est de cette forme :

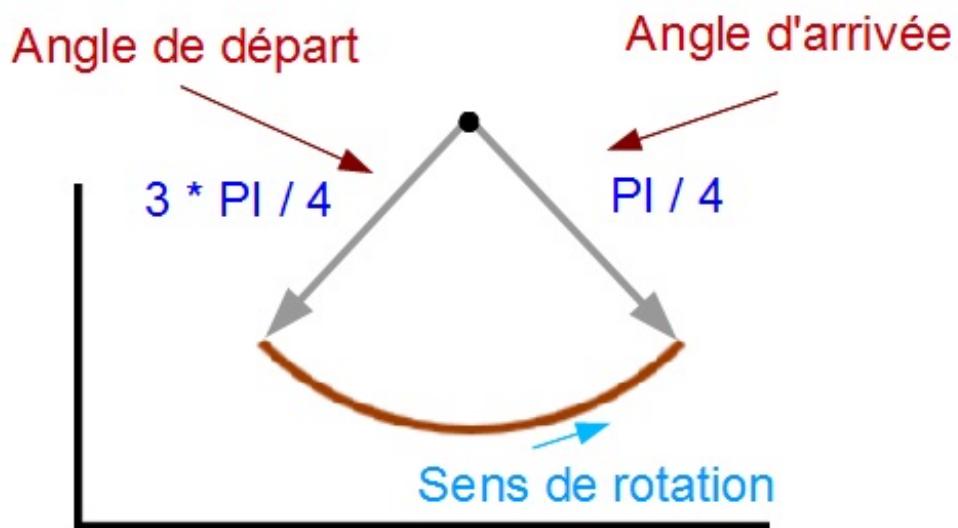
```
context.arc(Position X du centre, Position Y du centre, Rayon, Angle de départ, Angle d'arrivée, Sens de rotation (optionnel));
```

Voyons maintenant un autre exemple avec une rotation inversée et d'autres angles :

**Code : JavaScript**

```
var arc = new Kinetic.Shape({  
    drawFunc: function(context) {  
        context.arc(160, 140, 120, 3 * Math.PI / 4, Math.PI / 4, true);  
        this.stroke(context);  
    },  
    stroke: "rgb(160,60,0)",  
    strokeWidth: 4,  
});  
calque.add(arc);
```

Tester !



### *arcTo()*

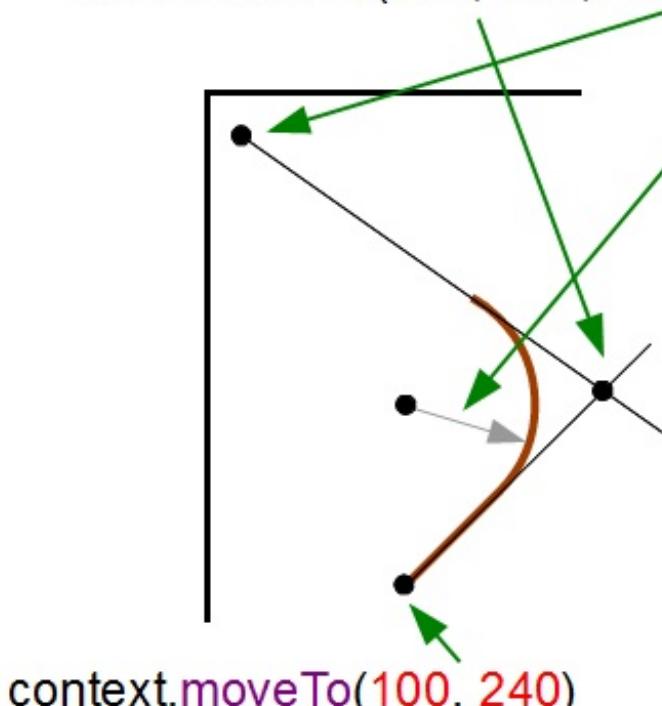
La deuxième possibilité pour dessiner un arc de cercle avec **Canvas** est la méthode **arcTo** :

#### Code : JavaScript

```
var arc = new Kinetic.Shape({
  drawFunc: function(context) {
    context.moveTo(100, 240);
    context.arcTo(200, 140, 20, 40, 60);
    this.stroke(context);
  },
  stroke: "rgb(160,60,0)",
  strokeWidth: 4,
});
calque.add(arc);
```

Tester !

`context.arcTo(200, 140, 20, 40, 60)`



Cette méthode est moins évidente à comprendre que la précédente. Voyons ça de plus près. On commence par positionner le stylo sur le point de référence de la première tangente :

**Code : JavaScript**

```
context.moveTo(100, 240);
```

Ensuite on utilise la méthode **arcTo** qui référence dans l'ordre : le point d'intersection des deux tangentes, le point de référence de la deuxième tangente et le rayon de courbure de l'arc :

**Code : JavaScript**

```
context.arcTo(200, 140, 20, 40, 60);
```

## Courbes

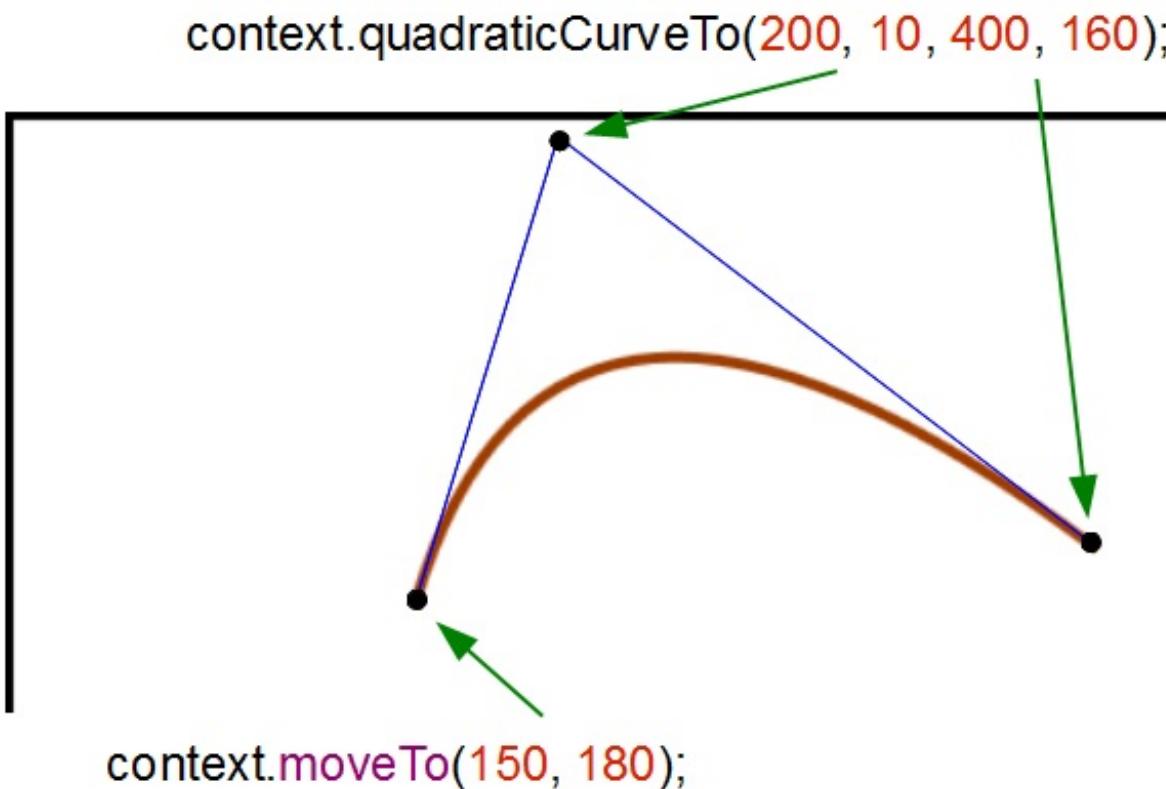
### *Courbe quadratique*

Un arc de cercle c'est bien, mais il faut parfois dessiner des courbes un peu plus élaborées. Une première option consiste à tracer une courbe quadratique :

**Code : JavaScript**

```
var arc = new Kinetic.Shape({
  drawFunc: function(context) {
    context.moveTo(150, 180);
    context.quadraticCurveTo(200, 10, 400, 160);
    this.stroke(context);
  },
  stroke: "rgb(160,60,0)",
  strokeWidth: 4,
});
calque.add(arc);
```

Tester !



On commence par positionner le premier point de la courbe avec la commande :

**Code : JavaScript**

```
context.moveTo(150, 180);
```

Il faut imaginer que vous avez un stylo à disposition et que vous le positionnez sur ce point. Ensuite la courbe est dessinée en précisant le point d'arrivée et le point de contrôle (intersection des deux tangentes) :

**Code : JavaScript**

```
context.quadraticCurveTo(200, 10, 400, 160);
```

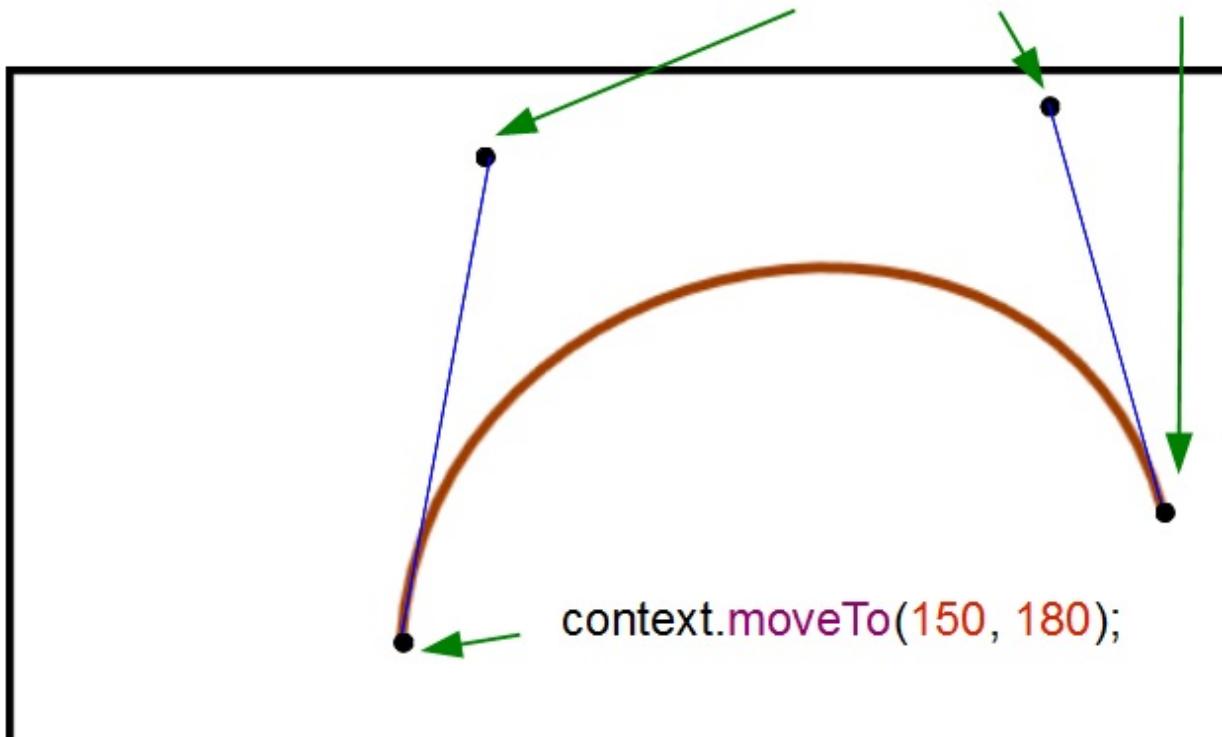
Si vous désirez en savoir plus sur les courbes quadratiques, il y a pas mal d'information sur Internet par exemple [ici](#).

### Courbe de Bézier

On peut créer des courbes encore plus élaborées en utilisant des courbes de Bézier. Cette fois on a deux points de contrôle :

**Code : JavaScript**

```
var arc = new Kinetic.Shape({
  drawFunc: function(context) {
    context.moveTo(150, 220);
    context.bezierCurveTo(160, 60, 400, 20, 440, 170);
    this.stroke(context);
  },
  stroke: "rgb(160,60,0)",
  strokeWidth: 4,
});
calque.add(arc);
```

[Tester !](#)`context.bezierCurveTo(160, 60, 400, 20, 440, 170);`

## Combinaisons

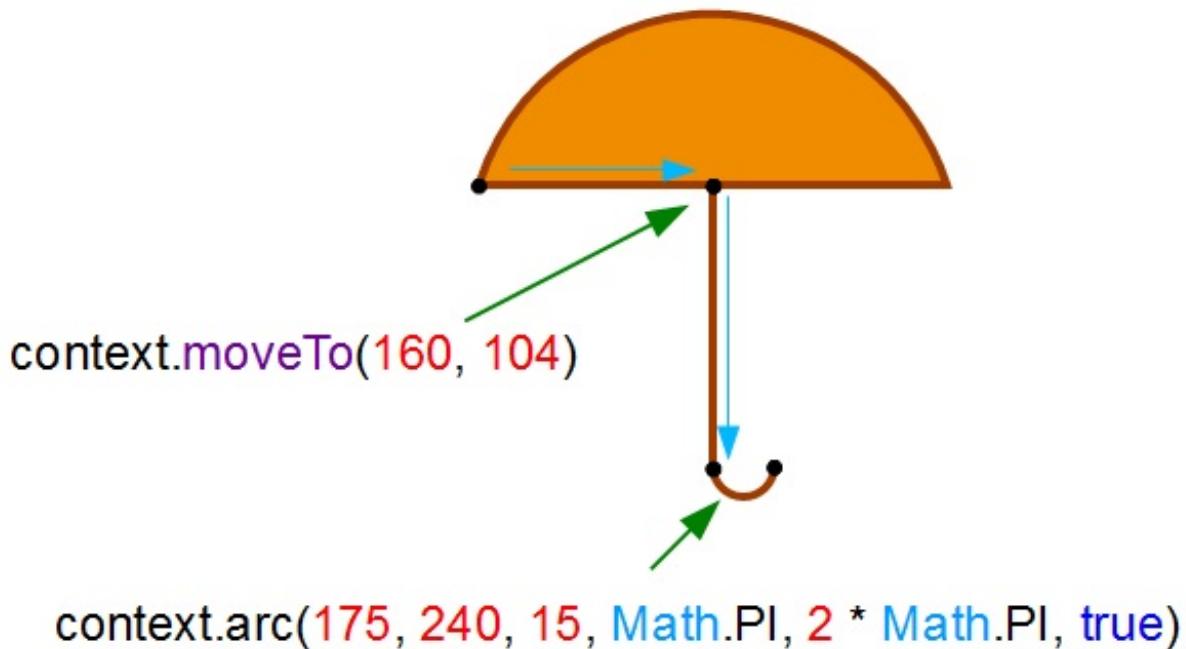
### *Un parapluie*

On va partir de la base de la forme précédente pour dessiner un parapluie. Voici le code complet (le code ajouté est souligné) :

**Code : JavaScript**

```
var parapluie = new Kinetic.Shape({
  drawFunc: function(context) {
    context.beginPath();
    context.arc(160, 140, 120, 1.1 * Math.PI, 1.9 * Math.PI);
    context.closePath();
    this.fill(context),
    context.moveTo(160, 104),
    context.arc(175, 240, 15, Math.PI, 2 * Math.PI, true);
    this.stroke(context);
  },
  fill: "rgb(240,140,0)",
  stroke: "rgb(160,60,0)",
  strokeWidth: 4,
});
calque.add(parapluie);
```

[Tester !](#)



A la fin du dessin du morceau de disque, le stylo se trouve à l'extrême gauche (représenté par le point noir). Il faut donc le déplacer au milieu pour tracer le manche :

**Code : JavaScript**

```
context.moveTo(160, 104),
```

Ensuite le fait de dessiner l'arc de cercle de la crosse provoque la liaison par un trait entre la position du stylo et le début de l'arc. Il est donc inutile de se préoccuper de ce trait.

**Un panier**

Voyons maintenant un autre exemple simple avec un panier stylisé composé d'une courbe quadratique pour l'anse et un polygone pour le corps :

**Code : JavaScript**

```
var anse = new Kinetic.Shape({
  drawFunc: function(context) {
    context.moveTo(100, 120);
    context.quadraticCurveTo(180, -80, 260, 120);
    this.stroke(context);
  },
  stroke: "orange",
  strokeWidth: 10,
});

var corps = new Kinetic.Polygon({
  points: [60, 110, 300, 110, 280, 240, 80, 240],
  fill: "#00D2FF",
  fill: "#b90",
});

calque.add(anse);
calque.add(corps);
```

Tester !



## Images

### Exemple de base

Voici un exemple :

#### Code : JavaScript

```
var img = new Image();
img.onload = function() {
    var image = new Kinetic.Image({
        x: 40,
        y: 40,
        image: img,
        width: 120,
        height: 120
    });

    calque.add(image);
    scène.add(calque);
};

img.src = "images/kineticjs-bolt-sticker.png";
```

Tester !

```
var image = new Kinetic.Image({
```

x: 40,

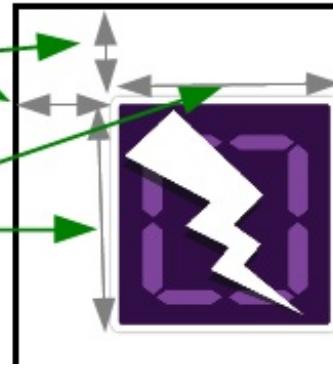
y: 40,

image: img,

width: 120,

height: 120

```
});
```



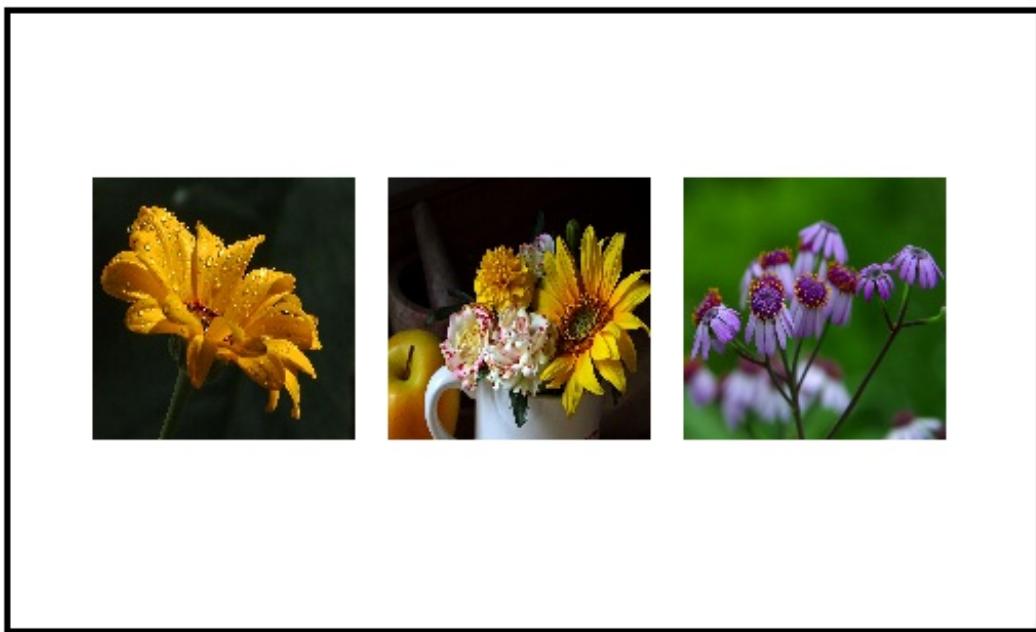
## Charger plusieurs images

Lorsqu'il y a plusieurs images à charger il faut prendre la précaution d'attendre le chargement complet des images avant d'initialiser les objets **image** de **KineticJS**. Voici un exemple avec 3 images :

### Code : JavaScript

```
// Chargement des images
function load_images(sources, callback) {
    var images = new Array();
    var loadedImages = 0;
    sources.forEach(function(value, index) {
        images[index] = new Image();
        images[index].onload = function() {
            if(++loadedImages >= sources.length) callback(images);
        };
        images[index].src = value;
    });
}
// Initialisation de la scène
function init_scène(images) {
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });
    var calque = new Kinetic.Layer();
    images.forEach(function(value, index) {
        var image = new Kinetic.Image({
            x: 40 + index * 144,
            y: 80,
            image: value,
            width: 128,
            height: 128
        });
        calque.add(image);
    });
    scène.add(calque);
}
// Chargement de la page
window.onload = function() {
    var sources_img = [
        "images/img01.jpg",
        "images/img02.jpg",
        "images/img03.jpg",
    ];
    load_images(sources_img, init_scène);
}
```

Tester !



## Utilisation d'un plugin pour le chargement des images

J'ai trouvé un petit plugin intéressant pour le chargement des images. Vous pouvez le charger [ici](#). Lorsqu'on l'utilise le code devient tout de suite plus léger pour le même résultat que le code précédent :

### Code : JavaScript

```
window.onload = function() {
  var scène = new Kinetic.Stage({
    container: "kinetic",
    width: 500,
    height: 300
  });
  var calque = new Kinetic.Layer();
  scène.add(calque);

  var toLoad = [
    {id:"myImage1",src:"images/img01.jpg"},
    {id:"myImage2",src:"images/img02.jpg"},
    {id:"myImage3",src:"images/img03.jpg"},
  ];

  var loader = new Kinetic.Loader(toLoad);

  loader.onComplete(function(){
    var index = 0;
    for(var key in Kinetic.Assets) {
      var image = new Kinetic.Image({
        x: 40 + index,
        y: 80,
        image: Kinetic.Assets[key],
        width: 128,
        height: 128
      });
      index += 144;
      calque.add(image);
    }
    calque.draw();
  });

  loader.load();
}
```

Tester !

Notez que ce plugin permet aussi de connaître le pourcentage de chargement, le nombre d'images chargées et le nombre total d'images à charger avec son événement **onProgress**.

## Style

Dans cette partie, nous allons voir comment remplir des formes avec des couleurs unies ou des dégradés et comment donner des effets de transparence et des ombrages.

### Traits et remplissage

#### Traits

Nous avons déjà vu comment dessiner des traits. voici un nouvel exemple simple :

##### Code : JavaScript

```
var ligne = new Kinetic.Line({
    points: [60, 60, 360, 60],
    stroke: "#aa4",
    strokeWidth: 6
});
calque.add(ligne);
```

Tester !



L'épaisseur du trait est donnée par la propriété **strokeWidth** et sa couleur par la propriété **stroke**. Voici un nouveau code pour un résultat identique :

##### Code : JavaScript

```
var ligne = new Kinetic.Line({
    points: [60, 60, 360, 60]
});

ligne.setStrokeWidth(6);
ligne.setStroke("#aa4");

calque.add(ligne);
```

Tester !

Cette fois on a pas utilisé les propriétés dans le constructeur, mais des setters ensuite dans le code. Pratiquement toutes les propriétés dans **KineticJS** bénéficient de getters et setters.

Voici un autre exemple :

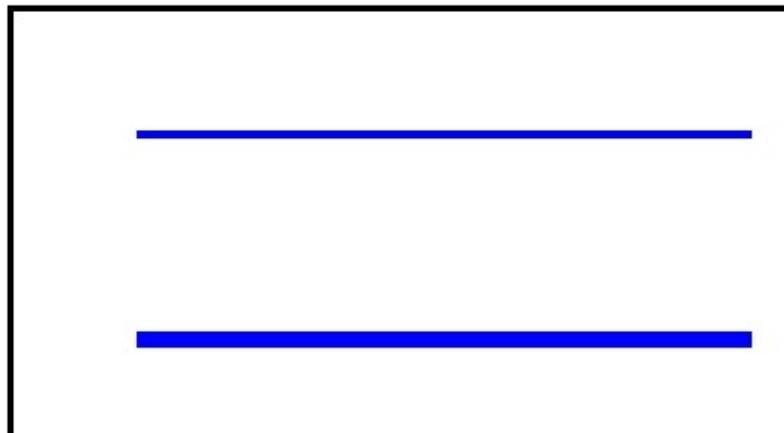
##### Code : JavaScript

```
var ligne1 = new Kinetic.Line({
    points: [60, 60, 360, 60],
    stroke: "blue",
    strokeWidth: 4
});

var ligne2 = new Kinetic.Line({
    points: [60, 160, 360, 160],
```

```
) ;  
  
ligne2.setStrokeWidth(ligne1.getStrokeWidth() + 4);  
ligne2.setStroke(ligne1.getStroke());  
  
calque.add(ligne1);  
calque.add(ligne2);
```

Tester !



## Remplissage uni

Nous avons également vu précédemment comment effectuer un remplissage uni. Par exemple comme ceci :

**Code : JavaScript**

```
var rectangle = new Kinetic.Rect({  
    x: 50,  
    y: 50,  
    width: 140,  
    height: 60,  
    fill: "blue"  
});  
calque.add(rectangle);
```

Tester !



Il suffit de renseigner la propriété **fill** avec au choix : un nom de couleur, une valeur hexadécimale ou une valeur obtenue avec la fonction **rgb**. Voici un nouveau code pour un résultat identique :

**Code : JavaScript**

```
var rectangle = new Kinetic.Rect({  
    x: 50,
```

```
y: 50,  
width: 140,  
height: 60  
});  
  
rectangle.setFill("blue");  
  
calque.add(rectangle);
```

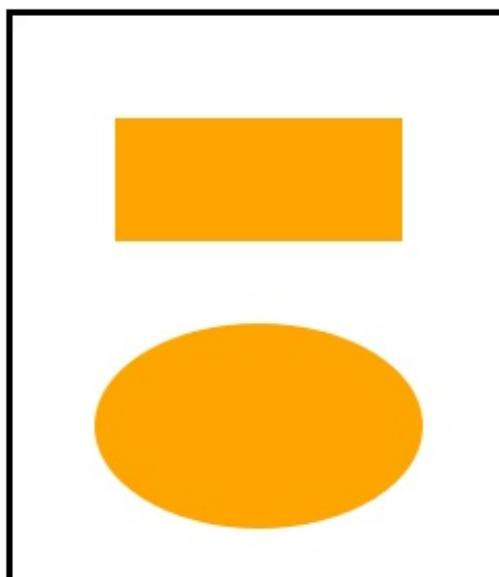
Tester !

Cette fois on a fait appel à un setter. Voici un autre exemple :

**Code : JavaScript**

```
var rectangle = new Kinetic.Rect({  
    x: 50,  
    y: 50,  
    width: 140,  
    height: 60,  
    fill: "orange"  
});  
  
var ellipse = new Kinetic.Ellipse({  
    x: 120,  
    y: 200,  
    radius: {  
        x: 80,  
        y: 50  
    }  
});  
  
ellipse.setFill(rectangle.getFill());  
  
calque.add(rectangle);  
calque.add(ellipse);
```

Tester !



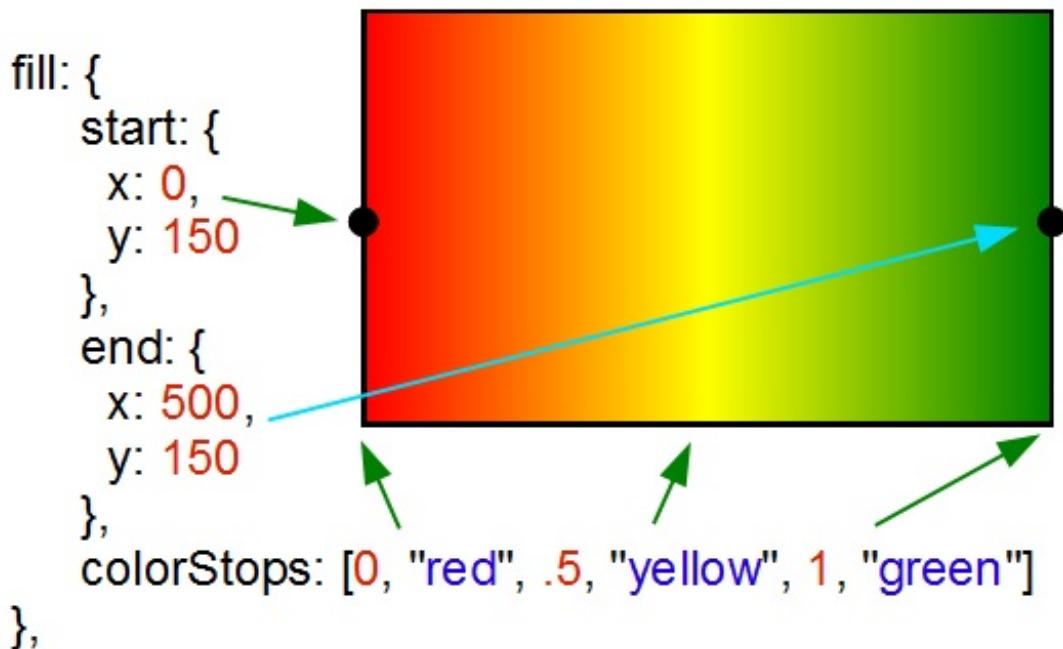
## Dégradé linéaire

Voici un premier exemple :

Code : JavaScript

```
var forme = new Kinetic.Rect({
    width: 500,
    height: 300,
    fill: {
        start: {
            x: 0,
            y: 150
        },
        end: {
            x: 500,
            y: 150
        },
        colorStops: [0, "red", .5, "yellow", 1, "green"]
    },
    stroke: "black",
    strokeWidth: 2,
});
calque.add(forme);
```

Tester !



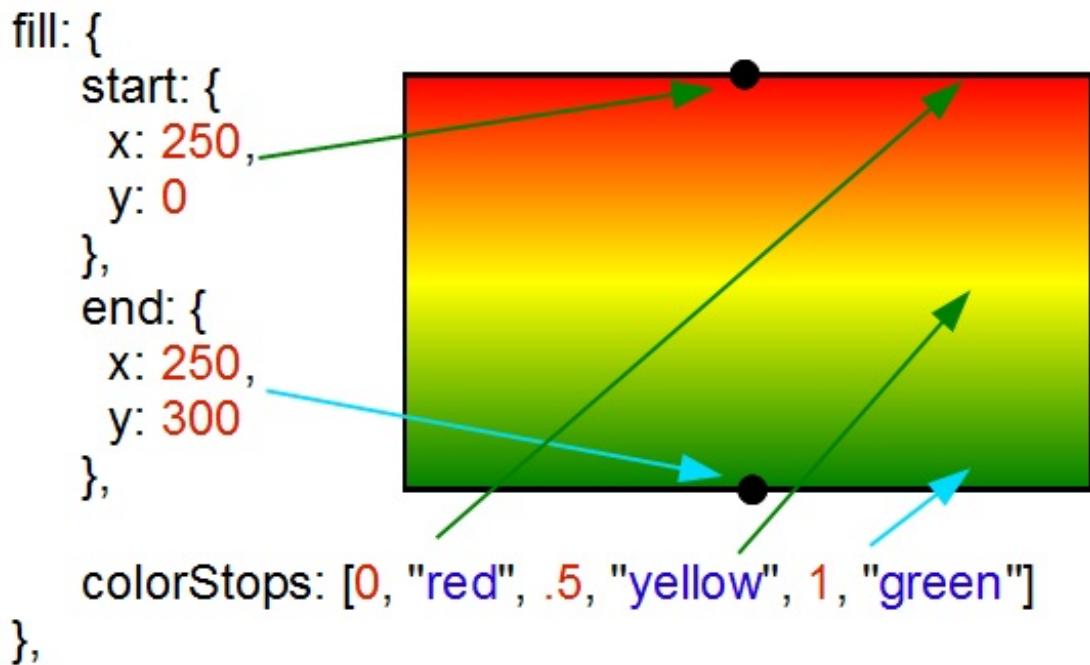
Il faut définir le point de départ avec la propriété **start** et le point d'arrivée avec la propriété **end**. Il faut aussi définir les couleurs utilisées et leur positionnement avec une valeur allant de 0 à 1.

On peut obtenir un dégradé vertical tout simplement en choisissant judicieusement les points de départ et d'arrivée :

Code : JavaScript

```
fill: {
    start: {
        x: 250,
        y: 0
    },
    end: {
        x: 250,
        y: 300
    },
    colorStops: [0, "red", .5, "yellow", 1, "green"]
}
```

}

[Tester !](#)

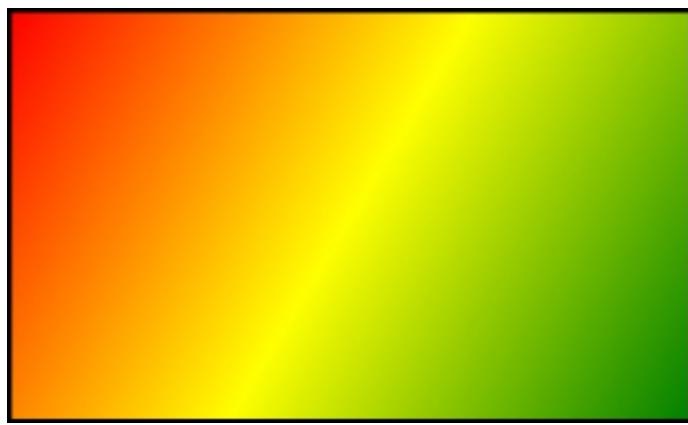
Et pourquoi pas en oblique ? Cette fois pour changer un peu j'utilise un setter :

**Code : JavaScript**

```
var forme = new Kinetic.Rect({
    width: 500,
    height: 300,
    stroke: "black",
    strokeWidth: 2,
});

forme.setFill({
    start: {
        x: 0,
        y: 0
    },
    end: {
        x: 500,
        y: 300
    },
    colorStops: [0, "red", .5, "yellow", 1, "green"]
});
calque.add(forme);
```

[Tester !](#)

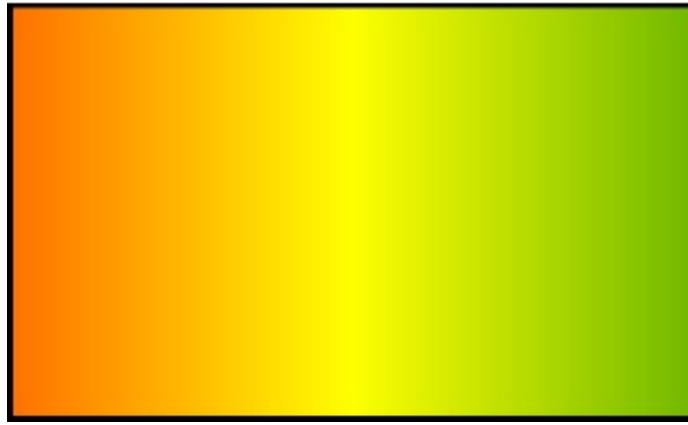


Jusque là j'ai fait en sorte que les points d'arrivée et de départ coïncident avec la limite du dessin. Que se passe-t-il si ce n'est pas le cas ?

**Code : JavaScript**

```
fill: {  
    start: {  
        x: -200,  
        y: 150  
    },  
    end: {  
        x: 700,  
        y: 150  
    },  
    colorStops: [0, "red", .5, "yellow", 1, "green"]  
}
```

Tester !



On se rend compte que le dégradé s'étale au-delà de la surface visible, on n'en voit donc qu'une partie.

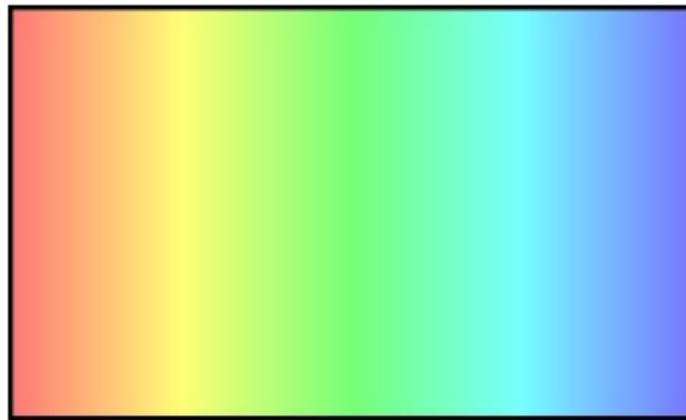
Est-ce qu'on peut faire un joli arc-en-ciel aussi ?

**Code : JavaScript**

```
fill: {  
    start: {  
        x: 0,  
        y: 150  
    },  
    end: {  
        x: 500,  
        y: 150  
    },  
    colorStops: [0, "red", .5, "yellow", 1, "green"]  
}
```

```
colorStops: [
  0, "#f77",
  .25, "#ff7",
  .5, "#7f7",
  .75, "#7ff",
  1, "#77f"
],
},
```

Tester !



Plutôt joli non ? 🍷

## Dégradé radial

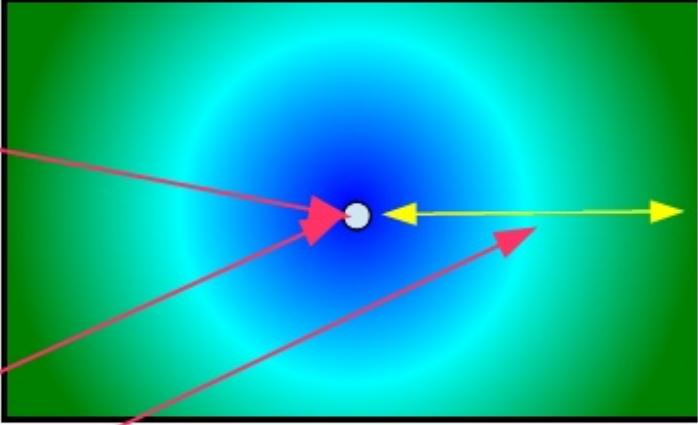
Voyons maintenant les dégradés radiaux. Voici un premier exemple :

Code : JavaScript

```
var forme = new Kinetic.Rect({
  width: 500,
  height: 300,
  fill: {
    start: {
      x: 250,
      y: 150,
      radius: 0
    },
    end: {
      x: 250,
      y: 150,
      radius: 250
    },
    colorStops: [0, "blue", .5, "cyan", 1, "green"]
  },
  stroke: "black",
  strokeWidth: 2,
}) ;
```

Tester !

```
fill: {  
    start: {  
        x: 250,  
        y: 150,  
        radius:0  
    },  
    end: {  
        x: 250,  
        y: 150,  
        radius: 250  
    },  
    colorStops: [0, "blue", .5, "cyan", 1, "green"]  
},
```



Ici encore on définit un départ et une arrivée, mais ce n'est plus seulement un point comme pour le dégradé linéaire, maintenant c'est un cercle dont on définit le centre et le rayon.

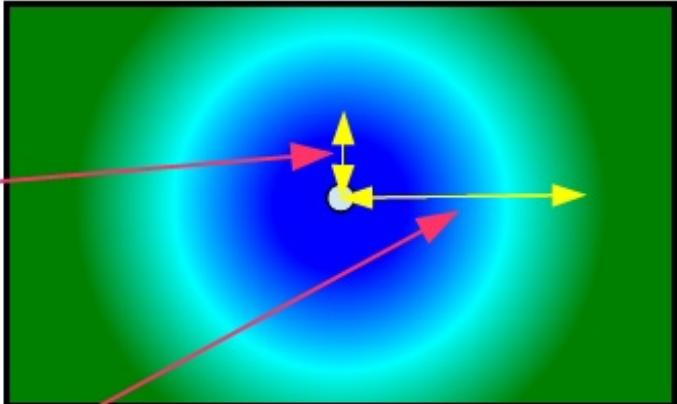
Voici un autre exemple en changeant les rayons :

**Code : JavaScript**

```
fill: {  
    start: {  
        x: 250,  
        y: 150,  
        radius: 50  
    },  
    end: {  
        x: 250,  
        y: 150,  
        radius: 200  
    },  
    colorStops: [0, "blue", .5, "cyan", 1, "green"]  
},
```

Tester !

```
fill: {  
    start: {  
        x: 250,  
        y: 150,  
        radius: 50  
    },  
    end: {  
        x: 250,  
        y: 150,  
        radius: 200  
    },  
    colorStops: [0, "blue", .5, "cyan", 1, "green"]  
},
```

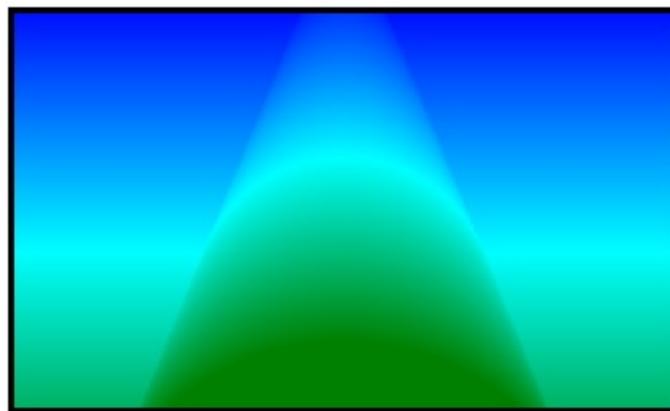


On peut créer des effets sympathiques en jouant sur les valeurs :

Code : JavaScript

```
fill: {  
    start: {  
        x: 250,  
        y: 0,  
        radius: 30  
    },  
    end: {  
        x: 250,  
        y: 450,  
        radius: 200  
    },  
    colorStops: [0, "blue", .5, "cyan", 1, "green"]  
},
```

Tester !



## Opacité et ombre

### Opacité

On peut rendre une forme transparente en jouant sur sa propriété **opacity** avec une valeur allant de 0 (totalement transparent) à 1

(opaque):

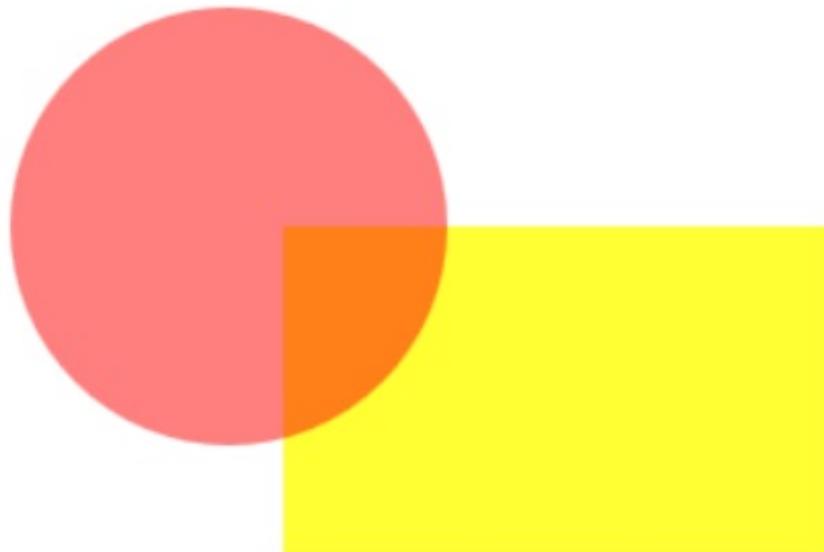
**Code : JavaScript**

```
var cercle = new Kinetic.Ellipse({
    x: 140,
    y: 100,
    radius: 80,
    fill: "#f00",
    opacity: .5
});

var rectangle = new Kinetic.Rect({
    x: 160,
    y: 100,
    width: 200,
    height: 120,
    fill: "yellow",
    opacity: .8
});

calque.add(rectangle);
calque.add(cercle);
```

Tester !



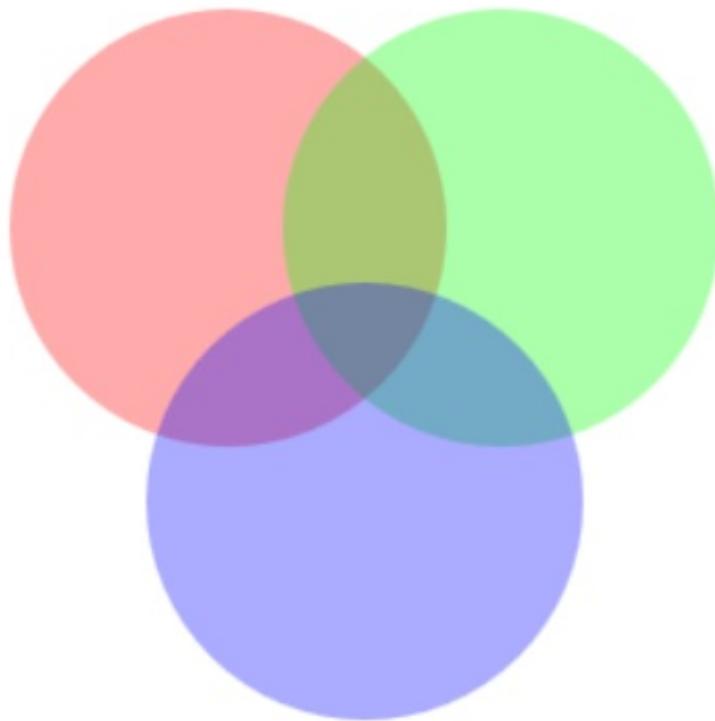
On peut utiliser les setters pour définir la valeur de la transparence comme je l'ai fait dans cet exemple évoquant la synthèse soustractive des couleurs :

**Code : JavaScript**

```
for(i = 0; i < 3; ++i) {
    var cercle = new Kinetic.Ellipse({
        radius: 80,
        opacity: .33
    });
    switch(i) {
        case 0:
            cercle.setX(140);
            cercle.setY(100);
            cercle.setFill("#f00");
            break;
        case 1:
```

```
cercle.setX(240);
cercle.setY(100);
cercle.setFill("#0f0");
break;
case 2:
  cercle.setX(190);
  cercle.setY(200);
  cercle.setFill("#00f");
}
calque.add(cercle);
}
```

Tester !



## Ombre

Il y a la propriété **shadow** pour créer une ombre :

Code : JavaScript

```
var cercle = new Kinetic.Ellipse({
  x: 100,
  y: 80,
  radius: 40,
  fill: "black",
  shadow: {
    color: "grey",
    blur: 12,
    offset: [8, 8],
    opacity: 0.7
  }
});

var rectangle = new Kinetic.Rect({
  x: 90,
  y: 100,
  width: 200,
```

```
height: 70,  
fill: "yellow",  
opacity: .7  
});  
  
rectangle.setShadow({  
color: "grey",  
blur: 12,  
offset: [8, 8],  
opacity: 0.7  
});  
  
calque.add(rectangle);  
calque.add(cercle);
```

Tester !



Pour le cercle j'ai utilisé directement la propriété dans le constructeur et pour le rectangle j'ai utilisé un setter.

La propriété **shadow** est un objet composé lui même de 4 propriétés :

Propriété	Effet
<b>color</b>	Couleur de l'ombre
<b>blur</b>	Flou de l'ombre
<b>offset</b>	Décalage de l'ombre (tableau avec les valeurs x et y)
<b>opacity</b>	Transparence de l'ombre

# Manipulations

Dessiner des formes, c'est bien, mais pouvoir les transformer, c'est encore mieux. Petite visite guidée...

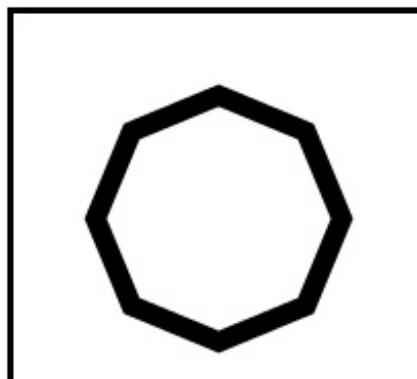
## Modifications Positionnement

Voici un exemple simple :

### Code : JavaScript

```
var poly = new Kinetic.RegularPolygon({  
    x: 100,  
    y: 100,  
    sides: 8,  
    radius: 60,  
    strokeWidth: 10  
});  
calque.add(poly);
```

Tester !



La position est donnée par les propriétés `x` et `y`. Voici un nouveau code pour un résultat identique :

### Code : JavaScript

```
var poly = new Kinetic.RegularPolygon({  
    sides: 8,  
    radius: 60,  
    strokeWidth: 10  
});  
  
poly.setX(100);  
poly.setY(100);  
  
calque.add(poly);
```

Tester !

Cette fois on a pas utilisé les propriétés dans le constructeur, mais des setters ensuite dans le code. Pratiquement toutes les propriétés dans **KineticJS** bénéficient de getters et setters.

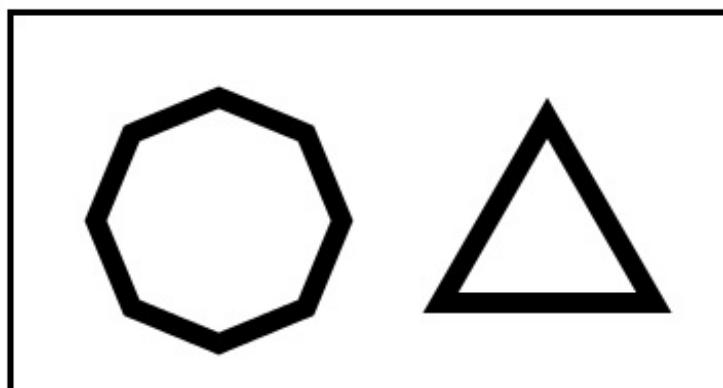
Voici un autre exemple cette fois avec un setter qui permet de définir simultanément `x` et `y` :

### Code : JavaScript

```
var poly1 = new Kinetic.RegularPolygon({  
    x: 100,
```

```
y: 100,  
sides: 8,  
radius: 60,  
strokeWidth: 10  
});  
  
var poly2 = new Kinetic.RegularPolygon({  
sides: 3,  
radius: 60,  
strokeWidth: 10  
});  
  
poly2.setPosition(poly1.getX() + 160, poly1.getY() + 10);  
  
calque.add(poly1);  
calque.add(poly2);
```

Tester !



## Rotation

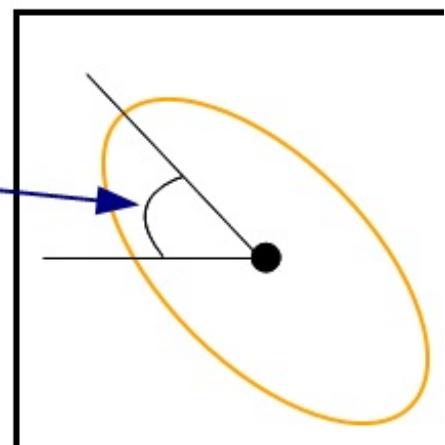
### *Rotation simple*

Code : JavaScript

```
var ellipse = new Kinetic.Ellipse({  
x: 120,  
y: 120,  
radius: [100, 50],  
stroke: "orange",  
strokeWidth: 5,  
rotation: Math.PI / 4  
});
```

Tester !

rotation: Math.PI / 4



La valeur de l'angle est en radian. Mais si vous préférez les degrés, il y a une méthode qui va vous plaire :

**Code : JavaScript**

```
var ellipse = new Kinetic.Ellipse({
  x: 120,
  y: 120,
  radius: [100, 50],
  stroke: "orange",
  strokeWidth: 5,
  rotationDeg: 45
});
```

Tester !

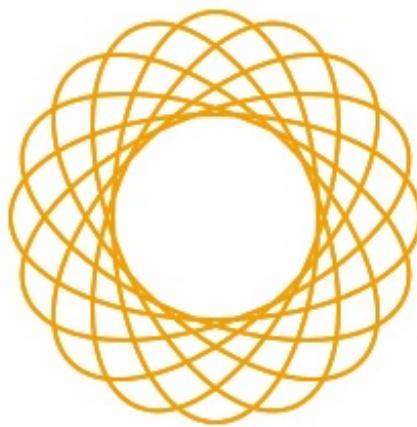
Avec un résultat identique au précédent 😊.

En jouant sur cette rotation on peut créer des figures intéressantes avec simplicité :

**Code : JavaScript**

```
for(var i = 0; i < 2 * Math.PI; i += Math.PI / 8) {
  calque.add(
    new Kinetic.Ellipse({
      x: 120,
      y: 120,
      radius: [100, 50],
      stroke: "rgb(240, 160, 0)",
      strokeWidth: 2,
      rotation: i
    })
  );
}
```

Tester !



Mais évidemment cette propriété prendra tout son sens avec les animations.

### Rotation avec offset

Les rotations que nous avons faites jusque-là étaient centrées sur le point de référence de la forme, pour une ellipse c'est le centre, pour un rectangle ce serait le coin supérieur gauche. Il y a la propriété **offset** pour décaler ce point de référence, c'est un tableau avec les valeurs **x** et **y** de décalage :

#### Code : JavaScript

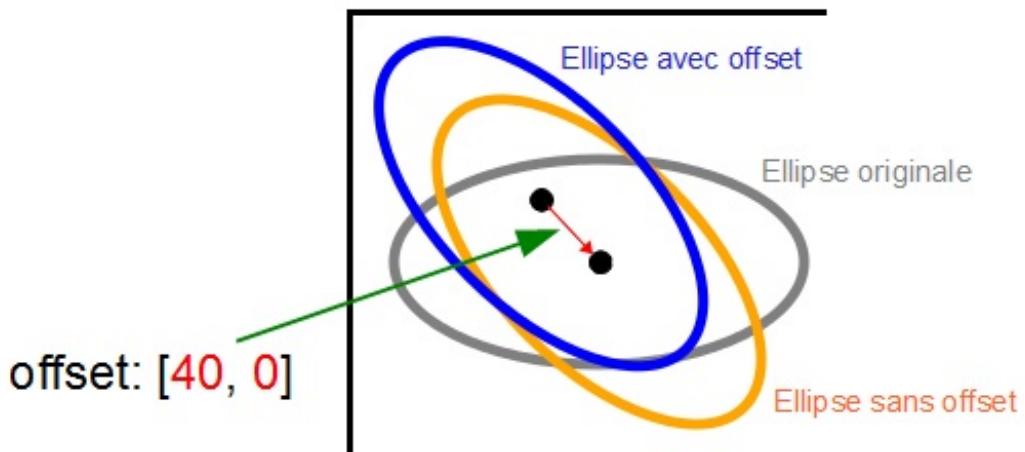
```
var ellipse0 = new Kinetic.Ellipse({
  x: 120,
  y: 120,
  radius: [100, 50],
  stroke: "grey",
  strokeWidth: 5,
});

var ellipsel = new Kinetic.Ellipse({
  x: 120,
  y: 120,
  radius: [100, 50],
  stroke: "orange",
  strokeWidth: 5,
  rotationDeg: 45
});

var ellipse2 = new Kinetic.Ellipse({
  x: 120,
  y: 120,
  radius: [100, 50],
  stroke: "blue",
  strokeWidth: 5,
  rotationDeg: 45,
  offset: [40, 0]
});

calque.add(ellipse0);
calque.add(ellipsel);
calque.add(ellipse2);
```

Tester !

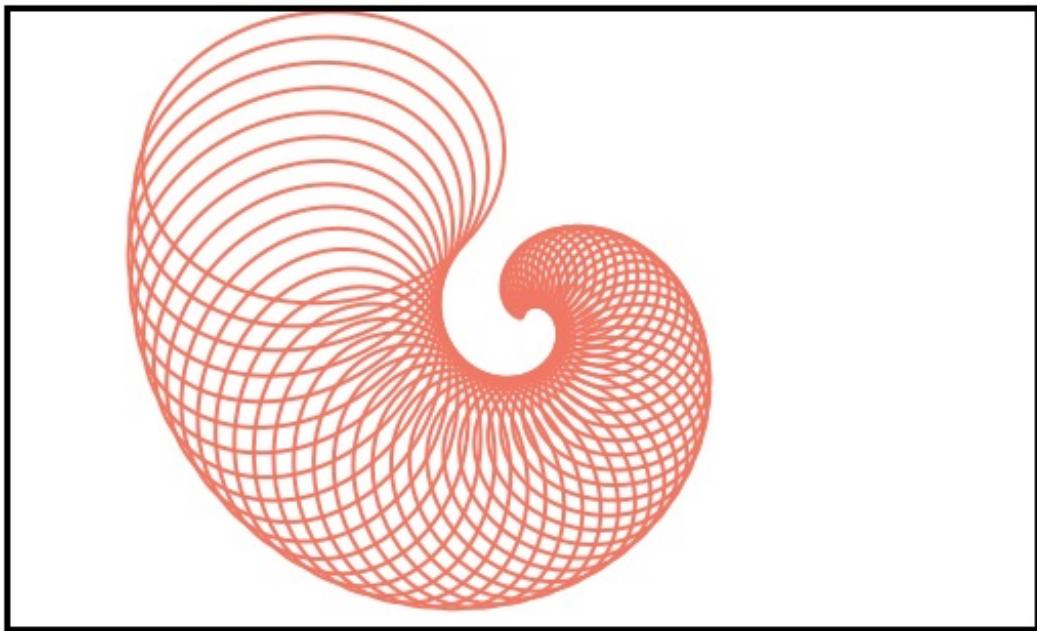


Ça permet d'envisager de jolies figures en jouant avec tous ces paramètres :

Code : JavaScript

```
for(var i = 0; i < 360; i += 6) {  
    calque.add(  
        new Kinetic.Ellipse({  
            x: 250,  
            y: 150,  
            radius: [i / 4, i / 5],  
            stroke: "rgb(240, 120, 100)",  
            strokeWidth: 2,  
            rotationDeg: i,  
            offset: i / 4  
        })  
    );  
}
```

Tester !



Remarquez que si on définit une seule valeur pour l'offset, et non plus un tableau de deux valeurs comme précédemment, elle est prise en compte pour x et y.

## Dimension

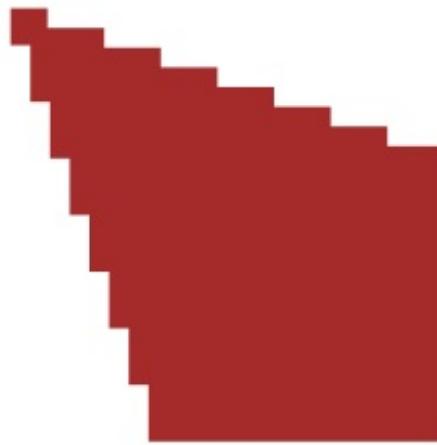
### Principe de base

Voici un premier exemple :

#### Code : JavaScript

```
for(var i = 0; i < 1; i += .12) {
    calque.add(new Kinetic.Rect({
        x: 90 + i * 80,
        y: 40 + i * 80,
        width: 150,
        height: 150,
        fill: "brown",
        scale: i
    })
}
scène.add(calque);
```

Tester !



Ici j'ai augmenté progressivement la taille d'un rectangle en le décalant, ce qui crée cette forme en créneaux. La propriété **scale** attend soit une valeur unique, auquel cas elle affecte le rapport de proportion aux deux dimensions, soit un tableau indiquant le rapport pour **x** et **y**.

### Changement dynamique

On peut agir de façon dynamique sur cette propriété (comme sur toutes les autres) avec un setter. Voici un exemple :

#### Code : HTML

```
<script>
window.onload = function() {
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

    var calque = new Kinetic.Layer();

    var rectangle = new Kinetic.Rect({
        x: 90,
        y: 40,
        width: 300,
```

```
        height: 200,
        fill: "orange"
    });

calque.add(rectangle);
scène.add(calque);

var petit = document.getElementById("petit");
var moyen = document.getElementById("moyen");
var grand = document.getElementById("grand");

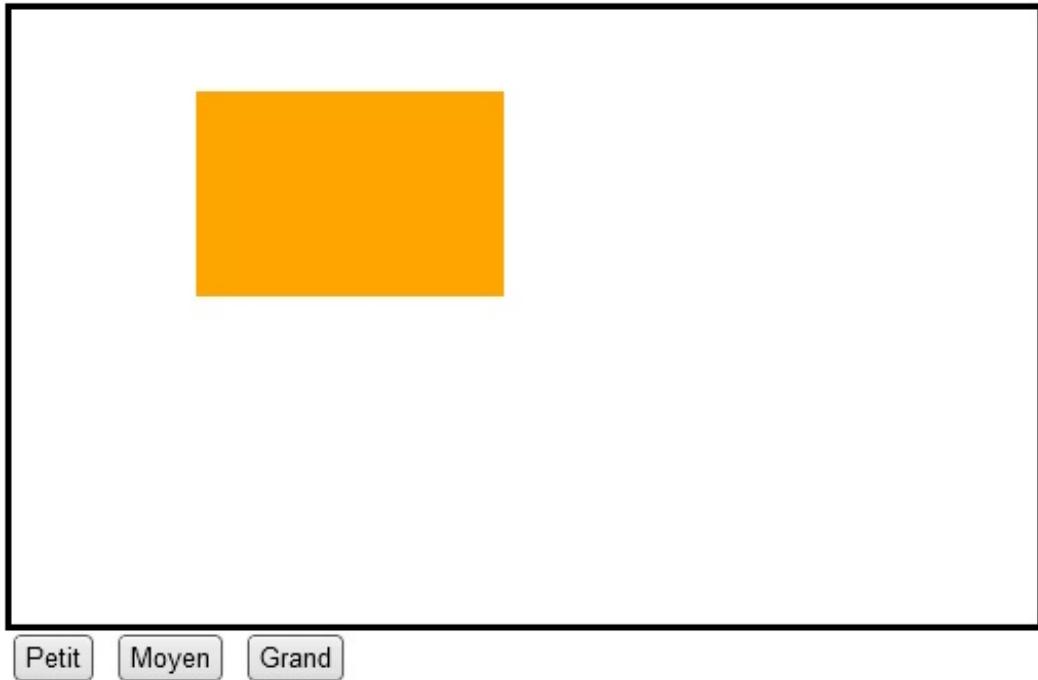
petit.onclick = function() {
    rectangle.setScale(.2);
    calque.draw();
};

moyen.onclick = function() {
    rectangle.setScale(.5);
    calque.draw();
};

grand.onclick = function() {
    rectangle.setScale(1);
    calque.draw();
};

</script>
</head>
<body>
<div id="kinetic"></div>
<input id="petit" type="button" value="Petit">
<input id="moyen" type="button" value="Moyen">
<input id="grand" type="button" value="Grand">
</body>
```

Tester !



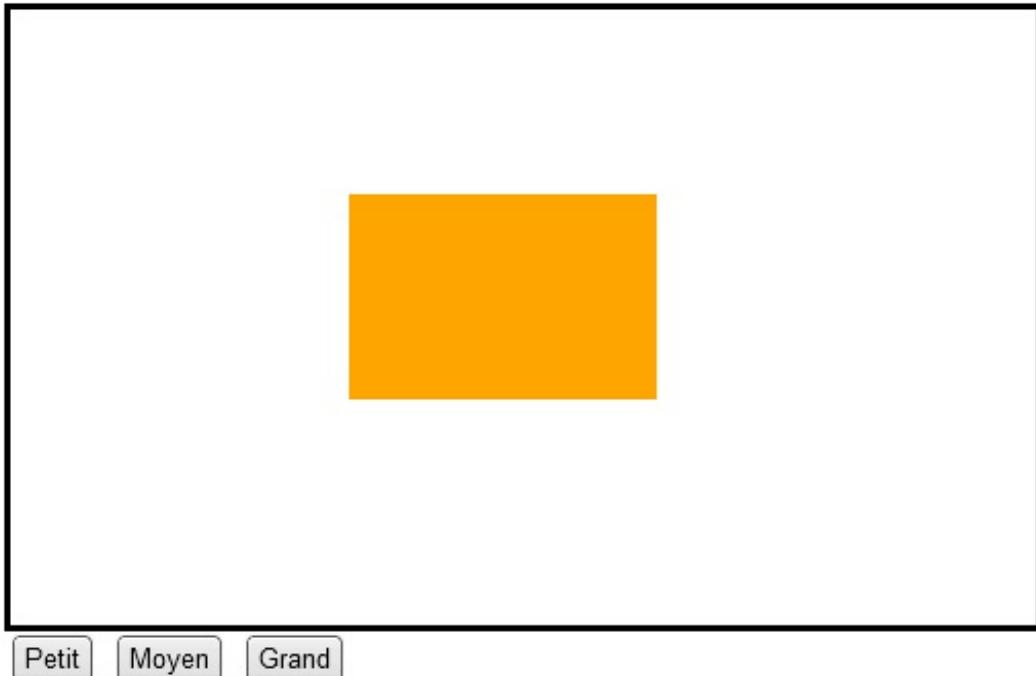
Remarquez qu'il faut utiliser la méthode **draw** du calque pour actualiser la modification en redessinant le rectangle.

Le point point de référence pour le redimensionnement est le même que pour la position. Pour obtenir un effet différent, par exemple centrer le redimensionnement sur le centre du rectangle il faut jouer sur la propriété **offset**. Voici le même exemple avec un offset qui permet un centrage de l'effet :

Code : JavaScript

```
var rectangle = new Kinetic.Rect({  
    x: 240,  
    y: 140,  
    width: 300,  
    height: 200,  
    fill: "orange",  
    offset: [150, 100]  
});
```

Tester !



Petit Moyen Grand

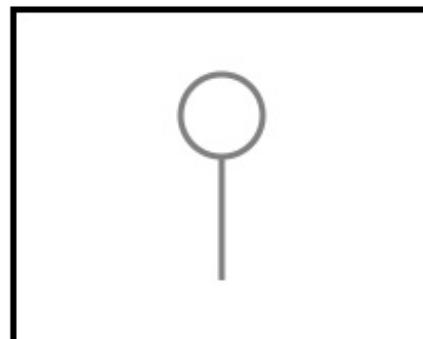
## Groupements Principe de base

Voici un objet sommaire composé d'un cercle et d'un trait groupés :

Code : JavaScript

```
var groupe = new Kinetic.Group({  
    x: 100,  
    y: 50  
});  
  
(function() {  
    var cercle = new Kinetic.Ellipse({  
        radius: 20,  
        stroke: "grey",  
        strokeWidth: 3  
    });  
    var ligne = new Kinetic.Line({  
        points: [0, 20, 0, 80],  
        stroke: "grey",  
        strokeWidth: 3  
    });  
    groupe.add(cercle);  
    groupe.add(ligne);  
})();  
  
calque.add(groupe);
```

Tester !



## Exemple de création d'objets

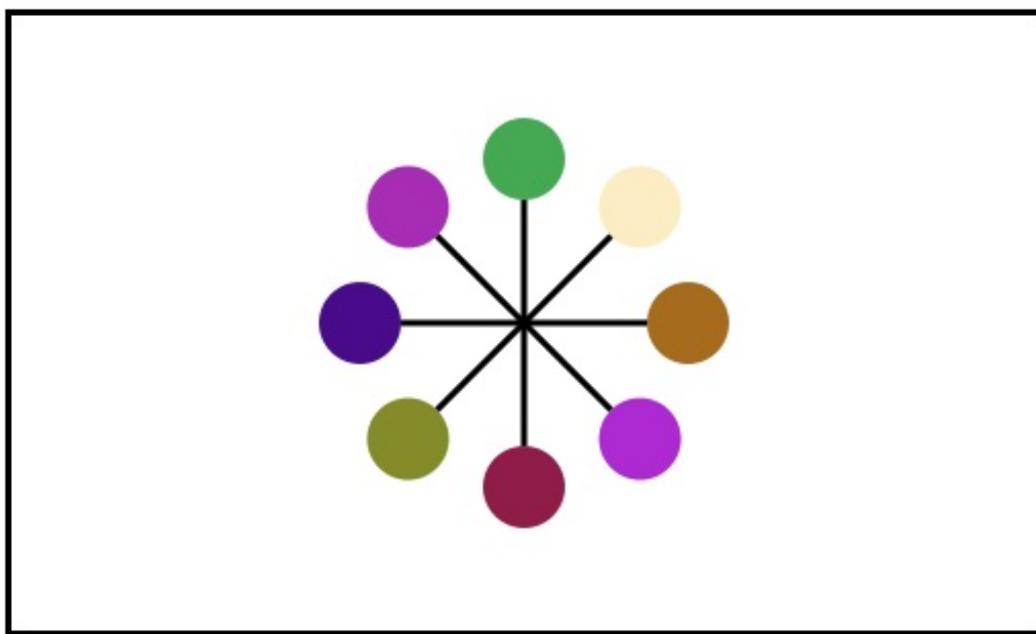
Partons de la base du groupement précédent pour créer une figure composée de plusieurs éléments groupés :

**Code : JavaScript**

```
for(var i = 0; i < 360; i += 45) {
    var groupe = new Kinetic.Group({
        x: 250,
        y: 150,
        offset: [0, 80],
        rotationDeg: i
    });

    (function() {
        var couleur = "rgb(" +
            (Math.floor(Math.random() * 256)) + ", "
            + (Math.floor(Math.random() * 256)) + ", "
            + (Math.floor(Math.random() * 256)) + ")";
        var cercle = new Kinetic.Ellipse({
            radius: 20,
            fill: couleur
        });
        var ligne = new Kinetic.Line({
            points: [0, 20, 0, 80],
            strokeWidth: 3
        });
        groupe.add(cercle);
        groupe.add(ligne);
    })();
    calque.add(groupe);
}
```

Tester !



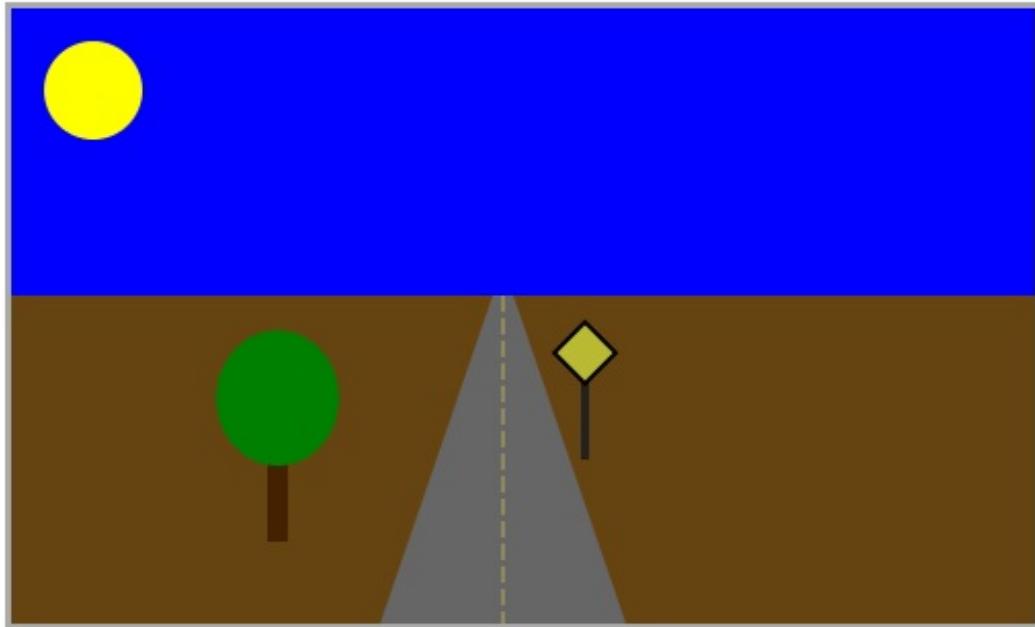
J'ai ajouté un offset à l'objet pour avoir comme point de référence la pointe de la tige. Ensuite je crée 8 objets en changeant la rotation de 45 degrés pour obtenir ce dessin symétrique. Pour l'esthétique je génère aléatoirement la couleur du disque. On voit ici l'intérêt du groupement qui permet d'appliquer un effet à plusieurs formes avec un code concis.

 TP

Vous avez désormais tout en main pour faire quelques TP.

### Un paysage

Voici le résultat que l'on veut obtenir :



[Tester !](#)

Tout cela réalisé avec les classes que l'on a vues jusqu'ici.

#### Secret ([cliquez pour afficher](#))

##### Code : HTML

```
<style>
canvas {
    border-style:solid;
    border-color:#aaa;
    box-shadow: 5px 5px 3px #999;
}
</style>
<script>
window.onload = function() {
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

    var calque = new Kinetic.Layer();

    // Ciel
    calque.add(
        new Kinetic.Rect({
            width: 500,
            height: 140,
            fill: "blue"
        })
    );

    // Sol
    calque.add(
        new Kinetic.Rect({
            width: 500,
            height: 160,
            fill: "#ccc"
        })
    );
}
```

```
new Kinetic.Rect({
    y: 140,
    width: 500,
    height: 160,
    fill: "#641"
})
);

// Soleil
calque.add(
    new Kinetic.Ellipse({
        x: 40,
        y: 40,
        radius: 24,
        fill: "yellow"
    })
);

// Route
calque.add(
    new Kinetic.Polygon({
        points: [235, 140, 245, 140, 300, 300, 180, 300],
        fill: "#666"
    })
);
calque.add(
    new Kinetic.Line({
        points: [240, 140, 240, 300],
        stroke: "#ba5",
        strokeWidth: 1,
        dashArray: [8, 3]
    })
);

// Panneau
calque.add(
    new Kinetic.Line({
        points: [280, 180, 280, 220],
        stroke: "#222",
        strokeWidth: 4
    })
);
calque.add(
    new Kinetic.RegularPolygon({
        x: 280,
        y: 168,
        sides: 4,
        radius: 15,
        stroke: "black",
        fill: "#bb3",
    })
);

// Arbre
calque.add(
    new Kinetic.Line({
        points: [130, 220, 130, 260],
        stroke: "#420",
        strokeWidth: 10
    })
);
calque.add(
    new Kinetic.Ellipse({
        x: 130,
        y: 190,
        radius: [30, 33],
        fill: "green"
    })
);
```

```
    scène.add(calque);
};

</script>
```

J'ai ajouté un peu de style au **canvas** pour l'esthétique.

Bien sûr ce n'est qu'un exercice de style parce qu'il n'est pas vraiment nécessaire d'utiliser des objets pour des éléments graphiques qui doivent rester immobiles. Il est alors évidemment bien plus simple d'utiliser une image :

#### Code : HTML

```
<style>
  canvas {
    border-style:solid;
    border-color:#aaa;
    box-shadow: 5px 5px 3px #999;
  }
</style>
<script>
window.onload = function() {
  var scène = new Kinetic.Stage({
    container: "kinetic",
    width: 500,
    height: 300
  });

  var calque = new Kinetic.Layer();

  var img = new Image();
  img.onload = function() {
    var image = new Kinetic.Image({
      image: img
    });

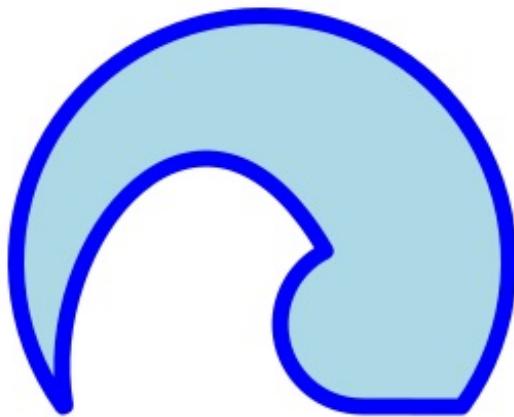
    calque.add(image);

    scène.add(calque);
  };
  img.src = "images/img01_47.png";
};
</script>
```

Tester !

## Une vague

L'objectif est de créer cette figuration de vague :



Tester !

**Secret** (cliquez pour afficher)

Code : JavaScript

```
window.onload = function() {
  var scène = new Kinetic.Stage({
    container: "kinetic",
    width: 500,
    height: 300
  });

  var calque = new Kinetic.Layer();

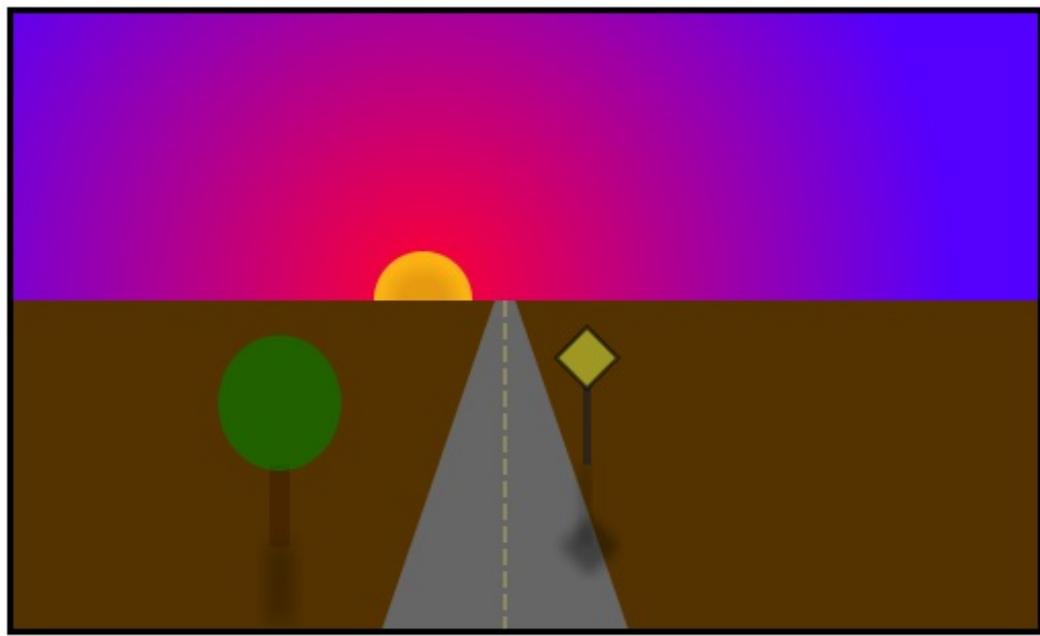
  var vague = new Kinetic.Shape({
    drawFunc: function(context) {
      context.beginPath();
      context.arc(160, 140, 120, .8 * Math.PI, 2.2 * Math.PI);
      context.arcTo(40, 210, 140, 160, 40);
      context.bezierCurveTo(130, 30, 50, 130, 64, 210);
      context.closePath();
      this.fill(context);
      this.stroke(context);
    },
    lineJoin: "round",
    fill: "lightblue",
    stroke: "blue",
    strokeWidth: 8,
  });
  calque.add(vague);

  scène.add(calque);
}
```

Il y a bien entendu d'autres façons d'obtenir ce résultat 😊

## Encore le paysage

Je vous propose de reprendre le paysage du premier TP en créant une ambiance nocturne :



Tester !

Bon d'accord ce n'est pas du grand art mais suffisant pour s'amuser avec les dégradés et les ombres 😊

**Secret** ([cliquez pour afficher](#))

**Code : JavaScript**

```
window.onload = function() {
  var scène = new Kinetic.Stage({
    container: "kinetic",
    width: 500,
    height: 300
  });

  var calque = new Kinetic.Layer();

  // Ciel
  calque.add(
    new Kinetic.Rect({
      width: 500,
      height: 140,
      fill: {
        start: {
          x: 200,
          y: 134,
          radius: 0
        },
        end: {
          x: 200,
          y: 134,
          radius: 260
        },
        colorStops: [0, "#f03", 1, "#50f"]
      }
    })
  );

  // Soleil
  calque.add(
    new Kinetic.Ellipse({
      x: 200,
      y: 140,
      radius: 24,
      opacity: .7,
      fill: "#ffcc00"
    })
  );
}
```

```
        fill: {
          start: {
            x: 0,
            y: 0,
            radius: 10
          },
          end: {
            x: 0,
            y: 0,
            radius: 20
          },
          colorStops: [0, "#dd0", 1, "#ff0"]
        },
      })
);

// Sol
calque.add(
  new Kinetic.Rect({
    y: 140,
    width: 500,
    height: 160,
    fill: "#530"
  })
);
// Route
calque.add(
  new Kinetic.Polygon({
    points: [235, 140, 245, 140, 300, 300, 180, 300],
    fill: "#666"
  })
);
calque.add(
  new Kinetic.Line({
    points: [240, 140, 240, 300],
    stroke: "#ba5",
    strokeWidth: 1,
    dashArray: [8, 3]
  })
);

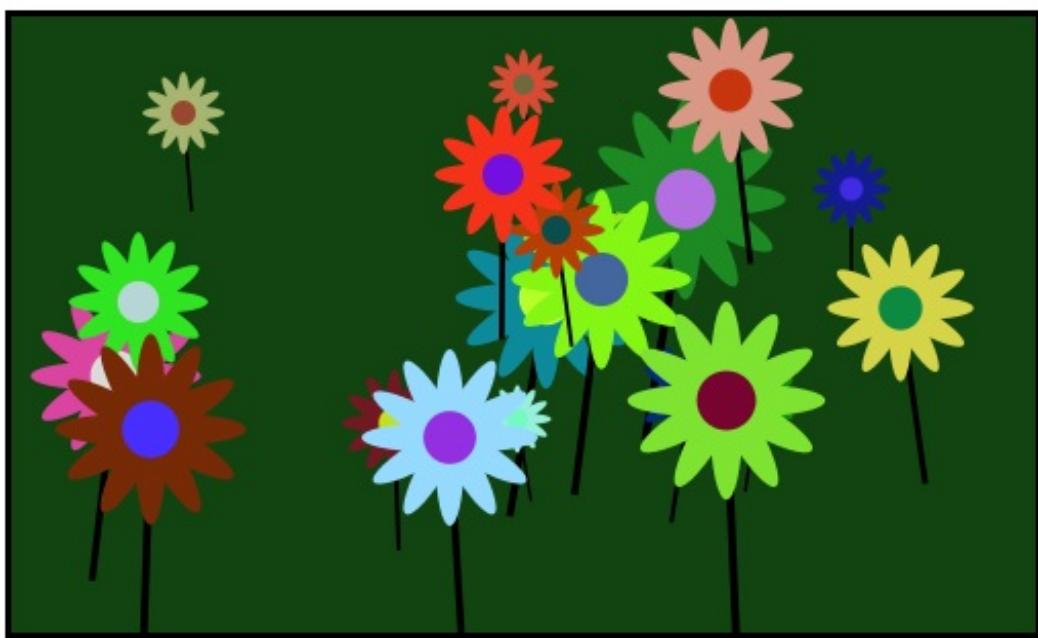
// Panneau
calque.add(
  new Kinetic.Line({
    points: [280, 180, 280, 220],
    stroke: "#222",
    strokeWidth: 4,
    opacity: .6,
    shadow: {
      color: "black",
      blur: 6,
      offset: [-2, 38],
      opacity: .6
    }
  })
);
calque.add(
  new Kinetic.RegularPolygon({
    x: 280,
    y: 168,
    sides: 4,
    radius: 15,
    stroke: "black",
    fill: "#bb3",
    alpha: .6,
    shadow: {
      color: "black",
      blur: 6,
      offset: [0, 90],
      opacity: .6
    }
  })
);
```

```
        }
    });
};

// Arbre
calque.add(
    new Kinetic.Line({
        points: [130, 220, 130, 260],
        stroke: "#420",
        strokeWidth: 10,
        opacity: .6,
        shadow: {
            color: "black",
            blur: 16,
            offset: [0, 38],
            opacity: .7
        },
    })
);
calque.add(
    new Kinetic.Ellipse({
        x: 130,
        y: 190,
        radius: [30, 33],
        fill: "green",
        opacity: .6
    })
);
scène.add(calque);
};
```

## Fleurs

Le but est d'obtenir une image de ce genre :



Tester !

Avec évidemment une génération aléatoire des positions, dimensions et couleurs pour obtenir un effet réaliste et varié.  
Rafraîchissez l'écran pour obtenir de nouvelles compositions.

Ne sautez pas sur ma solution tout de suite, le but est de vous entraîner à utiliser la librairie 😊.

**Secret (cliquez pour afficher)****Code : JavaScript**

```
function getColor() {
    return "rgb(" + getInteger(256) + "," + getInteger(256) + "," + 
getInteger(256) + ")";
}
function getInteger(valmax) {
    return Math.floor(Math.random() * (valmax + 1));
}
window.onload = function() {
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

    var calque = new Kinetic.Layer();

    // Fond vert
    var fond = new Kinetic.Rect({
        width: scène.getWidth(),
        height: scène.getHeight(),
        fill: "#141",
    });
    calque.add(fond);

    // Génération des fleurs
    for(var i = 0; i < 20; ++i) {
        var groupe = new Kinetic.Group({
            x: getInteger(400) + 50,
            y: getInteger(180) + 30,
        });
        (function() {
            // Tige
            var tige = new Kinetic.Line({
                points: [0, 0, 20 - getInteger(40), 120],
                stroke: "black",
                strokeWidth: 4
            });
            groupe.add(tige);
            // Pétales
            var couleur = getColor();
            for(var j = 0; j < 360; j += 30) {
                var pétalement = new Kinetic.Ellipse({
                    radius: [36, 8],
                    fill: couleur,
                    rotationDeg: j,
                    offset: [14, 0]
                });
                groupe.add(pétalement);
            }
            // Pistil
            var pistil = new Kinetic.Ellipse({
                radius: 15,
                fill: getColor(),
            });
            groupe.add(pistil);
            groupe.setScale(Math.random() * .7 + .3);
        })();
        calque.add(groupe);
    }

    scène.add(calque);
};
```

Je ne commente pas ce code, je vous laisse l'analyser et comparer avec ce que vous avez produit.

## Partie 2 : Animer

Maintenant que nous savons dessiner, introduisons un peu d'interactivité et d'animation.

### Interactions

Nous allons voir à présent comment agir, avec la souris, sur les formes que nous avons dessinées.

#### Événements

La classe **Node** de **KineticJS** est équipée pour gérer les principaux événements de la souris :

Événement	Description
<b>click</b>	Se produit lorsqu'on clique sur l'élément associé à l'événement.
<b>dblclick</b>	Se produit lorsqu'on fait un double clic sur l'élément associé à l'événement.
<b>mousedown</b>	Se produit lorsqu'on appuie sur le bouton sur l'élément associé à l'événement.
<b>mouseup</b>	Se produit lorsqu'on relâche le bouton sur l'élément associé à l'événement.
<b>mouseover</b>	Se produit lorsqu'on positionne le curseur sur l'élément associé à l'événement.
<b>mouseout</b>	Se produit lorsque le curseur quitte l'élément associé à l'événement.
<b>mousemove</b>	Se produit lorsque le curseur se déplace sur l'élément associé à l'événement.
<b>mouseenter</b>	Se produit lorsque le curseur se place sur l'élément associé à l'événement.
<b>mouseleave</b>	Se produit lorsque le curseur quitte l'élément associé à l'événement.



Mais c'est quoi la différence entre **mouseout** et **mouseleave** ?

Voilà une bonne question, vous avez une excellente réponse [ici](#). Remarquez que c'est la même chose pour les événements **mouseover** et **mouseenter**. Tant que vous n'avez pas d'éléments imbriqués, le fonctionnement est identique, mais dès que vous en avez alors les événements **mouseenter** et **mouseleave** sont vraiment très pratiques 😊.

Elle permet également de gérer les événements des terminaux mobiles comme les smartphones et les tablettes :

Événement	Description
<b>touchstart</b>	Se produit lorsque le doigt est placé sur l'élément associé à l'événement.
<b>touchmove</b>	Se produit lorsque le doigt est déplacé sur l'élément associé à l'événement.
<b>touchend</b>	Se produit lorsque le doigt est retiré de l'élément associé à l'événement.
<b>tap</b>	Se produit lors d'un toucher rapide sur l'élément associé à l'événement.
<b>dbltap</b>	Se produit lors d'un double toucher rapide sur l'élément associé à l'événement.

On peut facilement gérer ces événements sur les formes dessinées.

### Mise en place

La mise en place de l'écoute d'un événement se fait simplement. Il suffit d'utiliser la méthode **on** et de renseigner les deux paramètres : le type d'événement à écouter et une fonction à exécuter lorsque cet événement survient. Voici un exemple simple où on détecte le clic de la souris sur un disque coloré :

#### Code : JavaScript

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

```
function setText(text, color) {
    Texte.setText(text);
    Texte.setTextFill(color);
    calque.draw();
}

var scène = new Kinetic.Stage({
    container: "kinetic",
    width: 500,
    height: 300
});

var calque = new Kinetic.Layer();

var Texte = new Kinetic.Text({
    x: 10,
    y: 10,
    text: "Cliquez sur le disque rouge",
    fontSize: 18,
    fontFamily: "verdana",
    textFill: "grey"
});

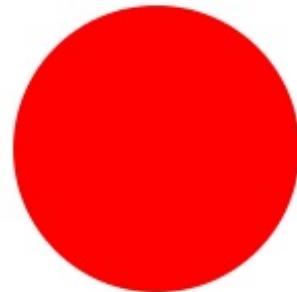
var disque = new Kinetic.Ellipse({
    x: 250,
    y: 150,
    radius: 70,
    fill: "red",
});

disque.on("click", function() {
    setText("On a cliqué sur le disque rouge !", "red");
    setTimeout(function() {
        setText("Cliquez sur le disque rouge", "grey");
    }, 2000);
});

calque.add(disque);
calque.add(Texte);
scène.add(calque);
};
```

Tester !

Cliquez sur le disque rouge



## Empilement d'événements

On est pas limité à un seul événement sur une forme. Pour en gérer plusieurs c'est tout simple, il suffit de les inscrire en les séparant par un espace. Voici l'exemple d'un disque qui change de couleur sur **mousedown** et **mouseup**:

Code : JavaScript

```
function getColor() {
    return "rgb(" + getInteger(256) + ", " + getInteger(256) + ", " +
getInteger(256) + ")";
}
function getInteger(valmax) {
    return Math.floor(Math.random() * (valmax + 1));
}
window.onload = function() {
    function setText(text, color) {
        Texte.setText(text);
        Texte.setTextFill(color);
        calque.draw();
    }

    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

    var calque = new Kinetic.Layer();

    var Texte = new Kinetic.Text({
        x: 10,
        y: 10,
        text: "Cliquez sur le disque, il change de couleur sur mousedown
et mouseup",
        fontSize: 14,
        fontFamily: "verdana",
        textFill: "grey",
        lineHeight: 2,
        width: 480,
        align: "center"
    });

    var disque = new Kinetic.Ellipse({
        x: 250,
        y: 170,
        radius: 70,
        fill: getColor(),
    });

    disque.on("mousedown mouseup", function() {
        disque.setFill(getColor());
        calque.draw();
    });

    calque.add(disque);
    calque.add(Texte);
    scène.add(calque);
};
```

Tester !



Cette possibilité est utile pour créer des applications qui doivent fonctionner sur écran normal et terminal mobile en associant par exemple **mousdown** et **touchstart**.

## Supprimer un événement

La suppression d'un événement est aussi aisée que sa mise en place, il suffit d'utiliser la méthode **off** en indiquant quel événement on veut supprimer. Voici un exemple avec un disque que l'on peut cliquer 3 fois. Au bout de ces trois clics, l'événement est supprimé :

### Code : JavaScript

```
window.onload = function() {
    function setText() {
        var text = "Cliquez sur le disque rouge\nVous avez encore " + i +
        " possibilité";
        if(i > 1) text += "s";
        text += "...";
        Texte.setText(text);
        calque.draw();
    }

    var i = 3;

    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

    var calque = new Kinetic.Layer();

    var Texte = new Kinetic.Text({
        x: 10,
        y: 10,
        text: "Cliquez sur le disque rouge\nVous avez encore 3
possibilités...",
        fontSize: 14,
        fontFamily: "verdana",
        textFill: "grey",
        lineHeight: 2
    });
}
```

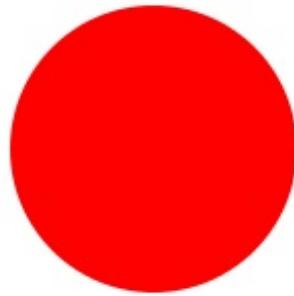
```
var disque = new Kinetic.Ellipse({
    x: 250,
    y: 150,
    radius: 70,
    fill: "red",
});

disque.on("click", function() {
    if(--i) setText();
    else {
        disque.off("click");
        Texte.setText("Vous ne pouvez plus cliquer !");
        calque.draw();
    }
});

calque.add(disque);
calque.add(Texte);
scène.add(calque);
};
```

Tester !

Cliquez sur le disque rouge  
Vous avez encore 3 possibilités...



## Suspendre un événement

La suppression totale d'un événement n'est pas toujours judicieuse. Parfois on a juste besoin de suspendre l'écoute pour la réactiver par la suite. La propriété **listening** nous permet de réaliser cela, il suffit de la renseigner avec la valeur **true** pour activer l'événement et la valeur **false** pour le désactiver. Voici un exemple avec l'événement **mouseover** qui permet de changer la couleur d'un disque lorsque le curseur de la souris passe dessus. Un clic sur le triangle situé à côté permet de désactiver et activer cet événement :

Code : JavaScript

```
function getColor() {
    return "rgb(" + getInteger(256) + ", " + getInteger(256) + ", " +
    getInteger(256) + ")";
}
function getInteger(valmax) {
    return Math.floor(Math.random() * (valmax + 1));
}
window.onload = function() {
```

```
function setText(text) {
    Texte.setText(text);
    calque.draw();
}

var scène = new Kinetic.Stage({
    container: "kinetic",
    width: 500,
    height: 300
});

var état = true;

var calque = new Kinetic.Layer();

var Texte = new Kinetic.Text({
    x: 10,
    y: 10,
    text: "Passez sur le disque pour changer sa couleur et cliquez sur le triangle pour désactiver l'événement",
    fontSize: 14,
    fontFamily: "verdana",
    textFill: "grey",
    lineHeight: 2,
    width : 480
});

var disque = new Kinetic.Ellipse({
    x: 150,
    y: 180,
    radius: 70,
    fill: getColor(),
});

var triangle = new Kinetic.RegularPolygon({
    x: 350,
    y: 200,
    sides: 3,
    radius: 90,
    fill: "red",
});

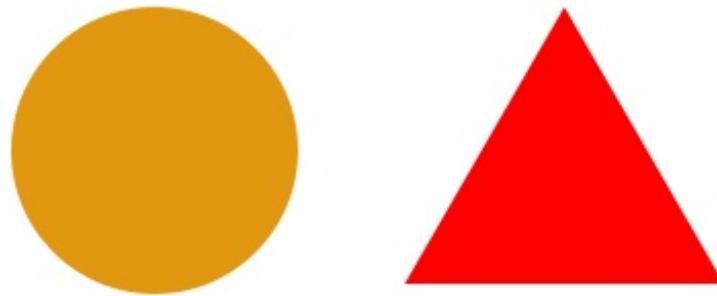
disque.on("mouseover", function() {
    disque.setFill(getColor());
    calque.draw();
});

triangle.on("click", function() {
    état = !état;
    disque.setListening(état);
    if(état) setText("Passez sur le disque pour changer sa couleur et cliquez sur le triangle pour désactiver l'événement");
    else setText("L'événement est désactivé");
});

calque.add(disque);
calque.add(triangle);
calque.add(Texte);
scène.add(calque);
};
```

Tester !

Passez sur le disque pour changer sa couleur et cliquez sur le triangle pour désactiver l'événement



## Position du curseur

### *Position absolue*

Voici un premier exemple :

#### Code : JavaScript

```
window.onload = function() {
    function setText() {
        var mousePos = scène.getMousePosition();
        var text = "Le curseur de la souris est en X = " + mousePos.x + " Y
= " + mousePos.y;
        Texte.setText(text);
        calque.draw();
    }

    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

    var calque = new Kinetic.Layer();

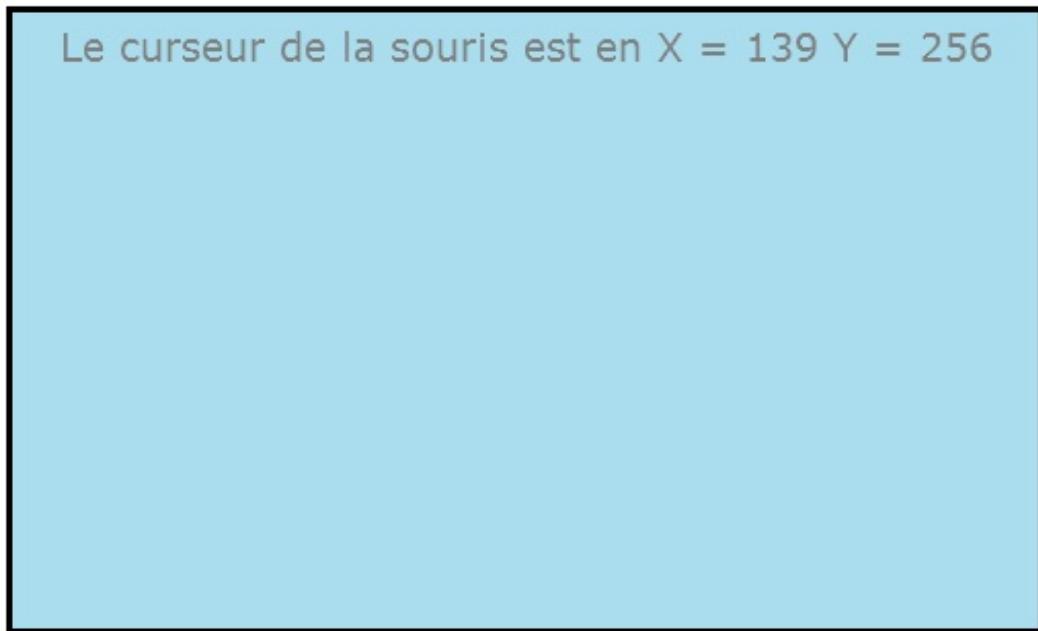
    var Rectangle = new Kinetic.Rect({
        width: scène.getWidth(),
        height: scène.getHeight(),
        fill: "#ade",
    });

    var Texte = new Kinetic.Text({
        x: 10,
        y: 10,
        text: "Bougez la souris",
        fontSize: 14,
        fontFamily: "verdana",
        textFill: "grey",
        lineHeight: 2,
        width: 480,
        align: "center"
    });
}
```

```
scène.on("mousemove", function() {
    setText();
});

calque.add(Rectangle);
calque.add(Texte);
scène.add(calque);
};
```

Tester !



### Position relative

Remarquez que dans l'exemple précédent j'ai créé un rectangle de la dimension de la scène pour faire fonctionner l'événement, en effet par défaut la détection se fait sur les formes, il faut donc en créer une. Pour déterminer la position de la souris sur une forme de dimension plus réduite que la scène, il faut ajuster avec l'offset de positionnement :

#### Code : JavaScript

```
window.onload = function() {
    function setText() {
        var mousePos = scène.getMousePosition();
        var x = mousePos.x - Rectangle.getX();
        var y = mousePos.y - Rectangle.getY();
        var text = "Le curseur de la souris est en X = " + x + " Y = " + y;
        Texte.setText(text);
        calque.draw();
    }

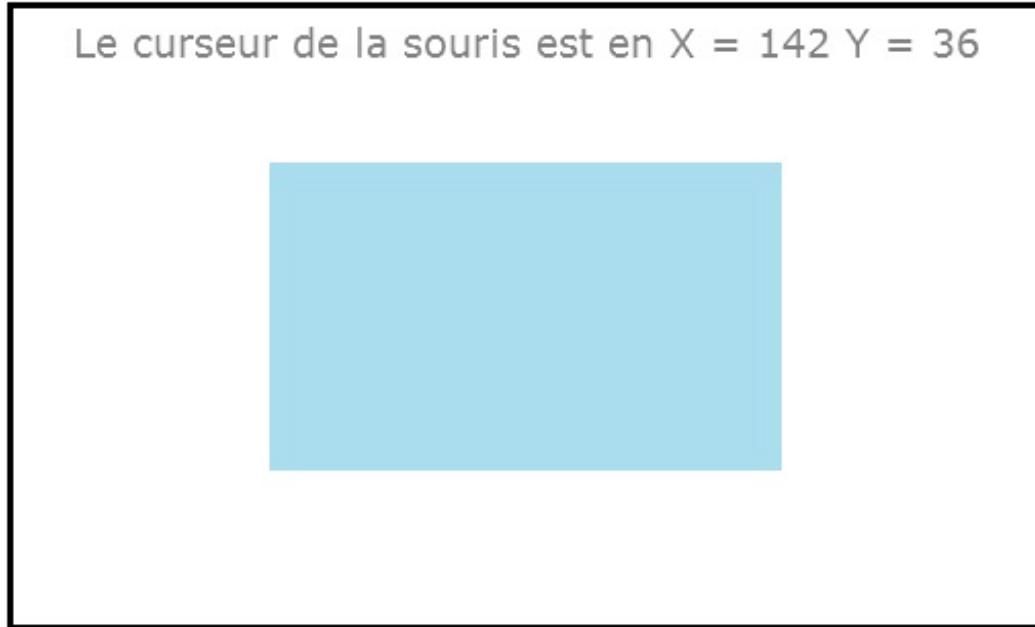
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

    var calque = new Kinetic.Layer();

    var Rectangle = new Kinetic.Rect({
        x: scène.getWidth() / 4,
        y: scène.getHeight() / 4,
```

```
width: scène.getWidth() / 2,  
height: scène.getHeight() / 2,  
fill: "#ade",  
});  
  
var Texte = new Kinetic.Text({  
x: 10,  
y: 10,  
text: "Bougez la souris",  
fontSize: 14,  
fontFamily: "verdana",  
textFill: "grey",  
lineHeight: 2,  
width: 480,  
align: "center"  
});  
  
Rectangle.on("mousemove", function() {  
    setText();  
});  
  
Rectangle.on("mouseout", function() {  
    Texte.setText("Curseur en dehors du rectangle");  
    calque.draw();  
});  
  
calque.add(Rectangle);  
calque.add(Texte);  
scène.add(calque);  
});
```

Tester !



## Déetecter des pixels

### *DéTECTER SUR UNE IMAGE*

Voici un exemple avec l'image d'une voiture et la mise en œuvre d'un événement **click** comme nous l'avons précédemment fait :

**Code : JavaScript**

```
window.onload = function() {
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

    var calque = new Kinetic.Layer();

    var img = new Image();
    img.onload = function() {
        var image = new Kinetic.Image({
            x: 100,
            y: 100,
            image: img,
            width: 120,
            height: 120,
        });

        image.on("click", function() {
            Texte.setText("Image cliquée");
            calque.draw();
            setTimeout(function(){
                Texte.setText("Cliquez sur l'image");
                calque.draw();
            }, 2000);
        });
    };

    calque.add(image);
    calque.draw();
};

img.src = "images/voiture.png";

var Texte = new Kinetic.Text({
    y: 10,
    text: "Cliquez sur l'image",
    fontSize: 14,
    fontFamily: "verdana",
    textFill: "grey",
    lineHeight: 2,
    width: 480,
    align: "center"
});

calque.add(Texte);
scène.add(calque);
};
```

Tester !



Vous vous rendez compte que le **clic** fonctionne bien, mais on détecte aussi les zones transparentes !

### *Amélioration de la détection*

Pour résoudre le problème vu précédemment il faut modifier un peu le code. Le voici corrigé :

#### Code : JavaScript

```
window.onload = function() {
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

    var calque = new Kinetic.Layer();

    var img = new Image();
    img.onload = function() {
        var image = new Kinetic.Image({
            x: 100,
            y: 100,
            image: img,
            width: 120,
            height: 120
        });
        image.createImageBuffer();

        image.on("click", function() {
            Texte.setText("Image cliquée");
            calque.draw();
            setTimeout(function() {
                Texte.setText("Cliquez sur l'image");
                calque.draw();
            }, 2000);
        });

        calque.add(image);
        calque.draw();
        image.saveImageData();
    };
    img.src = "images/voiture.png";

    var Texte = new Kinetic.Text({
```

```
        y: 10,
        text: "Cliquez sur l'image",
        fontSize: 14,
        fontFamily: "verdana",
        textFill: "grey",
        lineHeight: 2,
        width: 480,
        align: "center"
    });

    calque.add(Texte);
    scène.add(calque);
};
```

Tester !

On sauvegarde les pixels avec la méthode **createImageBuffer** pour gagner en précision et ne plus déclencher l'événement sur des zones transparentes.

## Glisser-déposer

Voyons maintenant un outil très pratique : le glisser-déposer, autrement appelé drag-and-drop par nos amis anglophones. C'est une procédure qui permet de manipuler visuellement des éléments à l'écran et de gérer ce déplacement avec des événements appropriés.

## Activation simple

Il y a la propriété **draggable** pour activer le glisser-déposer. Par défaut elle a la valeur **false**, mais on peut définir la valeur **true** pour la rendre opérationnelle. Voici un exemple avec une génération aléatoire de 10 étoiles toutes "draggables" et un peu transparentes pour faciliter la visualisation :

### Code : JavaScript

```
window.onload = function() {
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

    var calque = new Kinetic.Layer();

    for(var i = 0; i < 10; ++i) {
        var étoile = new Kinetic.Star({
            x: Math.random() * scène.getWidth(),
            y: Math.random() * scène.getHeight(),
            numPoints: 5,
            innerRadius: 40,
            outerRadius: 80,
            fill: '#823',
            stroke: '#612',
            opacity: 0.7,
            strokeWidth: 10,
            draggable: true,
            scale: (Math.random() * .8) + .2,
            rotationDeg: Math.random() * 180
        });
        calque.add(étoile);
    }

    scène.add(calque);
};
```

Tester !



Vous constatez que vous pouvez déplacer toutes les étoiles individuellement avec la souris.

## Un exemple avec des lignes

Voici un autre exemple avec des lignes :

Code : JavaScript

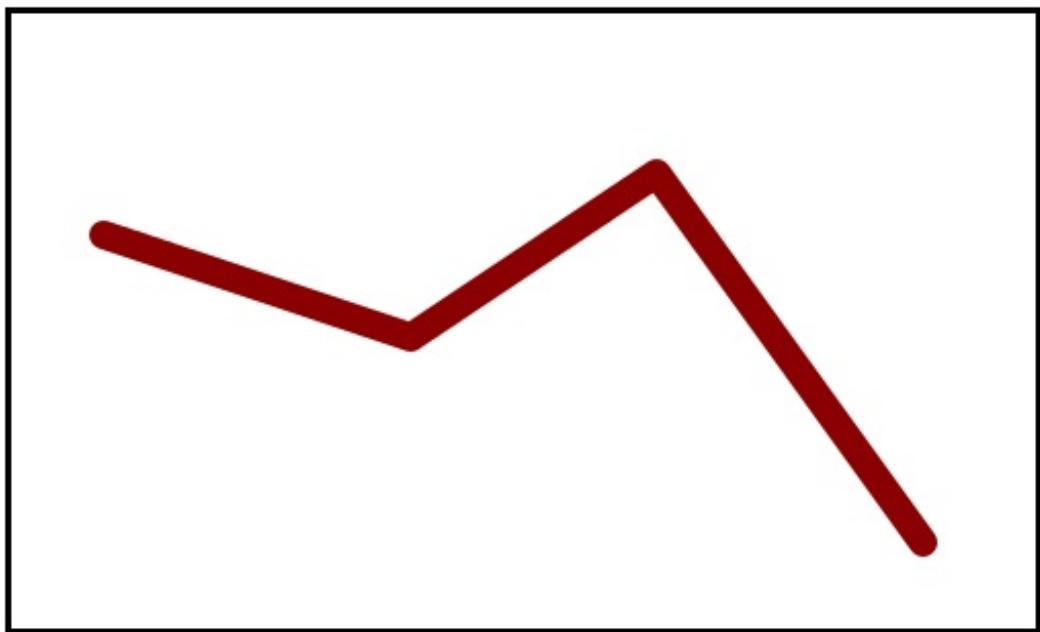
```
window.onload = function() {
  var scène = new Kinetic.Stage({
    container: "kinetic",
    width: 500,
    height: 300
  });

  var calque = new Kinetic.Layer();

  var ligne = new Kinetic.Line({
    points: [50, 50, 200, 100, 320, 20, 450, 200],
    stroke: "darkred",
    strokeWidth: 14,
    lineCap: "round",
    lineJoin: "round",
    draggable: true
  });

  calque.add(ligne);
  scène.add(calque);
};
```

Tester !



## Activation pour un groupe

Comment déplacer d'un bloc toutes les étoiles créées par le code précédent ? Il suffit de les grouper, comme nous l'avons vu dans un chapitre précédent et de déclarer ce groupe "draggable". voici le code modifié en conséquence :

### Code : JavaScript

```
window.onload = function() {
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

    var calque = new Kinetic.Layer();

    var groupe = new Kinetic.Group({
        draggable: true
    });

    for(var i = 0; i < 10; ++i) {
        var étoile = new Kinetic.Star({
            x: Math.random() * scène.getWidth(),
            y: Math.random() * scène.getHeight(),
            numPoints: 5,
            innerRadius: 40,
            outerRadius: 80,
            fill: '#823',
            stroke: '#612',
            opacity: 0.7,
            strokeWidth: 10,
            scale: (Math.random() * .8) + .2,
            rotationDeg: Math.random() * 180
        });
        groupe.add(étoile);
    }

    calque.add(groupe);
    scène.add(calque);
};
```

[Tester !](#)

Cette fois c'est bien l'ensemble des étoiles qu'on peut déplacer en bloc.

## Événements du glisser-déposer

Il y a 3 événements pour le glisser-déposer :

Événement	Description
<b>dragstart</b>	Se produit lorsqu'on commence à déplacer l'élément associé à l'événement.
<b>dragmove</b>	Se produit lorsqu'on déplace l'élément associé à l'événement.
<b>dragend</b>	Se produit lorsqu'on dépose l'élément associé à l'événement.

Voyons maintenant un exemple : on trace une ligne avec un disque à son extrémité. On veut pouvoir déplacer le disque avec la souris et que la ligne s'adapte automatiquement pour que son extrémité coïncide avec le disque :

**Code : JavaScript**

```
window.onload = function() {
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

    var calque = new Kinetic.Layer();

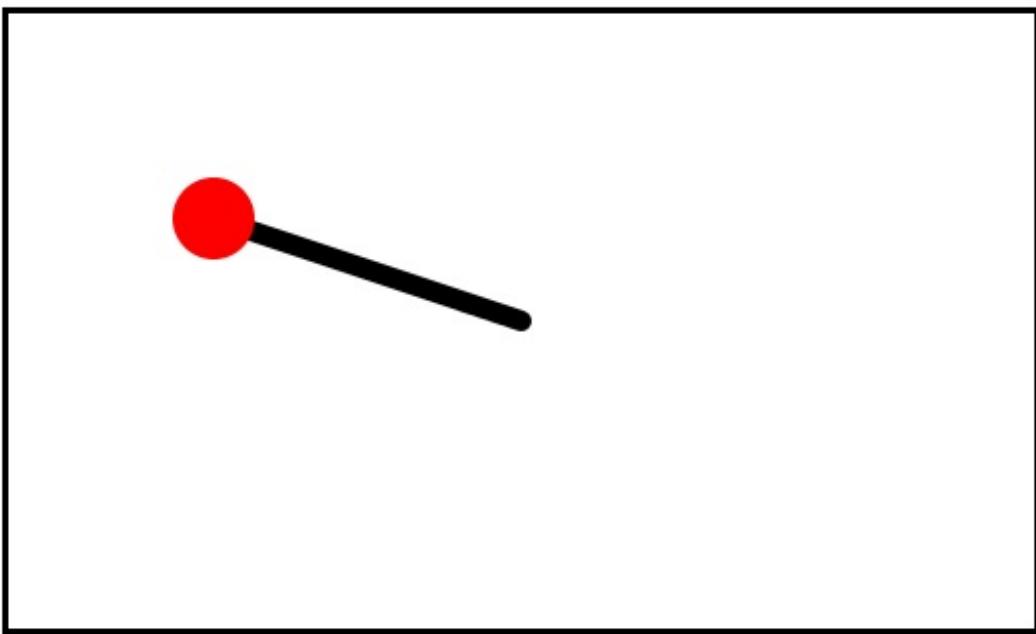
    var ligne = new Kinetic.Line({
        points: [250, 150, 100, 100],
        strokeWidth: 10,
        lineCap: "round",
        lineJoin: "round",
        draggable: true
    });

    var disque = new Kinetic.Ellipse({
        x: 100,
        y: 100,
        radius: 20,
        fill: "red",
        draggable: true
    });

    disque.on("dragmove", function() {
        var pos = this.getPosition();
        ligne.setPoints(250, 150, pos.x, pos.y);
        calque.draw();
    });

    calque.add(ligne);
    calque.add(disque);
    scène.add(calque);
};
```

[Tester !](#)



J'ai utilisé encore l'événement **dragmove**. On récupère la position du disque avec la méthode **getPosition** qui renvoie les coordonnées qu'il suffit ensuite d'appliquer à la ligne en définissant une nouvelle série de points de traçage.

## Contraintes de déplacement

On veut parfois interdire à la forme déplacée de dépasser une certaine limite, ou alors de rester dans un espace particulier, en un mot contraindre le déplacement selon nos besoins. **KineticJS** dans ses précédentes versions ne prévoyait que deux possibilités : une contrainte linéaire horizontale ou verticale, ou une limite de déplacement. La version 4.02 a amélioré les choses, désormais on dispose d'une fonction pour écrire le code que l'on veut 😊. La propriété à utiliser est **dragBoundFunc**. Elle prend comme valeur la fonction à utiliser pour effectuer la contrainte.

### Contrainte linéaire

On peut facilement contraindre le déplacement sur un trajet linéaire, par exemple horizontal et vertical. Voici un exemple :

#### Code : JavaScript

```
window.onload = function() {
  var scène = new Kinetic.Stage({
    container: "kinetic",
    width: 500,
    height: 300
  });

  var calque = new Kinetic.Layer();

  var trame_flèche =
  [{x:50,y:50},{x:90,y:90},{x:60,y:90},{x:60,y:190},{x:90,y:190},
  {x:50,y:230},{x:10,y:190},{x:40,y:190},{x:40,y:90},{x:10,y:90}];

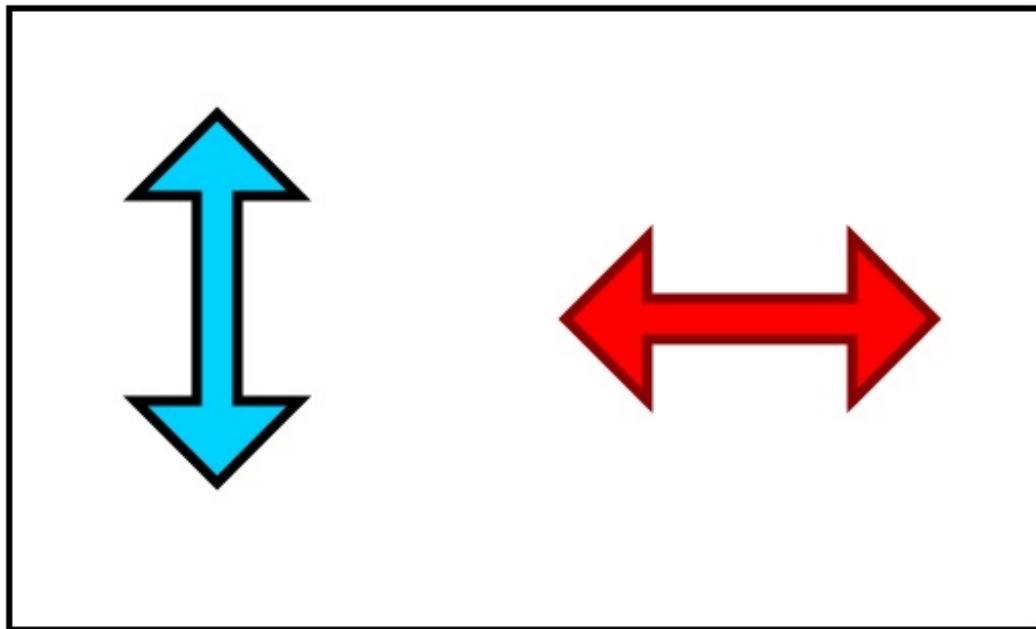
  var flèchel = new Kinetic.Polygon({
    points: trame_flèche,
    fill: "#00D2FF",
    stroke: "black",
    strokeWidth: 5,
    offset: [-50, 0],
    draggable: true,
    dragBoundFunc: function(pos) {
      return {
        x: this.getAbsolutePosition().x,
```

```
y: pos.y
    }
});
};

var flèche2 = new Kinetic.Polygon({
    points: trame_flèche,
    fill: "red",
    stroke: "darkred",
    strokeWidth: 5,
    offset: [-100, 500],
    rotationDeg: 90,
    draggable: true,
    dragBoundFunc: function(pos) {
        return {
            x: pos.x,
            y: this.getAbsolutePosition().y
        }
    }
});

calque.add(flaçhe1);
calque.add(flaçhe2);
scène.add(calque);
};
```

Tester !



La flèche bleue ne peut être déplacée que verticalement et la rouge seulement horizontalement.

On peut aussi contraindre un déplacement oblique :

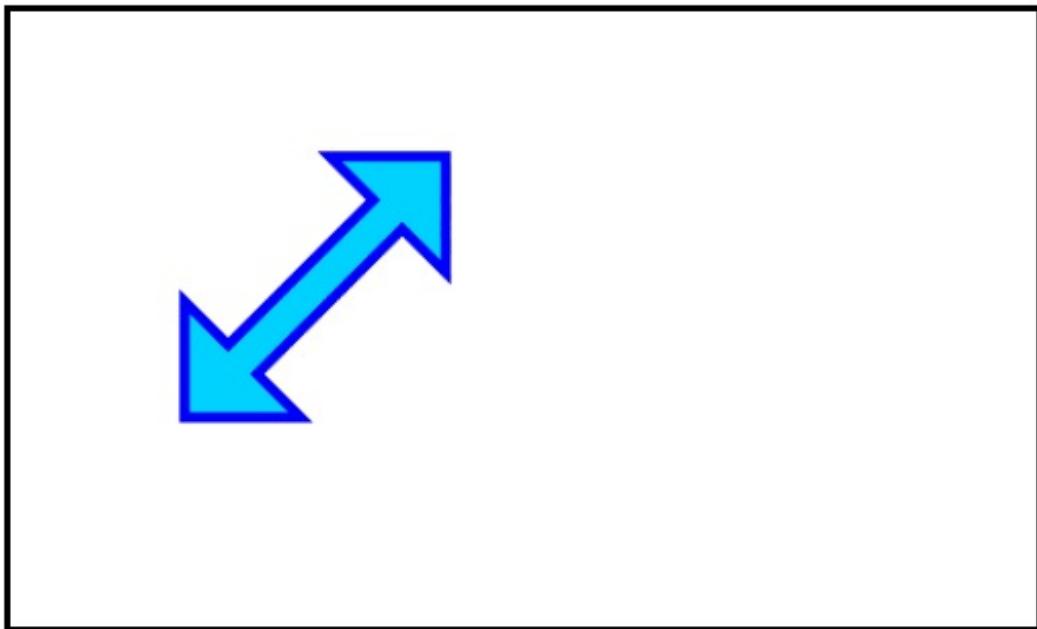
Code : JavaScript

```
window.onload = function() {
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

    var calque = new Kinetic.Layer();
    var trame_flèche =
```

```
[{x:50,y:50},{x:90,y:90},{x:60,y:90},{x:60,y:190},{x:90,y:190},  
 {x:50,y:230},{x:10,y:190},{x:40,y:190},{x:40,y:90},{x:10,y:90}];  
  
var flèche = new Kinetic.Polygon({  
    points: trame_flèche,  
    fill: "#00D2FF",  
    stroke: "#00f",  
    strokeWidth: 5,  
    offset: [-150, 150],  
    draggable: true,  
    rotation: Math.PI / 4,  
    dragBoundFunc: function(pos) {  
        if(pos.x < -85) pos.x = -85;  
        else if(pos.x > 70) pos.x = 70;  
        return {  
            x: pos.x,  
            y: -pos.x  
        }  
    }  
});  
  
calque.add(flèche);  
scène.add(calque);  
};
```

Tester !



## Limite de déplacement

On peut aussi définir des limites de déplacement dans les 4 directions de l'espace. Voici un premier exemple avec un carré dont on limite les déplacements à l'intérieur de la scène pour éviter le débordement :

**Code : JavaScript**

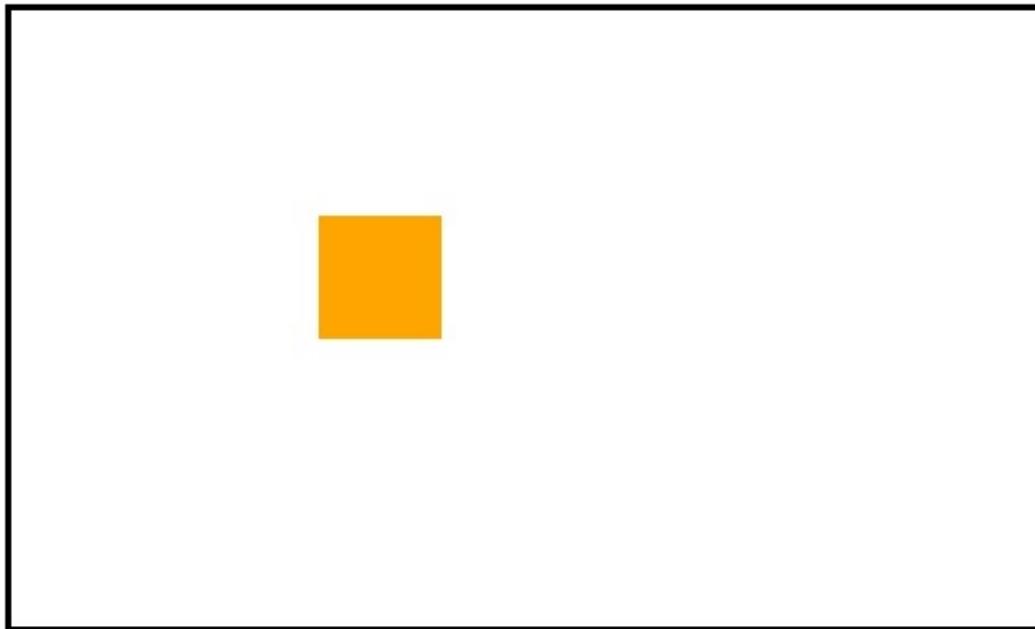
```
window.onload = function() {  
    var scène = new Kinetic.Stage({  
        container: "kinetic",  
        width: 500,  
        height: 300  
    });
```

```
var calque = new Kinetic.Layer();

var carré = new Kinetic.Rect({
    x: 150,
    y: 100,
    width: 60,
    height: 60,
    fill: "orange",
    draggable: true,
    dragBoundFunc: function(pos) {
        if(pos.x < 0) pos.x = 0;
        else if(pos.x > scène.getWidth() - 60) pos.x = scène.getWidth() - 60;
        if(pos.y < 0) pos.y = 0;
        else if(pos.y > scène.getHeight() - 60) pos.y = scène.getHeight() - 60;
        return {
            x: pos.x,
            y: pos.y
        };
    }
});

calque.add(carré);
scène.add(calque);
};
```

Tester !



Voici un autre exemple avec 4 flèches, chacune est limitée en déplacement dans la direction qu'elle indique pour ne pas déborder de la scène :

#### Code : JavaScript

```
window.onload = function() {
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

    var calque = new Kinetic.Layer();
```

```
var trame_flèche = [{x:50,y:50},{x:90,y:90},{x:60,y:90},{x:60,y:160},{x:40,y:160},{x:40,y:90},{x:100,y:50}];

var flèche1 = new Kinetic.Polygon({
    points: trame_flèche,
    fill: "#00D2FF",
    stroke: "#00f",
    strokeWidth: 5,
    offset: [-50, 0],
    draggable: true,
    dragBoundFunc: function(pos) {
        var Y = pos.y < -50 ? -50 : pos.y;
        return {
            x: this.getAbsolutePosition().x,
            y: Y
        }
    }
});

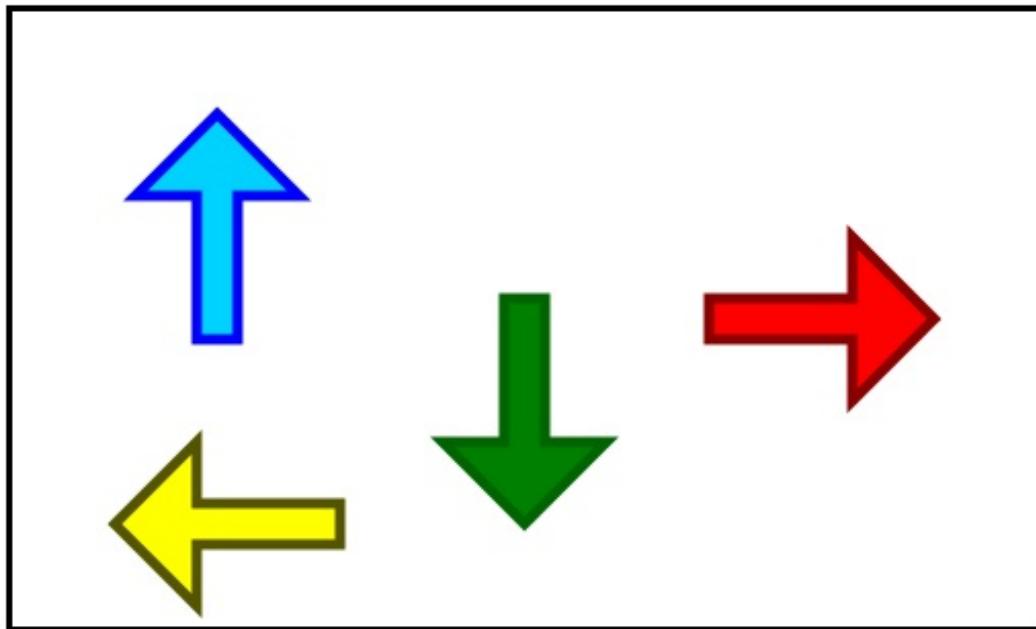
var flèche2 = new Kinetic.Polygon({
    points: trame_flèche,
    fill: "red",
    stroke: "darkred",
    strokeWidth: 5,
    offset: [-100, 500],
    rotationDeg: 90,
    draggable: true,
    dragBoundFunc: function(pos) {
        var X = pos.x > 50 ? 50 : pos.x;
        return {
            x: X,
            y: this.getAbsolutePosition().y,
        }
    }
});

var flèche3 = new Kinetic.Polygon({
    points: trame_flèche,
    fill: "green",
    stroke: "darkgreen",
    strokeWidth: 5,
    offset: [300, 300],
    rotationDeg: 180,
    draggable: true,
    dragBoundFunc: function(pos) {
        var Y = pos.y > 50 ? 50 : pos.y;
        return {
            x: this.getAbsolutePosition().x,
            y: Y
        }
    }
});

var flèche4 = new Kinetic.Polygon({
    points: trame_flèche,
    fill: "yellow",
    stroke: "#550",
    strokeWidth: 5,
    offset: [300, 0],
    rotationDeg: 270,
    draggable: true,
    dragBoundFunc: function(pos) {
        var X = pos.x < -50 ? -50 : pos.x;
        return {
            x: X,
            y: this.getAbsolutePosition().y,
        }
    }
});
```

```
calque.add(flèche1);
calque.add(flèche2);
calque.add(flèche3);
calque.add(flèche4);
scène.add(calque);
};
```

Tester !



## TP Un petit jeu

Je vous propose de réaliser un petit jeu qu'on va faire en deux étapes. Dans la première étape, on dessine un disque et on doit pouvoir le pousser avec le curseur de la souris de telle façon qu'il s'échappe à mesure qu'on le touche avec le curseur :

Attrapez le disque !



Tester !

Il y a bien sûr des tas de façons de réaliser ça, mais vous allez être obligé de choisir le bon événement et de le coder correctement ainsi que tous les éléments à dessiner à l'écran. Vous devez effacer le texte dès que vous commencez le jeu. Il y a la propriété **hide** pour réaliser cela.

**Secret** ([cliquez pour afficher](#))

**Code : JavaScript**

```
function getRandom() {
    if(Math.random() > .5) return 1; else return -1;
}
function getPos(val, maxi) {
    if(val < 20) return (val + 20);
    if(val > maxi - 20) return (val - 20);
    return val + 20 * getRandom();
}
window.onload = function() {
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

    var calque = new Kinetic.Layer();

    var Disque = new Kinetic.Ellipse({
        x: 250,
        y: 150,
        radius: 20,
        fill: "red"
    });

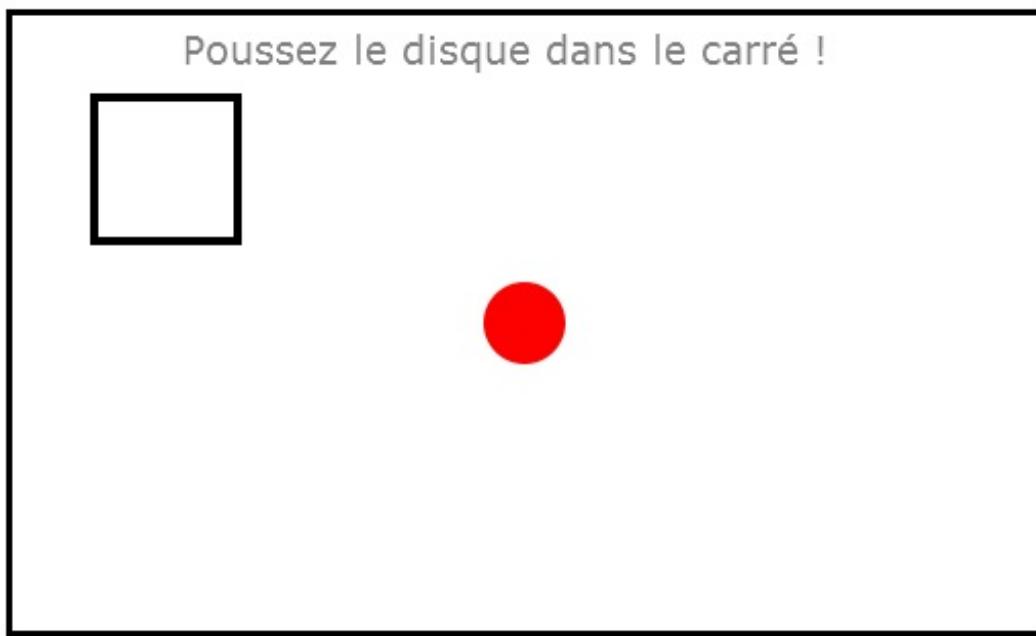
    var Texte = new Kinetic.Text({
        y: 10,
        text: "Attrapez le disque !",
        fontSize: 14,
        fontFamily: "verdana",
        textFill: "grey",
        lineHeight: 2,
        width: 480,
        align: "center"
    });

    Disque.on("mousemove", function() {
        Texte.hide();
        Disque.setX(getPos(Disque.getX(), 500));
        Disque.setY(getPos(Disque.getY(), 300));
        calque.draw();
    });

    calque.add(Disque);
    calque.add(Texte);
    scène.add(calque);
};
```

J'ai choisi l'événement **mousemove** qui me paraît adapté, je vous laisse analyser le code.

Passons à la deuxième étape de notre jeu. cette fois on va un peu améliorer en donnant un objectif, pousser le disque dans un carré :



Tester !

Vous pouvez afficher un message de félicitation à la fin en changeant le texte et en le faisant réapparaître avec la méthode **show**. Vous aurez un peu plus de travail de codage, mais rien de bien difficile :p.

**Secret** ([cliquez pour afficher](#))

**Code : JavaScript**

```
function getRandom() {
    if(Math.random() > .5) return 1; else return -1;
}
function getPos(val, maxi) {
    if(val < 20) return (val + 20);
    if(val > maxi - 20) return (val - 20);
    return val + 20 * getRandom();
}
function getPosBis(val, ref, maxi) {
    if(val < 20) return (val + 20);
    if(val > maxi - 20) return (val - 20);
    if(ref - val > 0) return val - 20;
    else return val + 20;
}
function testGain(x, y) {
    return (Math.abs(x - 75) < 20) && (Math.abs(y - 75) < 20);
}
window.onload = function() {
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

    var calque = new Kinetic.Layer();

    var Rectangle = new Kinetic.Rect({
        x: 40,
        y: 40,
        width: 70,
        height: 70,
        stroke: "black",
        strokeWidth: 4
    });

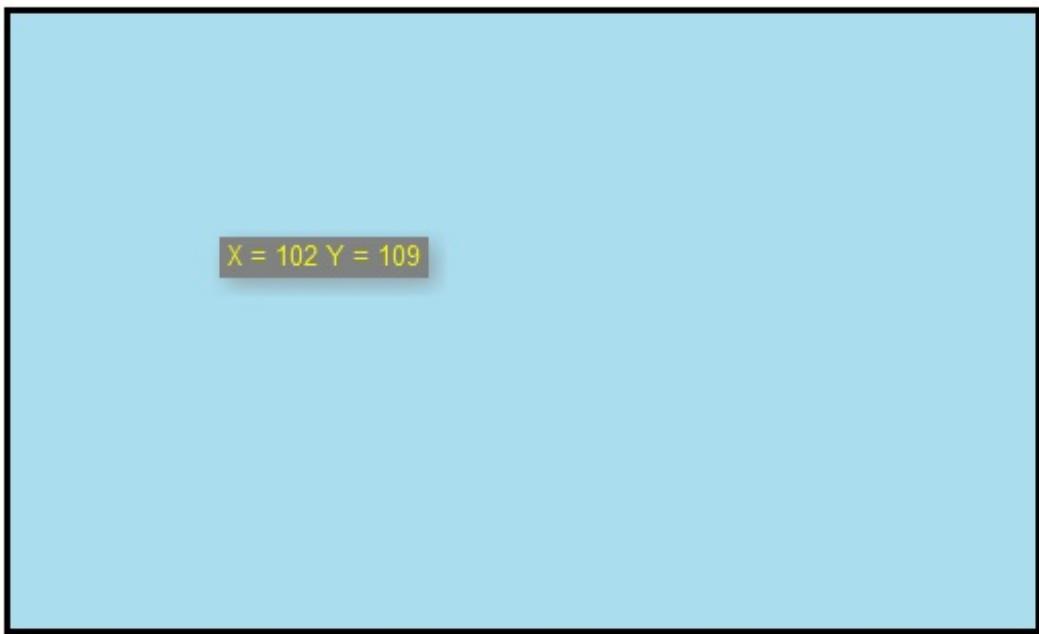
    var Disque = new Kinetic.Ellipse({
```

```
x: 250,  
y: 150,  
radius: 20,  
fill: "red"  
});  
  
var Texte = new Kinetic.Text({  
    y: 10,  
    text: "Poussez le disque dans le carré !",  
    fontSize: 14,  
    fontFamily: "verdana",  
    textFill: "grey",  
    lineHeight: 2,  
    width: 480,  
    align: "center"  
});  
  
Disque.on("mousemove", function() {  
    Texte.hide();  
    var mousePos = scène.getMousePosition();  
    if(Math.random() < .5) {  
        Disque.setX(getPos(Disque.getX(), 500));  
        Disque.setY(getPos(Disque.getY(), 300));  
    }  
    else {  
        Disque.setX(getPosBis(Disque.getX(), mousePos.x, 500));  
        Disque.setY(getPosBis(Disque.getY(), mousePos.y, 300));  
    }  
    if(testGain(Disque.getX(), Disque.getY())) {  
        Texte.setText("Bravo vous avez gagné !");  
        Texte.show();  
        Disque.setX(75);  
        Disque.setY(75);  
        Disque.off("mousemove");  
    }  
    calque.draw();  
});  
  
calque.add(Rectangle);  
calque.add(Disque);  
calque.add(Texte);  
scène.add(calque);  
});
```

Je n'ai pas particulièrement cherché à optimiser le code, le but n'est pas un exercice de style de **Javascript** mais de manipuler les objets de **KineticJS**. J'ai créé quelques sautes d'humeur aléatoires pour le déplacement du disque pour corser un peu le jeu.

## Un tool-tip

Je vous propose maintenant de reprendre l'exemple où on détectait la position du curseur de la souris, mais cette fois en faisant apparaître les valeurs dans un tooltip :



Tester !

Ce qui serait bien c'est que le tooltip n'apparaisse que lorsque le curseur est au-dessus de la scène. Il existe la propriété **visible** qui peut prendre les valeurs **true** ou **false** pour gérer la visibilité d'un objet. On peut changer cette valeur ensuite avec les méthodes **show** et **hide**.

#### Secret (cliquez pour afficher)

##### Code : JavaScript

```
window.onload = function() {
  var scène = new Kinetic.Stage({
    container: "kinetic",
    width: 500,
    height: 300
  });

  var calque = new Kinetic.Layer();

  var Rectangle = new Kinetic.Rect({
    width: scène.getWidth(),
    height: scène.getHeight(),
    fill: "#ade",
  });

  var Tooltip = new Kinetic.Text({
    text: "",
    textFill: "yellow",
    fontFamily: "Arial",
    fontSize: 10,
    padding: 4,
    fill: "grey",
    opacity: 0.9,
    shadow: {
      color: "grey",
      blur: 14,
      offset: [2, 2],
      opacity: 0.7
    },
    visible: false
  });

  scène.on("mousemove", function() {
    var mousePos = scène.getMousePosition();
    Tooltip.setPosition(mousePos);
    Tooltip.show();
  });
}
```

```
var x = mousePos.x;
var y = mousePos.y;
var text = "X = " + x + " Y = " + y;
if(x > 400) x = 400;
if(y > 280) y = 280;
with(Tooltip) {
    show();
    setPosition(x, y);
    setText(text);
}
calque.draw();
});

scène.on("mouseout", function() {
    Tooltip.hide();
    calque.draw();
});

calque.add(Rectangle);
calque.add(Tooltip);
scène.add(calque);
};
```

## Un exemple d'application

Maintenant que nous avons vu les principales fonctionnalités de **KineticJS**, en dehors de ses capacités d'animation, je vous propose d'analyser une application de création graphique modeste, mais qui permet de voir un certain nombre des possibilités de cette librairie dans un environnement pratique.

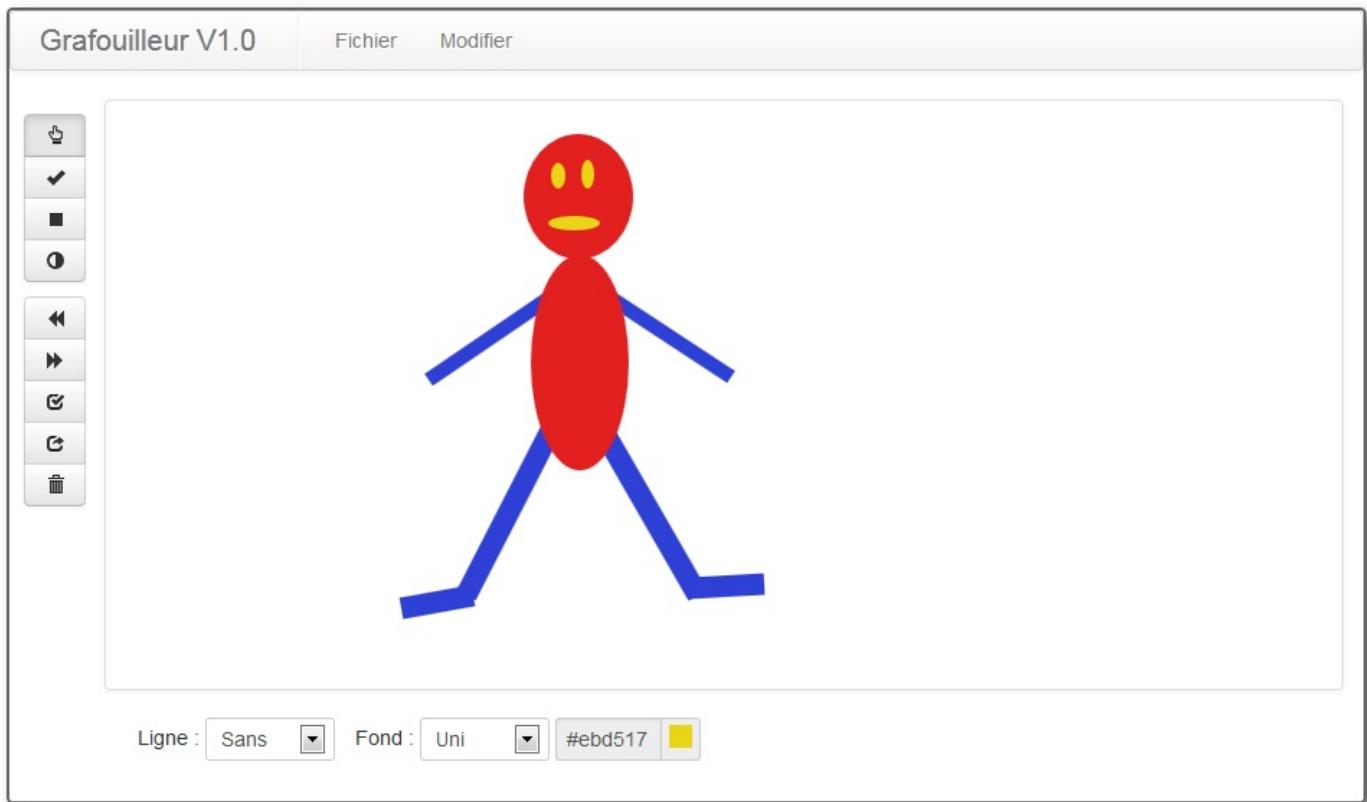
### Présentation

Cette application utilise le kit **CSS Bootstrap de Twitter**, elle me sert également d'exemple pour mon tutoriel sur ce kit. C'est dans ce tutoriel qu'est abordée la partie **HTML** et **CSS**.

Vous pouvez [tester l'éditeur en ligne](#) pour voir son fonctionnement.

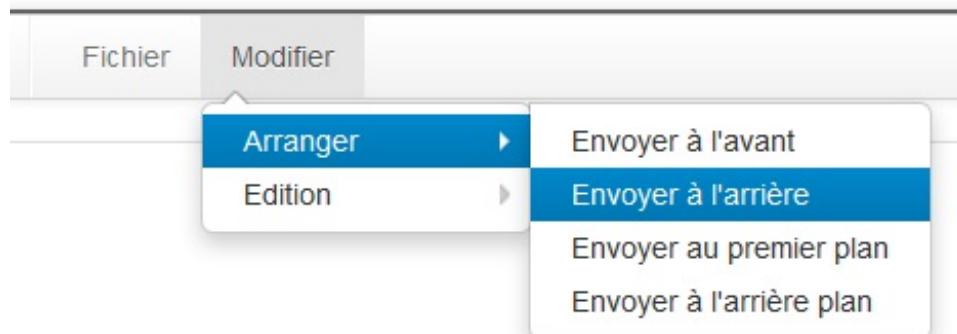
### Aspect de l'interface

L'interface se présente ainsi :



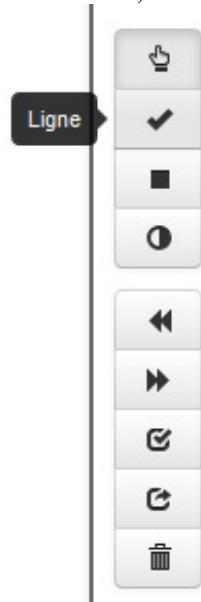
### Les menus

Les menus regroupent un certain nombre de fonctionnalités : réglage du niveau de visibilité des formes dessinées, copier-coller :



### Les barres de boutons

Ces boutons sont regroupés en deux catégories : une pour le dessin, l'autre pour l'édition:



## La barre inférieure

La barre inférieure rassemble les contrôles pour le réglage de l'épaisseur des lignes, leur couleur, et aussi le remplissage des formes :



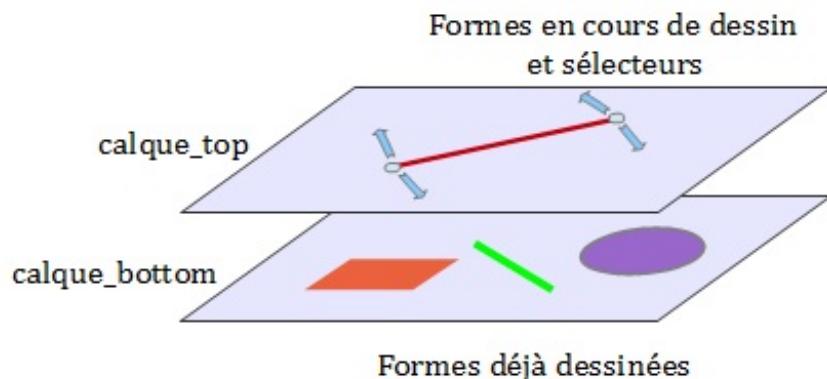
## Il sait faire quoi ?

Pour le fonctionnement on retrouve les opérations standards de ce type d'application :

- Mode sélection ou dessin (ligne, rectangle ou ellipse)
- Sélection au clic avec apparition de sélecteurs pour changer les dimensions
- Formes "draggables" lorsqu'elles sont sélectionnées
- Copier-coller des formes
- Annulation-répétition de toutes les formes
- Changement dynamique de l'épaisseur et de la couleur des traits
- Changement dynamique du remplissage
- Réglage de l'ordre de superposition des formes
- Enregistrement au format "image"
- Enregistrement au format JSON

## Principe de fonctionnement

Dans ce type d'application il faut considérer deux états : les formes déjà dessinées et les formes en cours de dessin ainsi que les sélecteurs. Pour gérer efficacement ces deux types d'objets on va utiliser deux calques. Un calque inférieur (calque\_bottom) pour contenir les formes déjà dessinées, et un calque supérieur (calque\_top) pour contenir les formes transitoires en cours de création et les sélecteurs servant à changer leur dimension. Voici une illustration de cette organisation :



Ces deux calques sont évidemment contenus dans une scène. On trouve donc au niveau du code ces éléments de base :

#### Code : JavaScript

```
// Création de la scène
var scène = new Kinetic.Stage({
    container: "kinetic",
    width: 850,
    height: 400
});

// Création des calques
var calque_top = new Kinetic.Layer();
var calque_bottom = new Kinetic.Layer();

// Ajout des calques à la scène
scène.add(calque_bottom);
scène.add(calque_top);
```

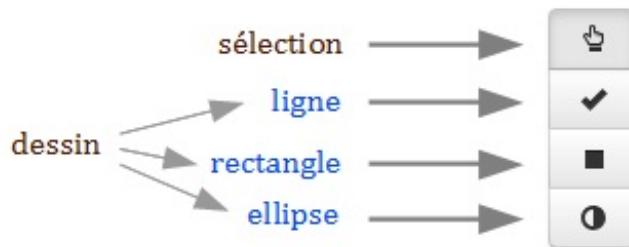
Les variables globales sont rassemblées dans un objet pour plus de clarté au niveau du code :

#### Code : JavaScript

```
// Paramètres globaux
var params = {
    modes: ['selection', 'ligne', 'rectangle', 'ellipse'],
    id_mode: 0,
    line_color: 'black',
    line_width: 2,
    fill_color: 'white',
    line_type: 'Unie',
    fill_type: 'Sans',
    en_cours: false,
    start_point: [],
    objet_modif: {},
    selecteurs: [],
    box: {},
    objets_effaces: [],
    objet_copie: null,
    inhibe_selection: false
};
```

### Le dessin

Lorsqu'on veut dessiner l'une des trois formes à disposition (ligne, rectangle ou ellipse) on commence par choisir cette forme en cliquant sur le bouton correspondant :



Ce qui a pour effet de transformer le curseur pour afficher une croix. Le processus de dessin peut alors avoir lieu, il s'effectue en 3 étapes :

1. Un appui sur le bouton de la souris pour fixer le point initial de la forme
2. Un mouvement de la souris pour dessiner une forme transitoire tout en gardant le bouton enfoncé
3. Un relâchement du bouton de la souris pour dessiner définitivement la forme

Nous avons vu que les événements de la souris peuvent être captés sur les **nodes** de **KineticJS**, par contre lorsqu'on est sur une zone vide du **canvas** ça ne marche plus. Il faut donc prévoir les événements de la souris directement sur les **canvas** :

#### Code : JavaScript

```
// Événements de souris sur le canvas
$( 'canvas' )
  .mousedown( function(e) {
    // Etape 1
  })
  .mousemove( function(e) {
    // Etape 2
  })
  .mouseup( function(e) {
    // Etape 3
  });
});
```

### Etape 1 : fixation du point initial

Lorsqu'on clique pour commencer à dessiner une forme il nous faut déterminer les coordonnées du curseur et les mémoriser :

#### Code : JavaScript

```
// Position de la souris
var offset = $('.kineticjs-content').offset();
var x = e.pageX - offset.left;
var y = e.pageY - offset.top;
// Mémorisation de la position
params.start_point = [x, y];
params.en_cours = true;
```

Les coordonnées sont enregistrées dans la variable **start\_point** et on renseigne aussi la variable **en\_cours** qui indique qu'on est en train de dessiner une forme.

### Etape 2 : positionnement sur le point final

Dans cette étape on garde le bouton de la souris pressé et on la déplace, la forme doit suivre de façon dynamique à l'écran sur le calque **calque\_top** prévu pour les éléments transitoires. Extrayons de l'événement **mousemove** les éléments essentiels :

**Code : JavaScript**

```
.mousemove (function(e) {  
    // Si mode sélection  
    if(params.modes[params.id_mode] == 'selection') {  
        // Mode sélection que nous voyons plus loin  
    }  
    // Si mode dessin  
    else if(params.en_cours) {  
        // Position de la souris  
        var offset = $('.kineticjs-content').offset();  
        var x = e.pageX - offset.left;  
        var y = e.pageY - offset.top;  
        ...  
        if(params.modes[params.id_mode] == 'ligne') {  
            // Dessin d'une ligne  
        }  
        // Cas du rectangle  
        else if(params.modes[params.id_mode] == 'rectangle') {  
            // Dessin d'un rectangle  
        }  
        // Cas de l'ellipse  
        else if(params.modes[params.id_mode] == 'ellipse') {  
            // Dessin d'une ellipse  
        }  
        calque_top.draw();  
    }  
})
```

On récupère bien sûr encore les coordonnées de la souris. On veut ensuite savoir ce qu'on a à dessiner, on prévoit donc 3 cas, et en dernier on rafraîchit le calque avec sa méthode **draw**. prenons par exemple le cas d'une ligne, voilà un autre extrait de code :

**Code : JavaScript**

```
// Shape en cours  
var shapes = calque_top.getChildren();  
// Cas de la ligne  
if(params.modes[params.id_mode] == 'ligne') {  
    // Ligne déjà créée  
    if(shapes.length) {  
        shapes[0].setPoints(params.start_point.concat([x, y]));  
    }  
    // Création de la ligne  
    else {  
        var ligne = new Kinetic.Line({  
            points: params.start_point.concat([x, y]),  
            stroke: params.line_color,  
            strokeWidth: params.line_width  
        });  
        calque_top.add(ligne);  
    }  
}
```

On doit envisager deux situations :

1. la ligne n'a pas encore été créée : il faut donc le faire
2. la ligne a déjà été créée : on peut alors se contenter d'actualiser sa position

La méthode **getChildren** permet de récupérer un tableau de tous les **nodes** présents sur un calque. Donc avec cette ligne :

#### Code : JavaScript

```
var shapes = calque_top.getChildren();
```

On obtient dans le tableau **shapes** ce qu'on a déjà dessiné, autrement dit rien si c'est juste le début, ou une **shape** si on a commencé le tracé. Donc si la ligne est déjà créée :

#### Code : JavaScript

```
// Ligne déjà créée
if(shapes.length) {
    shapes[0].setPoints(params.start_point.concat([x, y]));
}
```

On se contente de mettre à jour sa propriété **points**. Par contre si elle n'a jamais été créée il faut le faire en prenant les paramètres **line\_color** et **line\_width** actuels :

#### Code : JavaScript

```
// Création de la ligne
else {
    var ligne = new Kinetic.Line({
        points: params.start_point.concat([x, y]),
        stroke: params.line_color,
        strokeWidth: params.line_width
    });
    calque_top.add(ligne);
}
```

C'est évidemment le même scénario pour les carrés et les ellipses...

## Etape 3 : dessin définitif

Lorsqu'on est satisfait de la dimension et de la position de la forme dessinée, on relâche le bouton de la souris. Cela doit effacer la forme du calque **calque\_top** pour la transférer sur le calque **calque\_bottom**. On doit aussi passer en mode "sélection" pour cette forme. Il faut aussi actualiser les boutons de mode dans la barre verticale. Voyons cela :

#### Code : JavaScript

```
// Transfert sur calque inférieur
params.objet_modif = calque_top.getChildren()[0];
params.objet_modif.moveTo(calque_bottom);
params.objet_modif.setDraggable(true);
calque_bottom.draw();

...
reset_dessin();
```

La variable **objet\_modif** doit référencer la forme en cours de sélection, c'est ce qui est fait sur la ligne 2. Ensuite la méthode **moveTo** de la **shape** permet son transfert sur le calque **calque\_bottom**. On veut qu'un objet sélectionné soit "draggable", c'est l'objet de la ligne 4 où on met la propriété **draggable** à **true**. Il suffit au final d'actualiser ce calque avec sa méthode **draw**. La fonction **reset\_dessin** a pour but d'actualiser des variables et de passer en mode "sélection" :

#### Code : JavaScript

```
// Reset du dessin
function reset_dessin() {
    params.en_cours = false;
    params.id_mode = 0;
    $('#canvas').css('cursor', 'default');
    $('#mode .active').removeClass('active');
    $('#mode .btn:first').addClass('active');
}
```

La variable **en\_cours** doit être à **false** puisqu'on ne dessine plus. La variable **id\_mode** à 0 indique qu'on est en mode "sélection". Le curseur est remis normal. Et enfin on rend actif le bouton de sélection dans la barre de boutons verticale.

## Sélection et édition

### Sélection automatique de la forme qu'on vient de dessiner

On a vu que lorsqu'on dessine une forme elle passe en mode "sélection", il doit en être de même lorsqu'on clique sur une forme dans ce mode. Voyons déjà le premier cas que nous avons laissé en suspens :

#### Code : JavaScript

```
.mouseup(function(e) {
    // Dessin effectif
    if(params.modes[params.id_mode] != 'selection' && params.en_cours)
    {
        // Transfert sur calque inférieur
        ...

        // Mise en place événements de drag
        params.objet_modif.on("dragstart", function() {
            calque_top.removeChild();
            calque_top.draw();
        });
        params.objet_modif.on("dragend", function() {
            select_object();
            calque_top.draw();
        });
        select_object();
        calque_top.draw();

        ...
    }
});
```

Commençons par nous intéresser à la fonction **select\_object** qui est surlignée. Le but de cette fonction est de dessiner les sélecteurs pour modifier la forme à l'écran. Ces sélecteurs doivent évidemment se trouver sur le calque **calque\_top** :

#### Code : JavaScript

```
// Sélection d'un objet
function select_object() {
    if(params.objet_modif.shapeType == "Line") {
        var points = params.objet_modif.getPoints();
        var pos = params.objet_modif.getPosition();
        params.selecteurs = [];
    }
}
```

```
    params.selecteurs.push(build_selector(points[0].x + pos.x,
    points[0].y + pos.y));
    params.selecteurs.push(build_selector(points[1].x + pos.x,
    points[1].y + pos.y));
}
if(params.objet_modif.shapeType == "Rect") {
    ...
}

if(params.objet_modif.shapeType == "Ellipse") {
    ...
}

$( 'canvas' ).css( 'cursor', 'default' );
}
```

On procède évidemment différemment selon qu'il s'agit d'une ligne, d'un rectangle ou d'une ellipse. J'ai juste gardé le cas de la ligne dans le code ci-dessus. On se rend compte qu'on récupère la valeur de la propriété `points` de la ligne sélectionnée dans la variable `points`. On récupère aussi sa position dans la variable `pos`.



Mais pourquoi aller chercher cette propriété `position` ? On a dit qu'une ligne est parfaitement définie avec sa propriété `points` !

Oui et non. Lorsqu'on dessine une ligne en général on ne s'inquiète pas de sa position puisqu'on renseigne les points mais... si la `shape` est "draggable" et qu'on la bouge à l'écran la propriété `points` n'est pas modifiée, c'est la propriété `position` qui l'est ! Et comme on a dit que la `shape` sélectionnée est draggable il vaut mieux récupérer cette information pour dessiner les sélecteurs au bon endroit 😊.

Quand on a cette position, on peut dessiner les sélecteurs et les mémoriser dans la variable `selecteurs`. C'est la fonction `build_selector` qui est chargée de ce dessin :

#### Code : JavaScript

```
// Création des sélecteurs
function build_selector(x, y) {
    var selector = new Kinetic.Circle({
        x: x,
        y: y,
        radius: 5,
        stroke: "#666",
        fill: "#ddd",
        strokeWidth: 2,
        draggable: true,
        name: 'selector'
    });

    selector.on("mouseover", function() {
        $( 'canvas' ).css( 'cursor', 'pointer' );
        this.setOpacity(.1);
        inhibe_selection = true;
        calque_top.draw();
    });
    selector.on("mouseout", function() {
        $( 'canvas' ).css( 'cursor', 'default' );
        this.setOpacity(1);
        inhibe_selection = false;
        calque_top.draw();
    });

    calque_top.add(selector);
    return selector;
}
```

Dans la première partie, on dessine un cercle en le rendant "draggable". On met ensuite en place deux événements. Lorsque le curseur est sur le sélecteur (**mouseover**) on change son aspect, on réduit l'opacité pour presque faire disparaître le sélecteur (comme ça on voit mieux où on bouge l'extrémité d'une ligne), on met à **true** la variable **inhibe\_selection**, ce qui va nous éviter de sélectionner une autre shape alors qu'on veut en modifier une, et enfin on actualise le calque. Évidemment dans l'événement opposé (**mouseout**) on fait exactement l'inverse 😊.

Revenons maintenant un peu au moment où on a lâché le bouton de la souris pour dessiner la **shape** :

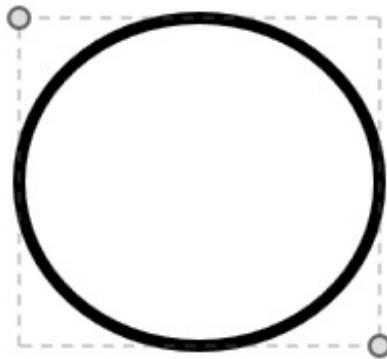
#### Code : JavaScript

```
.mouseup(function(e) {
    // Dessin effectif
    if(params.modes[params.id_mode] != 'selection' && params.en_cours) {
        ...
        // Mise en place événements de drag
        params.objet_modif.on("dragstart", function() {
            calque_top.removeChild();
            calque_top.draw();
        });
        params.objet_modif.on("dragend", function() {
            select_object();
            calque_top.draw();
        });
        ...
    }
});
```

C'est là qu'on met en place les événements du "glisser" pour supprimer les sélecteurs au début du mouvement et en redessiner à la fin de ce mouvement.

## Cas de l'ellipse

L'ellipse nécessite plus que des sélecteurs pour son édition étant donné sa forme particulière. Il est judicieux de représenter visuellement le cadre qui l'enveloppe :



Vous retrouvez cela au niveau du code pour l'ellipse :

#### Code : JavaScript

```
if(params.objet_modif.shapeType == "Ellipse") {
    var x = params.objet_modif.getX();
    var y = params.objet_modif.getY();
    var radius = params.objet_modif.getRadius();
```

```
params.selecteurs = [];
params.box = build_box(x - radius.x, y - radius.y, x + radius.x, y +
radius.y);
params.selecteurs.push(build_selector(x - radius.x, y - radius.y));
params.selecteurs.push(build_selector(x + radius.x, y + radius.y));
}
```

J'ai surligné le code particulier. On voit la création de ce cadre avec la fonction **build\_box** :

#### Code : JavaScript

```
// Cr ation du cadre en pointill s
function build_box(x1, y1, x2, y2) {
    var box = new Kinetic.Line({
        points: [x1, y1, x2, y1, x2, y2, x1, y2, x1, y1],
        dashArray: [5, 5],
        strokeWidth: 1,
        stroke: "#666",
        lineCap: "round",
        id: "box",
        opacity: 0.6
    });
    calque_top.add(box);
    return box;
}
```

Un trac  classique de ligne en renseignant correctement la propri t  **points**.

## S lection d'une forme avec un clic

On doit pouvoir s lectionner n'importe qu'elle forme dessin e en cliquant dessus. Alors petit retour en arri re sur l' v nement **mousedown** :

#### Code : JavaScript

```
$( 'canvas' )
.mousedown(function(e) {
    // Si mode s lection
    if(params.modes[params.id_mode] == 'selection') {
        if(!inhibe_selection) {
            var offset = $('.kineticjs-content').offset();
            var p = [e.pageX - offset.left, e.pageY - offset.top];
            var shapes = sc ne.getIntersections(p);
            if(shapes.length && shapes[0].getLayer() != calque_top &&
            shapes[0] != params.objet_modif) {
                calque_top.removeChildren();
                shapes[0].setDraggable(true);
                params.objet_modif = shapes[0];
                select_object();
                sync_controls();
                calque_top.draw();
            }
        }
    }
    // Si mode dessin
    else {
        ...
    }
})
```

J'ai surligné la ligne de code la plus importante. Elle contient la méthode **getIntersections** qui permet de connaître les shapes dont le point passé en paramètre fait partie de leur tracé. On obtient au retour un tableau des **shapes** concernées. Dans notre cas on demande les **shapes** qui se trouvent à la position du curseur. Pour simplifier, on ne considère que la première du tableau, celle avec l'index 0.

Si on en trouve une, on commence par enlever les sélecteurs éventuellement présents sur le calque `calque_top` avec la méthode **removeChildren** de ce calque. On met la **shape** qu'on veut sélectionner "draggable". On renseigne la variable **objet\_modif** avec cette **shape**. On active la sélection avec la fonction **select\_object** que nous avons vue ci-dessus. Pour ce qui concerne la synchronisation des contrôles nous le verrons plus loin. On finit par rafraîchir le calque. On se retrouve alors exactement dans la même situation que vu juste avant de la **shape** sélectionnée après son tracé.

## Désactivation de la sélection par touche d'échappement

On veut ne plus avoir de forme sélectionnée en appuyant sur la touche d'échappement :

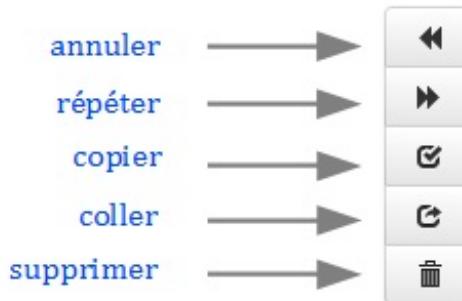
**Code : JavaScript**

```
// Annulation sur escape, ou destruction sur del
$(document).keydown(function(e) {
    // Annulation
    if(e.keyCode == 27) {
        if(params.modes[params.id_mode] == 'selection' || params.en_cours)
        {
            reset_dessin();
            params.objet_modif = {};
            calque_top.removeChildren();
            calque_top.draw();
        }
    }
    // Delete
    else if(e.keyCode == 46) del_object();
});
```

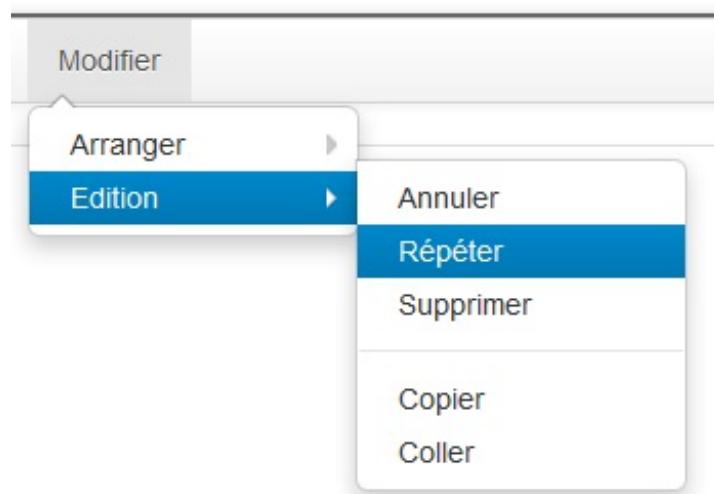
Là c'est plus simple : "reset" du dessin, vidage de la variable **objet\_modif** puisque plus rien ne va être sélectionné, effacement des sélecteurs et rafraîchissement du calque.

## Édition

J'ai rassemblé ces commandes parce qu'elles partagent un certain nombre de concepts. Elles sont mises en œuvre à partir de la barre de boutons :



Ou du menu :



### Annuler

Voici la fonction qui effectue cette opération :

#### Code : JavaScript

```
// Annuler dernière création d'objet
function annuler_objet() {
    reset_dessin();
    calque_top.removeChildren();
    calque_top.draw();
    var shapes = calque_bottom.getChildren();
    if(shapes.length) {
        params.objets_effaces.push(shapes[shapes.length - 1]);
        calque_bottom.remove(shapes[shapes.length - 1]);
        calque_bottom.draw();
    }
}
```

On fait un "reset" du dessin et on enlève tout ce qui se trouve sur le calque supérieur puisqu'on retire la **shape** sélectionnée. Ensuite on récupère un tableau de toutes les **shapes** présentes sur le calque **calque\_bottom** avec la méthode **getChildren**. Un petit test pour voir s'il y en a (des petits malins peuvent cliquer sur le bouton alors que rien n'est dessiné) et on envoie une référence de la dernière **shape** dessinée (celle qui est à la fin du tableau avec l'index *length - 1*) dans le tableau **objets\_effaces** qui contient les objets... effacés. On peut alors retirer la **shape** du calque avec la méthode **remove**. On finit évidemment avec un rafraîchissement du calque.

### Répéter

Cette commande est exactement l'inverse de la précédente puisqu'on veut faire passer une **shape** du tableau des effacées vers le calque **calque\_bottom** :

#### Code : JavaScript

```
// Répéter dernière création d'objet
function repete_objet() {
    if(params.objets_effaces.length) {
        reset_dessin();
        calque_top.removeChildren();
        calque_top.draw();
        params.objet_modif = {};
        var shape = params.objets_effaces.pop();
        calque_bottom.add(shape);
        // Nécessaire pour retrouver la détection
        Kinetic.Global.shapes[shape.colorKey] = shape;
        calque_bottom.draw();
    }
}
```

```
}
```

J'ai surligné les deux lignes importantes : on récupère la forme dans la variable **shape** avec un **pop** du tableau **objets\_effaces**. Il suffit ensuite d'ajouter cette **shape** avec la méthode **add**. Il y a également une ligne de code un peu spéciale à cause d'une particularité de **KineticJS** qui fait que l'on perd la possibilité de sélection d'une forme quand on l'enlève et la remet sur un calque. Cette ligne est destinée à faire en sorte que la forme soit toujours repérable, en particulier pour le "glisser".

### Copier

La copie se contente de mémoriser la forme sélectionnée dans la variable **objet\_copie** et le code tient en une seule ligne :

#### Code : JavaScript

```
params.objet_copie = params.objet_modif.clone();
```

La méthode **clone** crée un nouvel objet identique.

### Coller

Le collage est l'inverse de la copie et le code se passe de commentaire :

#### Code : JavaScript

```
if(params.objet_copie != null) {
    calque_bottom.add(params.objet_copie.clone());
    calque_bottom.draw();
}
```

### Supprimer

La suppression est très proche de l'annulation, si cette dernière enlève la dernière forme tracée, la suppression quant à elle enlève la forme sélectionnée :

#### Code : JavaScript

```
// Suppression d'un objet
function del_object() {
    if(test_select()) {
        params.objets_effaces.push(params.objet_modif);
        calque_bottom.remove(params.objet_modif);
        calque_bottom.draw();
        reset_dessin();
        calque_top.removeChild();
        calque_top.draw();
    }
}
```

Le code se passe de commentaire parce qu'il reprend ce que nous avons déjà vu.

## Apparence et enregistrement

### Contrôle de l'apparence

Nous allons voir à présent comment s'effectue le réglage de l'épaisseur des traits, leur couleur, ainsi que le traitement du remplissage. Lorsqu'on sélectionne une forme, il faut que les contrôles se mettent à jour :

**Code : JavaScript**

```
// Si mode sélection
if(params.modes[params.id_mode] == 'selection') {
    if(!inhibe_selection) {

        ...

        if(shapes.length && shapes[0].getLayer() != calque_top &&
shapes[0] != params.objet_modif) {

            ...

            sync_controls();

            ...

        }
    }
}
```

Voyons cette fonction **sync\_controls** :

**Code : JavaScript**

```
// Synchronisation des contrôles avec la shape sélectionnée
function sync_controls() {
    // Largeur de ligne
    if(params.objet_modif.getStrokeWidth() != undefined)
        $('#width').val(params.objet_modif.getStrokeWidth());
    // Couleur de ligne
    if(params.objet_modif.getStroke() != undefined) {
        $('#typeligne').val('Unie');
        $('#colorligne').val(params.objet_modif.getStroke());
        $('#couleurligne i').css('background-color',
            params.objet_modif.getStroke());
        $('#couleurligne').removeClass('hide');
        $('#widthline').removeClass('hide');
    }
    else {
        $('#typeligne').val('Sans');
        $('#couleurligne').addClass('hide');
        $('#widthline').addClass('hide');
    }
    // Couleur de remplissage
    if(params.objet_modif.setFill() != undefined) {
        $('#typefond').val('Uni');
        $('#couleurfond').removeClass('hide');
        $('#colorremplissage').val(params.objet_modif.setFill());
        $('#couleurfond i').css('background-color',
            params.objet_modif.setFill());
    }
    else {
        $('#couleurfond').addClass('hide');
        $('#typefond').val('Sans');
    }
}
```

Une utilisation intensive de **jQuery** permet de renseigner les contrôles correspondants, de les cacher ou les afficher quand c'est nécessaire. On utilise des **setter** de **kineticJS** pour récupérer les informations : **getStrokeWidth**, **getStroke**, **getFill**.

Je ne vais pas rentrer dans le détail du processus inverse, c'est à dire du changement au niveau de la **shape** sélectionnée quand on modifie la valeur d'un contrôle. Voici par exemple le cas de l'épaisseur de la ligne :

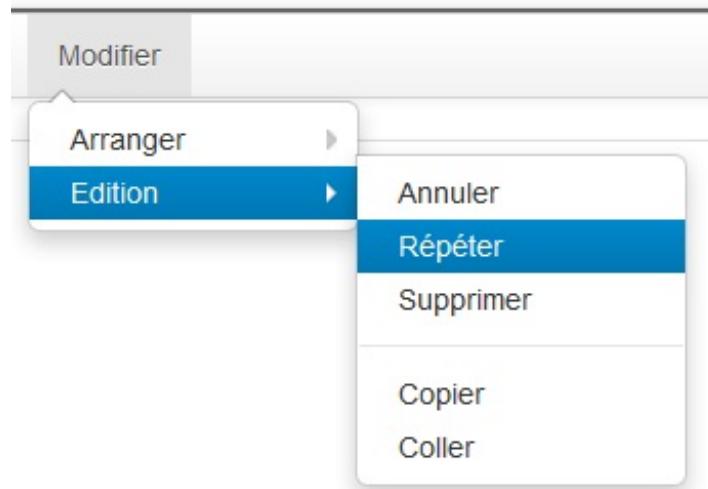
**Code : JavaScript**

```
// Largeur de ligne
$('#width').change(function() {
    params.line_width = $(this).val();
    if(test_select() && params.objet_modif.getStrokeWidth() != 'undefined') {
        params.objet_modif.setStrokeWidth(params.line_width);
        calque_bottom.draw();
    }
});
```

Le setter **setStrokeWidth** permet de mettre à jour la **shape**. C'est le même principe pour les autres contrôles.

## Ordre de superposition des formes

L'ordre de superposition des formes peut être changé par le menu :



**KineticJS** nous permet de faire cela très simplement :

### Code : JavaScript

```
$('#go_front').click(function() {
    if(test_select()) {
        params.objet_modif.moveToTop();
        calque_bottom.draw();
    }
});
$('#go_back').click(function() {
    if(test_select()) {
        params.objet_modif.moveToBottom();
        calque_bottom.draw();
    }
});
$('#go_up').click(function() {
    if(test_select()) {
        params.objet_modif.moveUp();
        calque_bottom.draw();
    }
});
$('#go_down').click(function() {
    if(test_select()) {
        params.objet_modif.moveDown();
        calque_bottom.draw();
    }
});
```

Les méthodes **moveToTop**, **moveToBottom**, **moveUp** et **moveDown** effectuent ces opérations.

## Enregistrement en image ou en JSON

Pour terminer, j'ai prévu l'enregistrement du dessin au format "image" ou **JSON**.

### *Image*

L'enregistrement direct d'une image est évidemment impossible avec **JavaScript** mais on peut par contre envoyer le dessin sur une page, il ne reste plus alors qu'à enregistrer l'image :

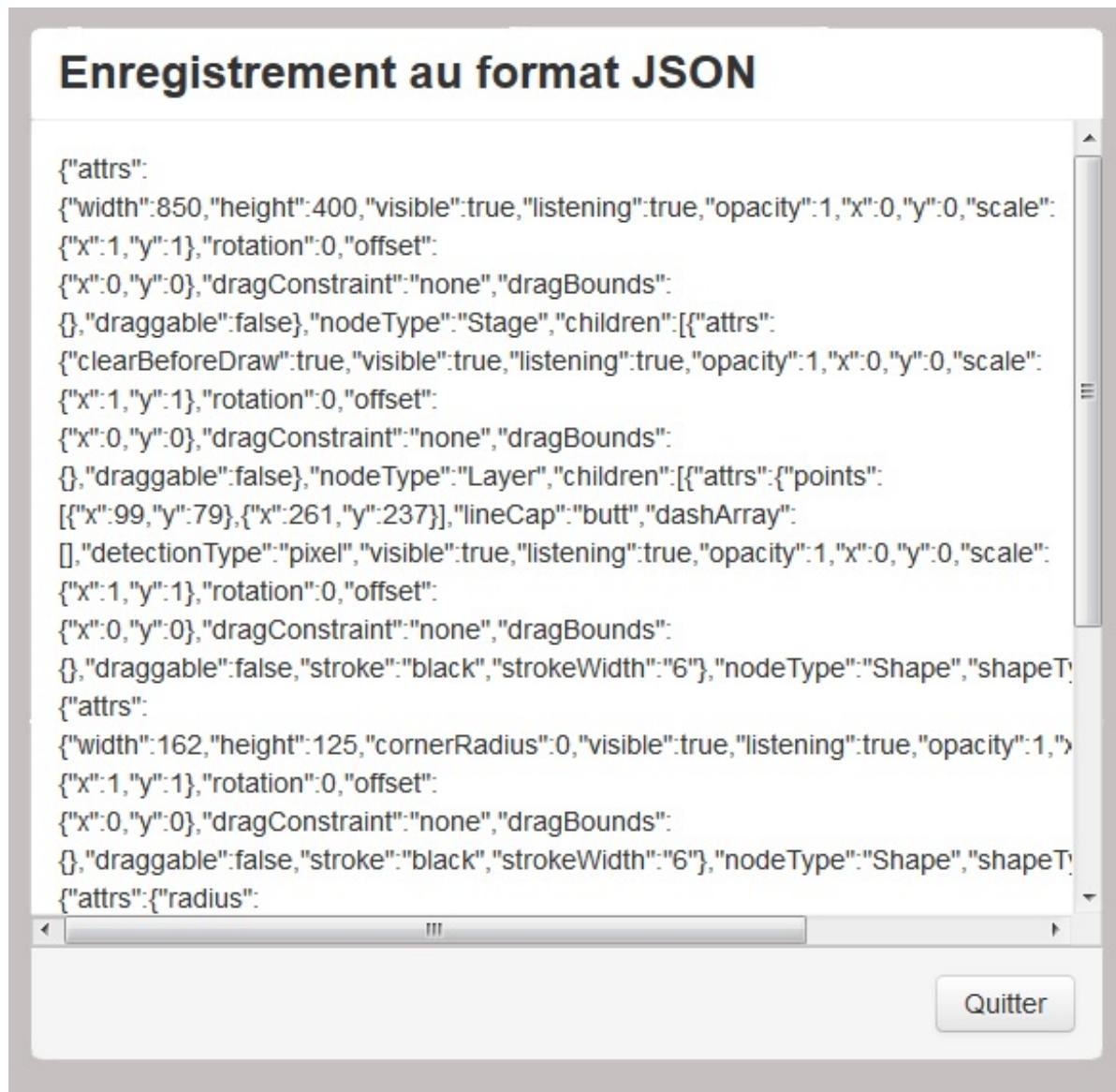
#### Code : JavaScript

```
$('#enregistrer_image').click(function() {
    reset_dessin();
    calque_top.removeChild();
    calque_top.draw();
    scène.toDataURL({
        callback: function(dataUrl) {
            window.open(dataUrl);
        }
    });
});
```

On commence par enlever ce qui pourrait se trouver sur la calque **calque\_top**. On utilise ensuite la méthode **toDataURL** et dans le callback on ouvre une fenêtre avec l'image.

### *JSON*

Pour le JSON j'ai utilisé une fenêtre modale :



Le code est aussi assez simple :

#### Code : JavaScript

```
$('#enregistrer_json').click(function() {  
    reset_dessin();  
    calque_top.removeChild();  
    calque_top.draw();  
    $('#text_json').text(scène.toJSON());  
    $('#modal_save').modal();  
});
```

La méthode utilisée est **toJSON**.

## Animations

Nous allons maintenant voir la partie la plus intéressante avec les possibilités d'interaction de **KineticJS** et ses capacités d'animation.

### Les transitions

#### Une transition simple pour commencer

Pour réaliser une transition sur une forme on utilise la méthode **transitionTo** qui attend un objet contenant les propriétés à changer avec leur valeur à atteindre. On ajoute la durée du phénomène et le tour est joué 😊.

Voici un premier exemple où on part d'une étoile, on fait varier sa position, sa rotation, sa taille, le tout sur une durée de 6 secondes :

#### Code : JavaScript

```
window.onload = function() {
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

    var calque = new Kinetic.Layer();

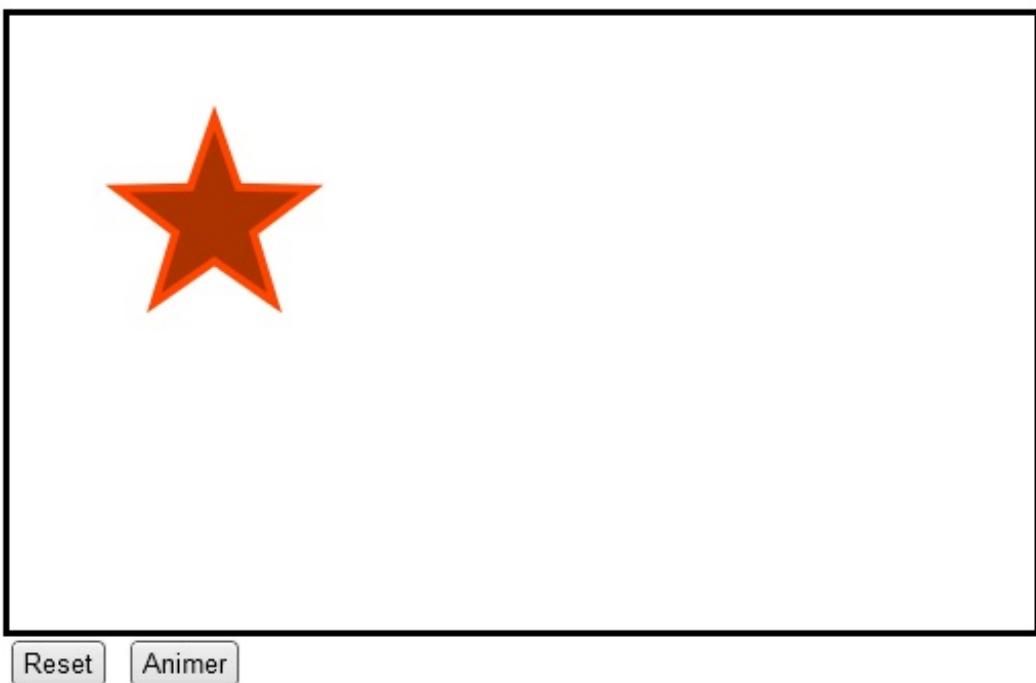
    var étoile = new Kinetic.Star({
        x: 100,
        y: 100,
        numPoints: 5,
        innerRadius: 20,
        outerRadius: 50,
        fill: "#a30",
        stroke: "orangered",
        scale: 1,
        strokeWidth: 4
    });

    calque.add(étoile);
    scène.add(calque);

    document.getElementById("animer").addEventListener("click",
    function() {
        étoile.transitionTo({
            x: 400,
            y: 200,
            rotation: Math.PI,
            scale: {x: 1.6, y: 1.6},
            duration: 6
        });
        , false);
    }

    document.getElementById("reset").addEventListener("click",
    function() {
        with(étoile) {
            setX(100);
            setY(100);
            setRotation(0);
            setScale({x: 1, y: 1});
        }
        calque.draw();
    }, false);
};
```

Tester !



J'ai utilisé deux boutons, un pour l'animation et un autre pour retrouver l'état initial. La partie intéressante est celle qui est surlignée dans le code, avec l'utilisation de la méthode **transitionTo**. On se rend compte qu'on se contente de renseigner les propriétés et qu'on ajoute la durée de la transition. La transformation est linéaire, il existe une propriété pour obtenir un effet non linéaire, c'est ce que nous allons voir à présent. Comme nous abordons les animations, nous allons aussi commencer à utiliser plusieurs calques pour séparer les éléments fixes et les éléments mobiles et éviter ainsi de dessiner des choses inutiles lors des mouvements.

## Les effets de mouvement

Il y a trois catégories d'effets :

- Amortissement
- Élastique
- Rebond

Nous allons voir des exemples de ces effets.

### Amortissement

L'amortissement est une variation sinusoïdale des valeurs changées. On peut définir cet amortissement en début de mouvement, en fin, ou les deux. Il suffit de renseigner correctement la propriété **easing** :

Valeur	effet
<b>ease-in</b>	Amortissement en début de transition
<b>ease-out</b>	Amortissement en fin de transition
<b>ease-in-out</b>	Amortissement en début et fin de transition

Voici un exemple illustratif :

#### Code : JavaScript

```
window.onload = function() {
  var scène = new Kinetic.Stage({
    container: "kinetic",
```

```
        width: 500,
        height: 300
    });

var trame = new Kinetic.Layer();
var balles = new Kinetic.Layer();
var texte = ["ease-in", "ease-out", "ease-in-out"];

// Trame de fond
var rectangle = new Kinetic.Rect({
    width: 500,
    height: 300,
    fill: "#eee",
});
trame.add(rectangle)
for(var i = 100; i < 300; i += 100) {
    trame.add(
        new Kinetic.Line({
            points: [0, i, 500, i],
            strokeWidth: 3
        })
    );
}
for(var i = 0; i < 3; ++i) {
    trame.add(
        new Kinetic.Text({
            y: 78 + i * 100,
            text: texte[i],
            fontSize: 12,
            fontFamily: "Verdana",
            textFill: "#3aa",
            width: 180,
            strokeWidth: 3,
            padding: 4
        })
    );
}
scène.add(trame);

// Balles
for(var i = 0; i < 3; ++i) {
    balles.add(
        new Kinetic.Ellipse({
            x: 100,
            y: i * 100 + 40,
            radius: 20,
            fill: "#c66",
            name: "balle"
        })
    );
}

scène.add(balles);

document.getElementById("animer").addEventListener("click",
function() {
    for(var i = 0; i < 3; ++i) {
        var balle = scène.get(".balle")[i];
        balle.transitionTo({
            x: 400,
            duration: 2,
            easing: texte[i]
        });
    }
}, false);

document.getElementById("reset").addEventListener("click",
function() {
    for(var i = 0; i < 3; ++i) {
        var balle = scène.get(".balle")[i];
        balle.x = 100;
        balle.y = i * 100 + 40;
    }
});
```

```
        balle.setX(100);
        balles.draw();
    },
}, false);
};
```

Tester !



Il y a d'autres éléments intéressants dans ce code comme par exemple le référencement des 3 balles. Lors de leur création, on leur donne le même nom :

#### Code : JavaScript

```
name: "balle"
```

On peut ensuite obtenir un référence de chaque balle en utilisant la méthode `get` appliquée à la scène en passant en paramètre le nom (remarquez la syntaxe qui impose de faire précédé avec un point) :

#### Code : JavaScript

```
var balle = scène.get(".balle") [i];
```

La valeur de retour est un tableau indicé, il est ainsi facile de référencer les balles.

Le librairie prévoit une variation de l'amortissement avec un effet plus tonique :

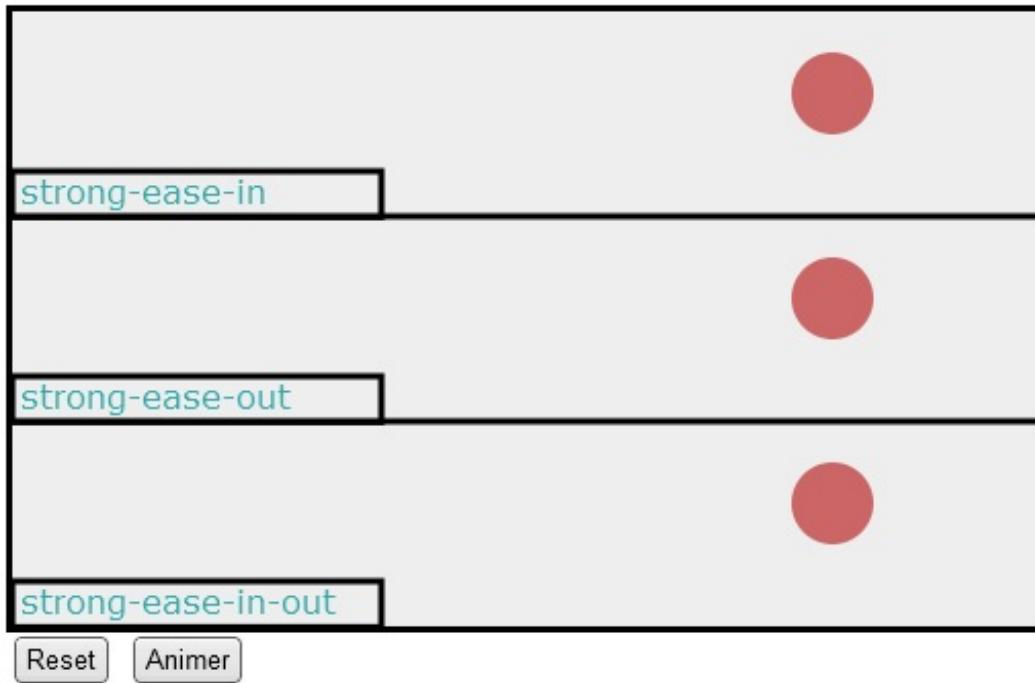
Valeur	effet
<b>strong-ease-in</b>	Amortissement tonique en début de transition
<b>strong-ease-out</b>	Amortissement tonique en fin de transition
<b>strong-ease-in-out</b>	Amortissement tonique en début et fin de transition

Voici le même code que précédemment, mais en utilisant les nouvelles valeurs, on se contente de modifier cette ligne :

**Code : JavaScript**

```
var texte = ["strong-ease-in", "strong-ease-out", "strong-ease-in-out"];
```

Tester !



Une troisième possibilité d'amortissement consiste à obtenir un effet de recul avec ces valeurs :

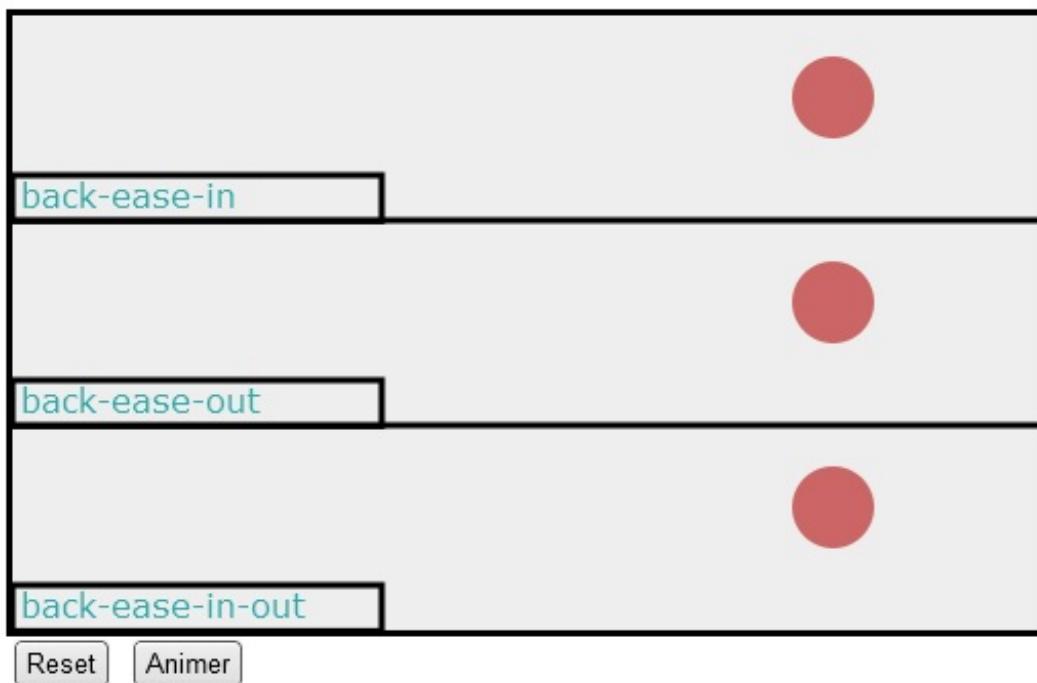
Valeur	effet
<b>back-ease-in</b>	Amortissement avec recul en début de transition
<b>back-ease-out</b>	Amortissement avec dépassement en fin de transition
<b>back-ease-in-out</b>	Amortissement avec recul en début et dépassement en fin de transition

Reprendons le même code avec les nouvelles valeurs, on se contente de modifier cette ligne :

**Code : JavaScript**

```
var texte = ["back-ease-in", "back-ease-out", "back-ease-in-out"];
```

Tester !



### Elastique

Le deuxième effet disponible est un effet élastique. Il se produit une oscillation autour du point de départ, d'arrivée ou les deux :

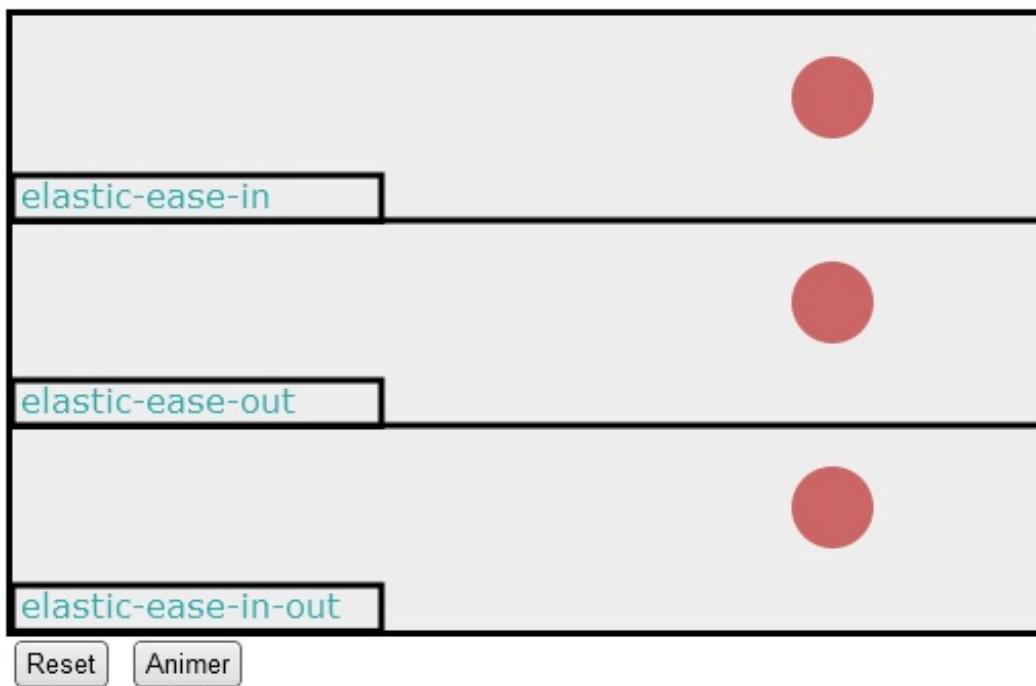
Valeur	effet
<b>elastic-ease-out</b>	Oscillation en début de transition
<b>elastic-ease-in</b>	Oscillation en fin de transition
<b>elastic-ease-in-out</b>	Oscillation en début et fin de transition

Reprendons notre code d'exemple pour tester cet effet :

#### Code : JavaScript

```
var texte = ["elastic-ease-in", "elastic-ease-out", "elastic-ease-in-out"];
```

Tester !



### Rebond

Le dernier effet disponible est un effet de rebond qui se produit sur le point de départ, d'arrivée ou les deux :

Valeur	effet
<b>bounce-ease-in</b>	Rebond en début de transition
<b>bounce-ease-out</b>	Rebond en fin de transition
<b>bounce-ease-in-out</b>	Rebond en début et fin de transition

Reprendons notre code d'exemple pour tester cet effet :

#### Code : JavaScript

```
var texte = ["bounce-ease-in", "bounce-ease-out", "bounce-ease-in-out"];
```

Tester !



## Fonction de rappel

Il est possible de prévoir une fonction de rappel (callback) en fin de transition, ce qui est bien pratique par exemple pour poursuivre un mouvement. Pour illustrer cette possibilité, je vous propose la réalisation d'un simple pendule. Je vais en profiter pour rappeler la possibilité de grouper deux formes. Ici ça sera le fil constitué avec une ligne et la boule créée avec une ellipse. Rien de bien nouveau si ce n'est une autre manière de mettre les éléments en action :

### Code : JavaScript

```
window.onload = function() {
    var sens = -1;

    // Transition avec fonction de rappel
    function transition() {
        sens *= -1;
        pendule.transitionTo({
            rotation: sens * Math.PI / 4,
            duration: 2,
            easing: "ease-in-out",
            callback: function() {transition();}
        });
    }

    // Scène
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

    var calque = new Kinetic.Layer();

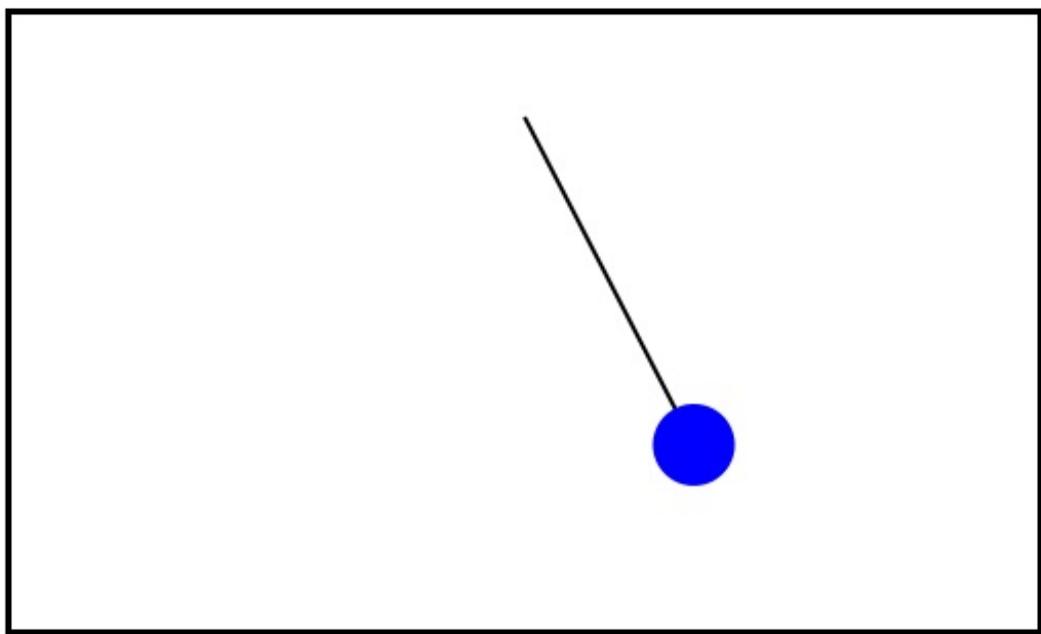
    // Construction du pendule avec un groupe
    var pendule = new Kinetic.Group({
        x: 250,
        y: 50
    });
    var boule = new Kinetic.Ellipse({
        x: 0,
        y: 180,
        radius: 20,
```

```
    fill: "blue"
  );
var fil = new Kinetic.Line({
  points: [0, 0, 0, 180],
  stroke: "black"
});

pendule.add(fil);
pendule.add(boule);
calque.add(pendule);
scène.add(calque);

transition();
};
```

Tester !



Quelle facilité pour créer un mouvement pendulaire ! 😊

## Stopper et reprendre une animation

Une autre possibilité intéressante pour l'interactivité est qu'une transition peut être stoppée et reprise facilement en utilisant les méthodes **stop** et **resume**. On va donc appliquer cette possibilité à notre pendule en prévoyant deux boutons pour agir sur son mouvement :

Code : JavaScript

```
window.onload = function() {
  var sens = -1;
  var trans;

  // Transition avec fonction de rappel
  function transition() {
    sens *= -1;
    trans = pendule.transitionTo({
      rotation: sens * Math.PI / 4,
      duration: 2,
      easing: "ease-in-out",
      callback: function() {transition(); }
    });
  }
}
```

```
// Scène
var scène = new Kinetic.Stage({
  container: "kinetic",
  width: 500,
  height: 300
});

var calque = new Kinetic.Layer();

// Construction du pendule avec un groupe
var pendule = new Kinetic.Group( {
  x: 250,
  y: 50
});
var boule = new Kinetic.Ellipse({
  x: 0,
  y: 180,
  radius: 20,
  fill: "blue"
});
var fil = new Kinetic.Line({
  points: [0, 0, 0, 180],
  stroke: "black"
});

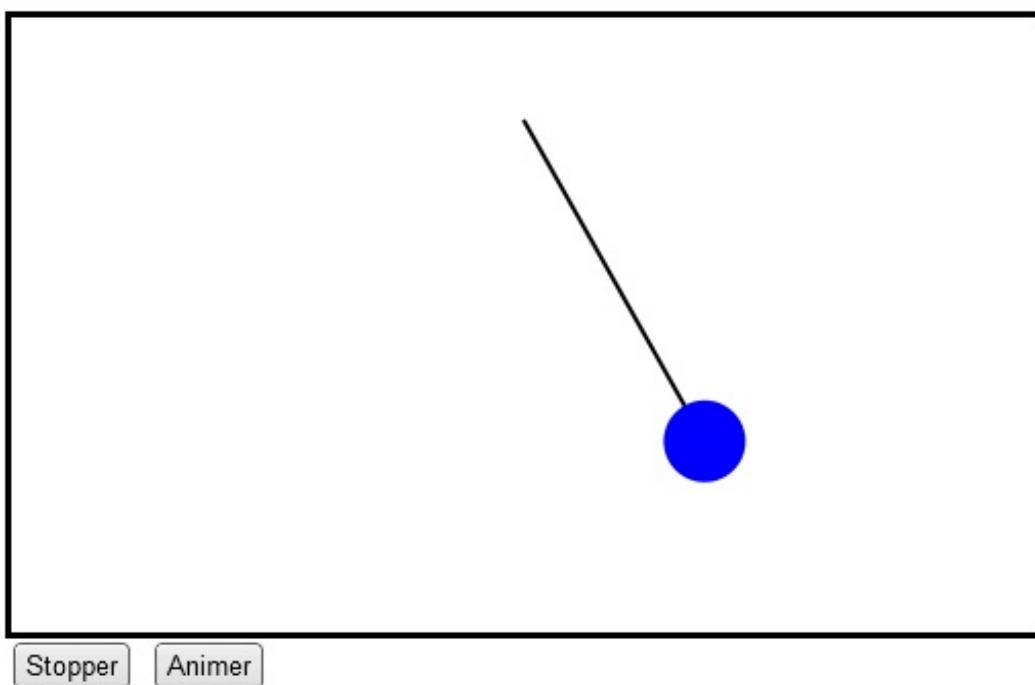
pendule.add(fil);
pendule.add(boule);
calque.add(pendule);
scène.add(calque);

transition();

document.getElementById("animer").addEventListener("click",
function() {
trans.resume();
}, false);

document.getElementById("stopper").addEventListener("click",
function() {
trans.stop();
}, false);
};
```

Tester !

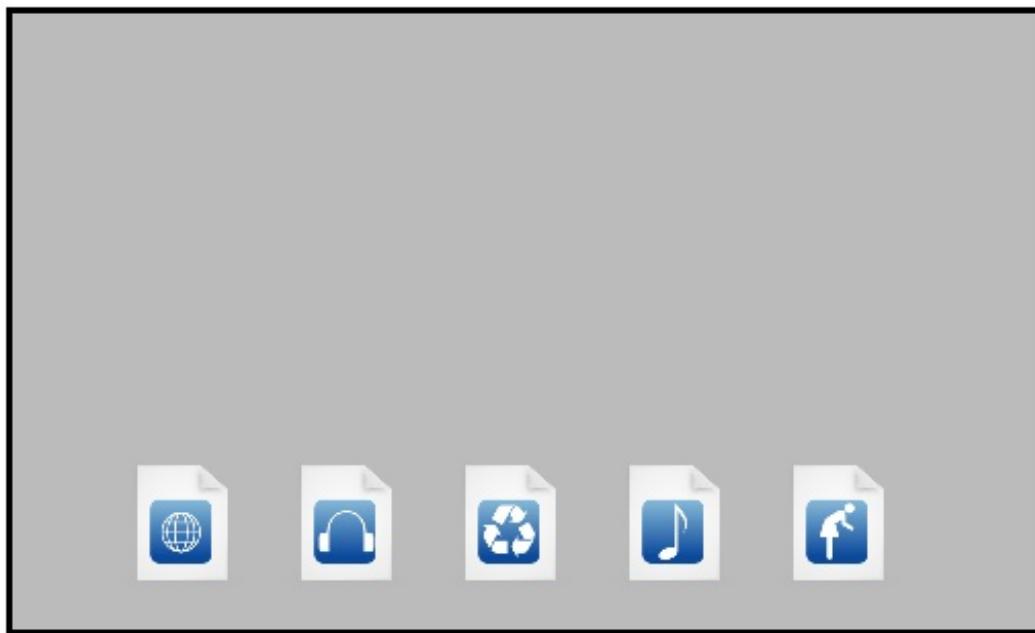


J'ai surligné le code modifié. Pour utiliser les méthodes **stop** et **resume** on a besoin d'avoir une référence de la transition, d'où la création de la variable **trans** pour pointer sur celle-ci.

## TP

Pour récapituler ce que nous avons vu dans ce chapitre, je vous propose de réaliser une animation d'icônes. Plutôt qu'un grand discours, allez voir directement le résultat :

[Tester !](#)



Comme vous le constatez, on affiche 5 icônes et quand le curseur de la souris passe sur l'une d'elles déjà il change d'aspect et en plus l'icône augmente de taille avec un effet esthétique au niveau du mouvement. Une bonne occasion de revoir comment on charge des images par la même occasion.

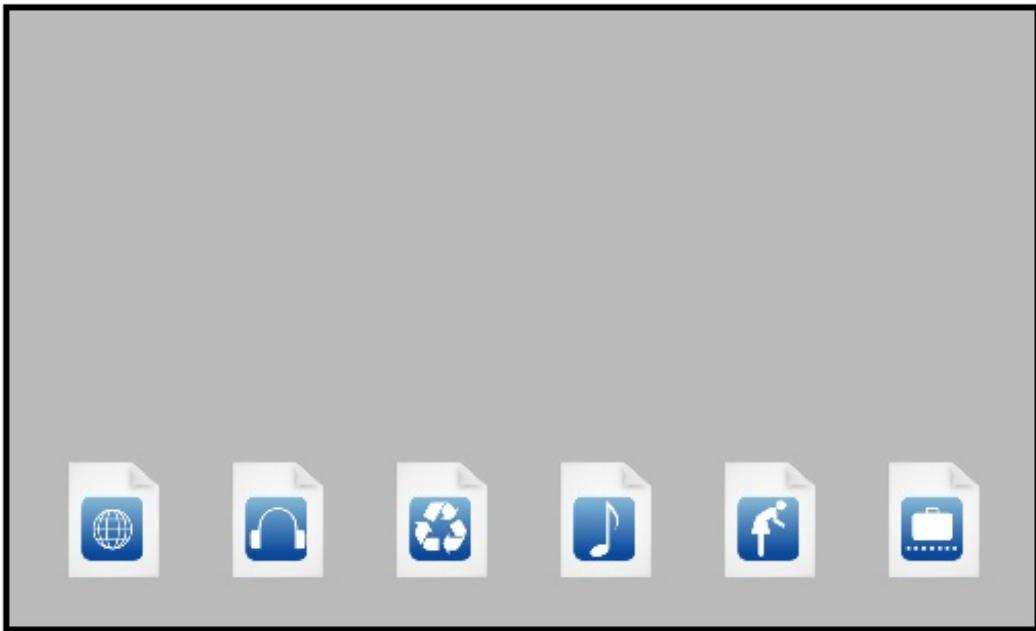
[Secret \(cliquez pour afficher\)](#)

Code : JavaScript

```
// Chargement des images
function load_images(sources, callback) {
    var images = new Array();
    var loadedImages = 0;
    sources.forEach(function(value, index) {
        images[index] = new Image();
        images[index].onload = function() {
            if(++loadedImages >= sources.length) callback(images);
        };
        images[index].src = value;
    });
}
// Initialisation de la scène
function init_scène(images) {
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });
    var fond = new Kinetic.Layer();
    var calque = new Kinetic.Layer();
    var rectangle = new Kinetic.Rect({
        width: 500,
        height: 300,
        fill: "#bbb",
    });
    fond.add(rectangle);
    images.forEach(function(value, index) {
        var image = new Kinetic.Image({
            x: 82 + index * 80,
            y: 252,
            offset: [32, 32],
            image: value,
            width: 64,
            height: 64
        });
        image.on("mouseover", function() {
            document.body.style.cursor = "pointer";
            this.transitionTo({
                scale: {x: 1.6, y: 1.6},
                duration: .4,
                easing: "ease-in-out"
            });
        });
        image.on("mouseout", function() {
            document.body.style.cursor = "default";
            this.transitionTo({
                scale: {x: 1, y: 1},
                duration: .4,
                easing: "ease-in-out"
            });
        });
        calque.add(image);
    });
    scène.add(fond);
    scène.add(calque);
}
// Chargement de la page
window.onload = function() {
    var sources_icônes = [
        "images/icône01.png",
        "images/icône02.png",
        "images/icône03.png",
        "images/icône04.png",
        "images/icône05.png"
    ];
    load_images(sources_icônes, init_scène);
}
```

Je vous propose d'aller un peu plus loin dans cet exemple. Maintenant on va prendre 10 icônes. Donc trop pour pouvoir les afficher en même temps. On va donc créer une barre d'icônes "draggables". Une petite astuce pour vous éviter un codage furieux : on peut rendre tout un calque "draggable" 😊

Tester !



Secret (cliquez pour afficher)

Code : JavaScript

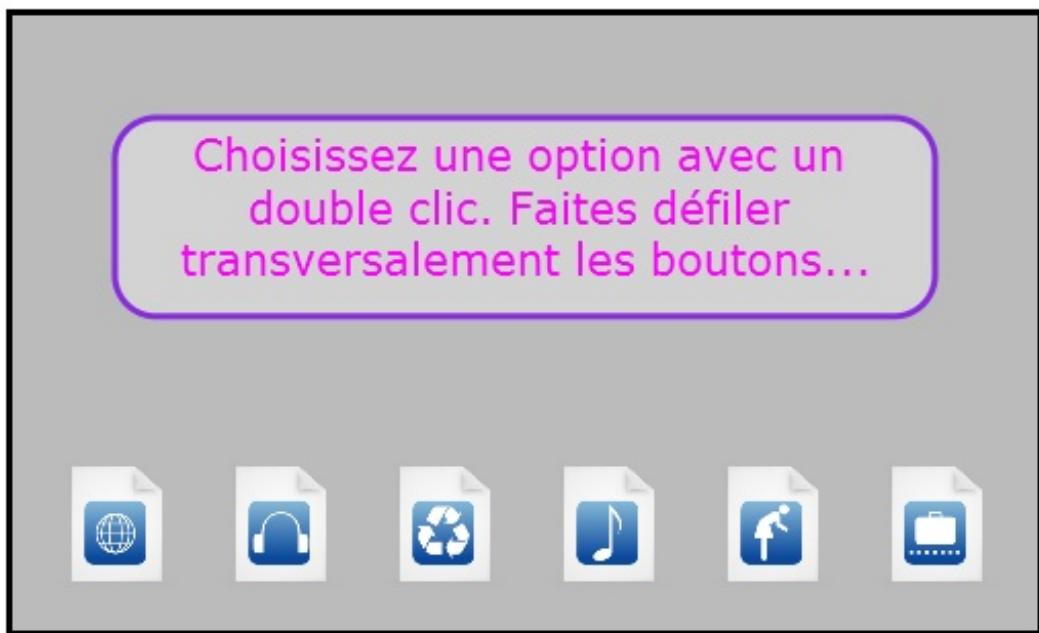
```
// Chargement des images
function load_images(sources, callback) {
  var images = new Array();
  var loadedImages = 0;
  sources.forEach(function(value, index) {
    images[index] = new Image();
    images[index].onload = function() {
      if (++loadedImages >= sources.length) callback(images);
    };
    images[index].src = value;
  });
}
// Initialisation de la scène
function init_scène(images) {
  var scène = new Kinetic.Stage({
    container: "kinetic",
    width: 500,
    height: 300
  );
  var fond = new Kinetic.Layer();
  var calque = new Kinetic.Layer({
    draggable: true,
    dragConstraint: "horizontal",
    dragBounds: {
      left: -320,
      right : 0
    },
    offset:[-250, 0]
  });
}
```

```
var rectangle = new Kinetic.Rect({
    width: 500,
    height: 300,
    fill: "#bbb",
});
fond.add(rectangle)
images.forEach(function(value, index) {
    var image = new Kinetic.Image({
        x: -200 + index * 80,
        y: 252,
        offset: [32, 32],
        image: value,
        width: 64,
        height: 64
    });
    image.on("mouseover",function() {
        document.body.style.cursor = "pointer";
        this.transitionTo({
            scale: {x: 1.6, y: 1.6},
            duration: .4,
            easing: "ease-in-out"
        });
    });
    image.on("mouseout",function() {
        document.body.style.cursor = "default";
        this.transitionTo({
            scale: {x: 1, y: 1},
            duration: .4,
            easing: "ease-in-out"
        });
    });
    calque.add(image);
});
scène.add(fond);
scène.add(calque);

}
// Chargement de la page
window.onload = function() {
    var sources_icônes = [
        "images/icône01.png",
        "images/icône02.png",
        "images/icône03.png",
        "images/icône04.png",
        "images/icône05.png",
        "images/icône06.png",
        "images/icône07.png",
        "images/icône08.png",
        "images/icône09.png",
        "images/icône10.png"
    ];
    load_images(sources_icônes, init_scène);
}
```

Comme je vous sens en forme, on va aller encore plus loin. On va ajouter une zone de texte, un double clic sur une icône doit afficher son index dans la zone de texte :

Tester !



**Secret** (cliquez pour afficher)

**Code : JavaScript**

```
// Chargement des images
function load_images(sources, callback) {
    var images = new Array();
    var loadedImages = 0;
    sources.forEach(function(value, index) {
        images[index] = new Image();
        images[index].onload = function() {
            if(++loadedImages >= sources.length) callback(images);
        };
        images[index].src = value;
    });
}
// Initialisation de la scène
function init_scène(images) {
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });
    var fond = new Kinetic.Layer();
    var calque = new Kinetic.Layer({
        draggable: true,
        dragConstraint: "horizontal",
        dragBounds: {
            left: -320,
            right : 0
        },
        offset:[-250, 0]
    });
    var rectangle = new Kinetic.Rect({
        width: 500,
        height: 300,
        fill: "#bbb",
    });
    var text = new Kinetic.Text({
        x: 50,
        y: 50,
        text: "Choisissez une option avec un double clic. Faites défiler transversalement les boutons...",
    });
    fond.add(rectangle);
    fond.add(text);
    scène.add(fond);
    scène.add(calque);
}
```

```
fontSize: 16,
fontFamily: "Verdana",
textFill: "magenta",
width: 400,
align: "center",
stroke: "blueviolet",
strokeWidth: 3,
padding: 10,
cornerRadius: 20,
fill: "lightgrey",
lineHeight: 1.6
});
fond.add(rectangle)
fond.add(text)
images.forEach(function(value, index) {
  var image = new Kinetic.Image({
    x: -200 + index * 80,
    y: 252,
    offset: [32, 32],
    image: value,
    width: 64,
    height: 64,
    id: index
  });
  image.on("mouseover",function() {
    document.body.style.cursor = "pointer";
    this.transitionTo({
      scale: {x: 1.6, y: 1.6},
      duration: .4,
      easing: "ease-in-out"
    });
  });
  image.on("mouseout",function() {
    document.body.style.cursor = "default";
    this.transitionTo({
      scale: {x: 1, y: 1},
      duration: .4,
      easing: "ease-in-out"
    });
  });
  image.on("dblclick",function() {
    text.setText("Vous avez sélectionné le bouton avec le numéro
d'index " + this.getId());
    fond.draw();
  });
  calque.add(image);
});
scène.add(fond);
scène.add(calque);

}
// Chargement de la page
window.onload = function() {
  var sources_icônes = [
    "images/icône01.png",
    "images/icône02.png",
    "images/icône03.png",
    "images/icône04.png",
    "images/icône05.png",
    "images/icône06.png",
    "images/icône07.png",
    "images/icône08.png",
    "images/icône09.png",
    "images/icône10.png"
  ];
  load_images(sources_icônes, init_scène);
}
```

## Les sprites

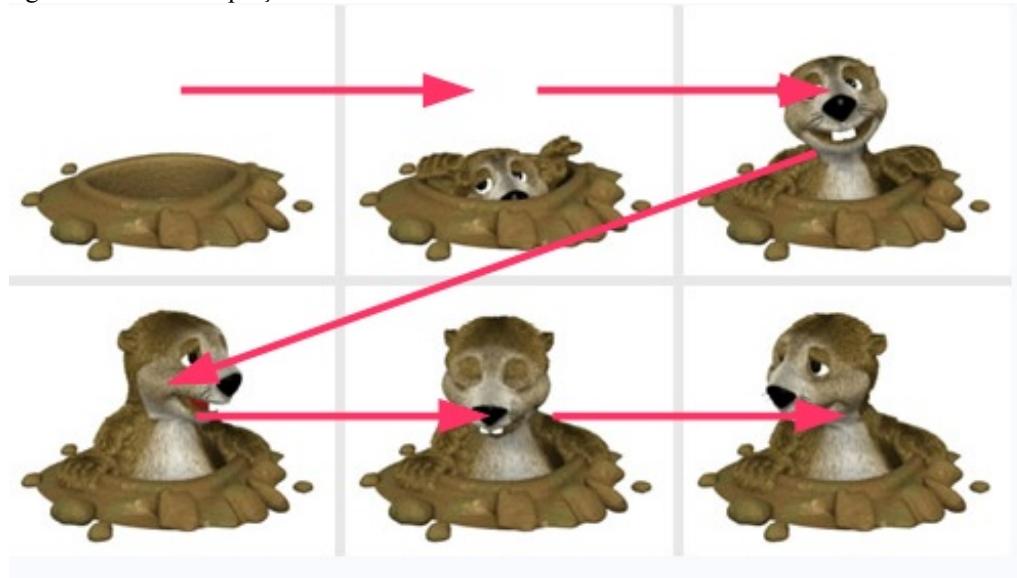
Les sprites sont des images d'une animation regroupées sur une seule image.

## Le principe des sprites

Voici le sprite qui va nous servir pour les exemples (recueilli sur [ce blog](#)) :



Il est composé de 6 images, chacune étant une étape de l'animation. La lecture normale des images se fait en partant de la première en haut à gauche et en se déplaçant ensuite vers la droite et ainsi de suite :



## Animation d'un sprite

C'est la classe **Sprite** qui nous permet d'animer des sprites. Son constructeur attend un positionnement, une image, des animations et éventuellement une vitesse. Voici le code pour mettre en mouvement la petite taupe :

### Code : JavaScript

```
window.onload = function() {  
    var scène = new Kinetic.Stage({  
        container: "kinetic",  
        width: 500,  
        height: 300  
    });
```

```
var calque = new Kinetic.Layer();

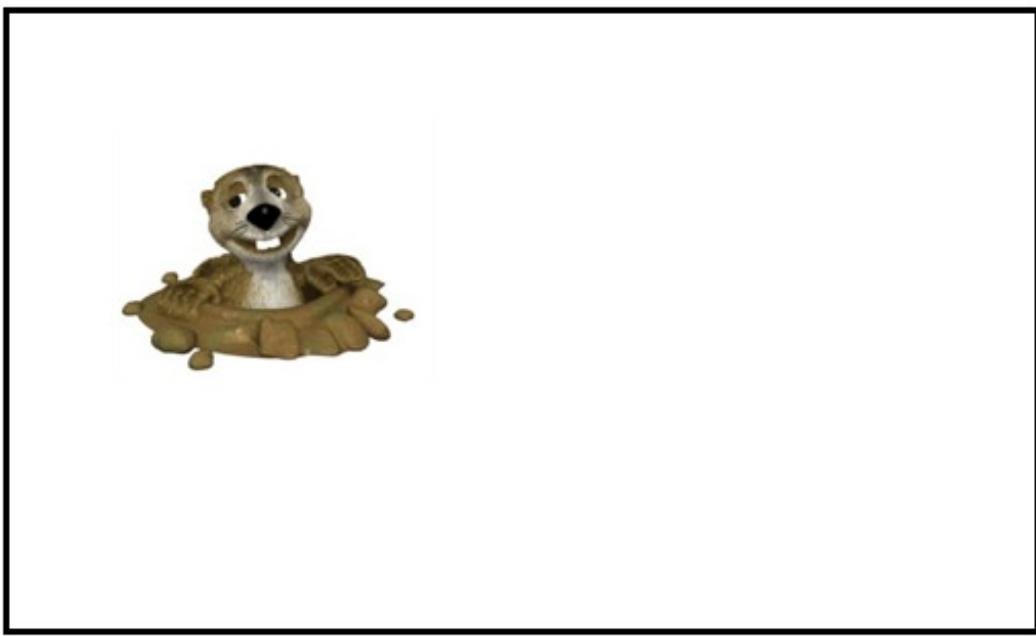
var animations = {
    sortie: [
        {x: 10, y: 8, width: 158, height: 130},
        {x: 175, y: 8, width: 158, height: 130},
        {x: 340, y: 8, width: 158, height: 130},
        {x: 340, y: 145, width: 158, height: 130},
        {x: 175, y: 145, width: 158, height: 130},
        {x: 10, y: 145, width: 158, height: 130}
    ]
};

var imageObj = new Image();
imageObj.onload = function() {
    var animal = new Kinetic.Sprite({
        x: 50, y: 50,
        image: imageObj,
        animation: "sortie",
        animations: animations,
        frameRate: 4
    });

    calque.add(animal);
    scene.add(calque);
    animal.start();
};

imageObj.src = "images/gopher.jpg";
};
```

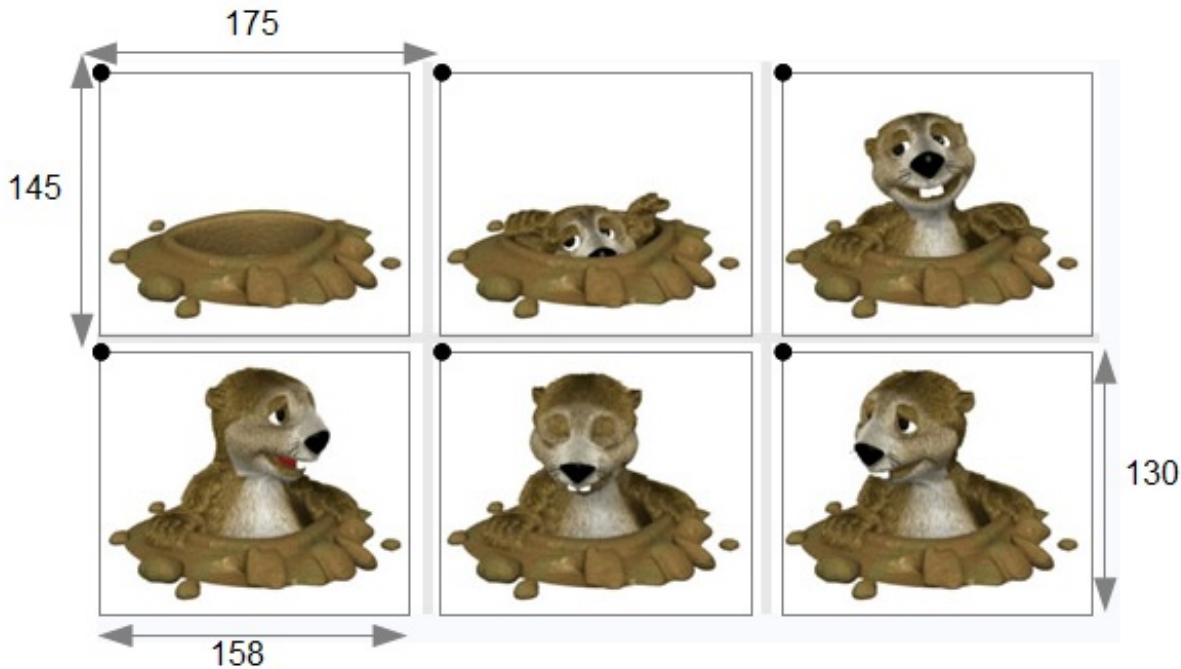
Tester !



Regardons la partie animation. On crée un objet **animations** qui contient une seule propriété (sortie) qui est en fait le nom d'une animation. Celle-ci est composée d'un tableau d'objets dont chacun est constitué de 4 propriétés :

Propriété	Fonction
<b>x</b>	Position sur l'axe X du point de référence (coin haut gauche)
<b>y</b>	Position sur l'axe Y du point de référence (coin haut gauche)
<b>width</b>	Largeur en pixels de l'image élémentaire
<b>height</b>	Hauteur en pixels de l'image élémentaire

Vous avez compris qu'il faut donc connaître l'organisation de l'image. En particulier il faut absolument que les images élémentaires soient uniformément réparties pour éviter des prises de tête sur la détermination des valeurs. En voici quelques-unes reportées sur le sprite :



C'est une méthode efficace, mais un peu laborieuse. On se rend compte de la répétition des largeurs et hauteurs et aussi de la répartition harmonieuse. On peut donc améliorer notre script en évitant toutes ces répétitions aussi pénibles qu'inesthétiques.

## Où on améliore notre script

L'amélioration consiste essentiellement à optimiser la création du tableau des animations :

### Code : JavaScript

```
window.onload = function() {
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

    var calque = new Kinetic.Layer();

    // Génération des frames
    function frames(base_x, base_y, offset_x, offset_y, nbr_x, nbr_y,
width, height) {
        var frames tab = new Array();
        for(var i = 0; i < nbr_x * nbr_y; ++i) {
            frames tab.push({
                x: base_x + offset_x * ((i + nbr_x) % nbr_x),
                y: base_y + offset_y * parseInt(i / nbr_x),
                width: width,
                height: height
            });
        }
        return frames_tab;
    }

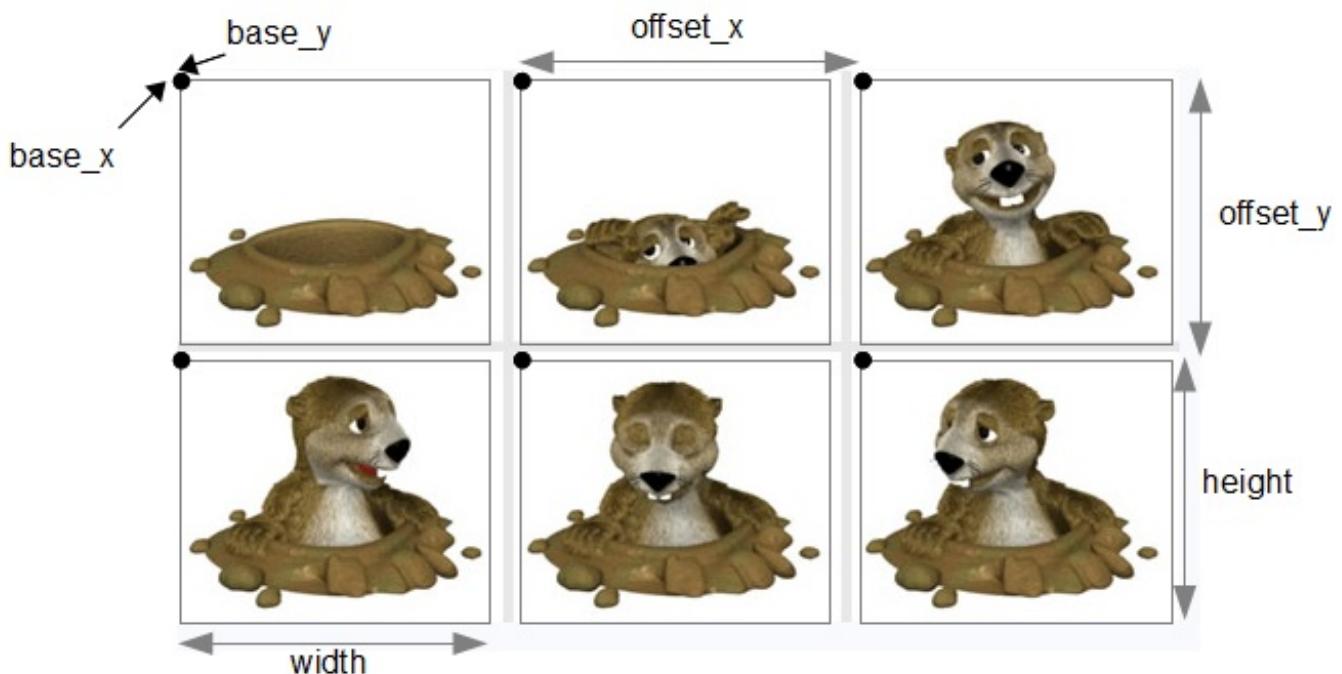
    // Animations
    var animations = {
        sortie: frames(10, 8, 165, 137, 3, 2, 158, 130)
    };

    // Image
    var imageObj = new Image();
    imageObj.onload = function() {
        var animal = new Kinetic.Sprite({
            x: 50,
            y: 50,
            image: imageObj,
            animation: "sortie",
            animations: animations,
            frameRate: 4
        });

        calque.add(animal);
        scène.add(calque);
        animal.start();
    };
    imageObj.src = "images/gopher.jpg";
};
```

Tester !

Une fonction **frames** est chargée de faire le calcul du positionnement des images élémentaires. Elle attend 6 paramètres dont voici une visualisation :



C'est déjà moins prise de tête pour constituer une animation 😊.

## Plusieurs animations

Jusque-là, nous nous sommes contentés de créer une seule animation avec toutes les images prévues dans l'ordre. On pourrait aussi avoir envie de créer des animations avec certaines images à chaque fois, et pas forcément dans l'ordre. Notre code alors ne convient plus et il nous faut le rendre plus générique. Voici un exemple avec deux animations déclenchées par des boutons :

### Code : JavaScript

```

window.onload = function() {
  var scène = new Kinetic.Stage({
    container: "kinetic",
    width: 500,
    height: 300
  });

  var calque = new Kinetic.Layer();

  // Génération d'une frame
  function frame(id, base_x, base_y, offset_x, offset_y, nbr_x,
  width, height) {
    return {
      x: base_x + offset_x * ((id + nbr_x) % nbr_x),
      y: base_y + offset_y * parseInt(id / nbr_x),
      width: width,
      height: height
    };
  }

  // Génération des frames
  function frames(ids, base_x, base_y, offset_x, offset_y, nbr_x,
  nbr_y, width, height) {
    var frames_tab = new Array();
    for(var i = 0; i < ids.length; ++i)
      frames_tab.push(frame(ids[i], base_x, base_y, offset_x, offset_y,
      nbr_x, width, height));
    return frames_tab;
  }

  // Animations
  var animations = {
    ...
  };
}

```

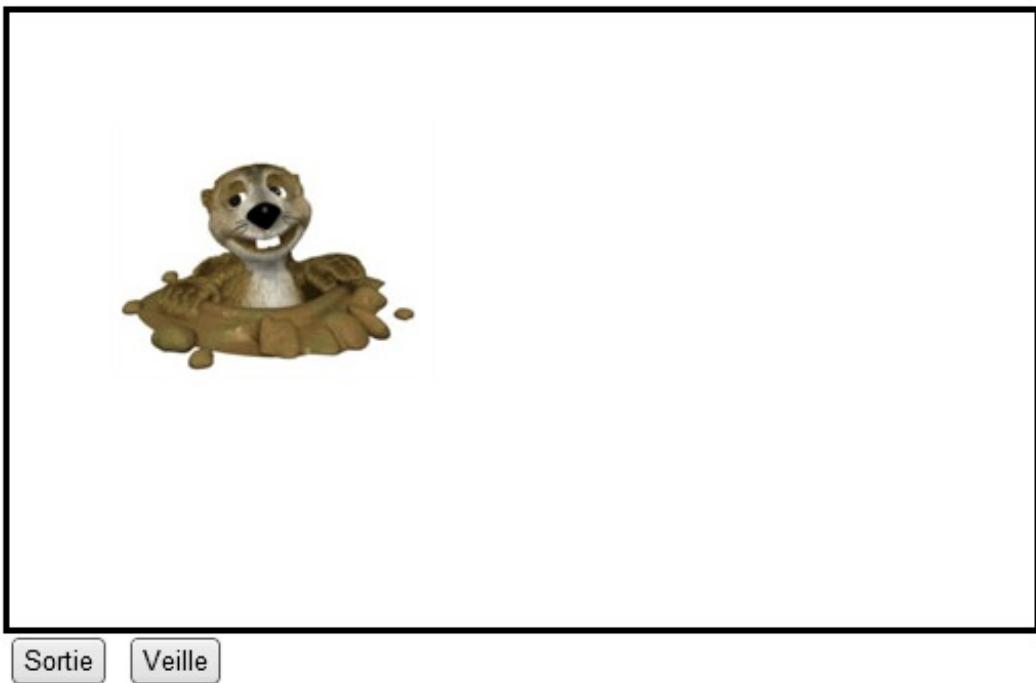
```
sortie: frames([0,1,2], 10, 8, 165, 137, 3, 2, 158, 130),
veille: frames([3,4,5,4], 10, 8, 165, 137, 3, 2, 158, 130)
};

// Image
var imageObj = new Image();
imageObj.onload = function() {
    var animal = new Kinetic.Sprite({
        x: 50,
        y: 50,
        image: imageObj,
        animation: "sortie",
        animations: animations,
        frameRate: 4
    });

    calque.add(animal);
    scène.add(calque);
    // Sortie initiale
    animal.afterFrame(2, function() {
        animal.stop();
    });
    animal.start();

    // Commandes
    document.getElementById("sortie").addEventListener("click",
function() {
    if(animal.getAnimation() != "sortie") {
        animal.stop();
        animal.setAnimation("sortie");
        animal.start();
        animal.afterFrame(2, function() {
            animal.stop();
        });
    }
}, false);
    document.getElementById("veille").addEventListener("click",
function() {
    if(animal.getAnimation() != "veille") {
        animal.stop();
        animal.setAnimation("veille");
        animal.start();
        animal.afterFrame(5, function() {
            animal.stop();
        });
    }
}, false);
};
imageObj.src = "images/gopher.jpg";
};
```

Tester !



Le code devient un peu plus lourd. Remarquez déjà les deux méthodes **start** et **stop** pour démarrer et arrêter l'animation. Remarquez également l'événement **afterFrame** qui permet d'exécuter du code après une certaine image. J'ai utilisé également la méthode **setAnimation** qui affecte l'animation en cours avec son nom.

Mais c'est surtout la fonction **frames** qui a été chamboulée. Maintenant elle attend non seulement les paramètres de positionnement et dimension des images, mais également un tableau des index des images composant l'animation dans l'ordre. Du coup l'écriture du code des animations devient un jeu d'enfant :

#### Code : JavaScript

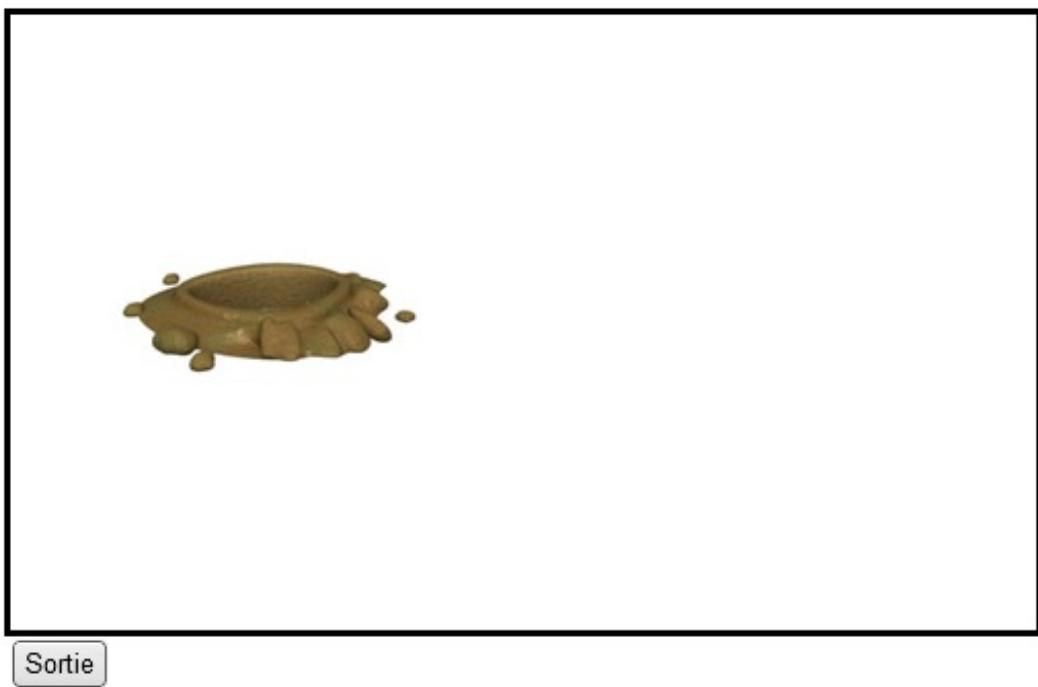
```
var animations = {
    sortie: frames([0,1,2], 10, 8, 165, 137, 3, 2, 158, 130),
    veille: frames([3,4,5,4], 10, 8, 165, 137, 3, 2, 158, 130)
};
```

On se contente d'envoyer un tableau avec les index des images que l'on veut 😊.

## TP

Maintenant vous êtes prêts à faire un petit TP dans la lignée des scripts précédents. Cette fois on désire 3 animations : une pour la sortie de la taupe, une pour la faire regarder autour d'elle, et enfin une dernière pour qu'elle retourne dans son trou. On utilise trois boutons qui ne doivent apparaître que lorsqu'ils sont utiles. Voici l'exemple traité qui sera beaucoup mieux que toutes mes explications :

Tester !



Au départ la taupe peut seulement sortir, donc seulement le bouton "Sortie" doit apparaître. Lorsqu'elle est sortie, elle peut alors soit surveiller, soit rentrer, donc les deux autres boutons doivent être affichés. Lorsqu'elle surveille, elle ne peut plus que rentrer... Au niveau du code il vous faut bien gérer l'événement **afterFrame** pour éviter les surprises.

#### Secret (cliquez pour afficher)

##### Code : JavaScript

```
window.onload = function() {
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

    var calque = new Kinetic.Layer();

    // Génération d'une frame
    function frame(id, base_x, base_y, offset_x, offset_y, nbr_x,
width, height) {
        return {
            x: base_x + offset_x * ((id + nbr_x) % nbr_x),
            y: base_y + offset_y * parseInt(id / nbr_x),
            width: width,
            height: height
        };
    }

    // Génération des frames
    function frames(ids, base_x, base_y, offset_x, offset_y, nbr_x,
nbr_y, width, height) {
        var frames_tab = new Array();
        for(var i = 0; i < ids.length; ++i)
            frames_tab.push(frame(ids[i], base_x, base_y, offset_x,
offset_y, nbr_x, width, height));
        return frames_tab;
    }

    // Animations
    var animations = {
        sortie: frames([0,1,2], 10, 8, 165, 137, 3, 2, 158, 130),
        veille: frames([3,4,5,4], 10, 8, 165, 137, 3, 2, 158, 130),
    }
}
```

```
entrée: frames([2,1,0], 10, 8, 165, 137, 3, 2, 158, 130)
};

// Image
var imageObj = new Image();
imageObj.onload = function() {
    var animal = new Kinetic.Sprite({
        x: 50,
        y: 50,
        image: imageObj,
        animation: "sortie",
        animations: animations,
        frameRate: 4
    });

    calque.add(animal);
    scène.add(calque);

    // Commandes
    document.getElementById("sortie").addEventListener("click",
    function() {
        set_animation("sortie", 2, "inline", "none", "inline");
    }, false);
    document.getElementById("veille").addEventListener("click",
    function() {
        set_animation("veille", 5, "none", "none", "inline");
    }, false);
    document.getElementById("entrée").addEventListener("click",
    function() {
        set_animation("entrée", 2, "none", "inline", "none");
    }, false);

    // Gestion de l'animation et des boutons
    function set_animation(animation, index, veille, sortie, entrée)
    {
        animal.stop();
        animal.setAnimation(animation);
        animal.start();
        animal.afterFrame(index, function() {animal.stop();});
        document.getElementById("veille").style.display=veille;
        document.getElementById("entrée").style.display=entrée;
        document.getElementById("sortie").style.display=sortie;
    }
};
imageObj.src = "images/gopher.jpg";
};
```

## Les animations

Nous allons pour terminer aborder les animations libres. C'est sans doute l'aspect le plus créatif de cette bibliothèque, mais en même temps elle va peu nous aider, parce qu'il faut écrire le code pour l'animation, mais elle nous offre une infrastructure performante.

## Quand on retrouve les transitions

Chaque fois qu'il est possible d'utiliser les transitions, il est évidemment judicieux de les utiliser plutôt que d'écrire le code du calcul du déplacement. Voici un exemple élémentaire avec une balle qui se déplace de façon aléatoire :

### Code : JavaScript

```
window.onload = function() {
    function animation() {
        balle.transitionTo({
            x: Math.random() * 460 + 20,
            y: Math.random() * 260 + 20,
            duration: Math.random() * 2,
```

```
callback: function() {
    animation();
}
};

var scène = new Kinetic.Stage({
    container: "kinetic",
    width: 500,
    height: 300
});

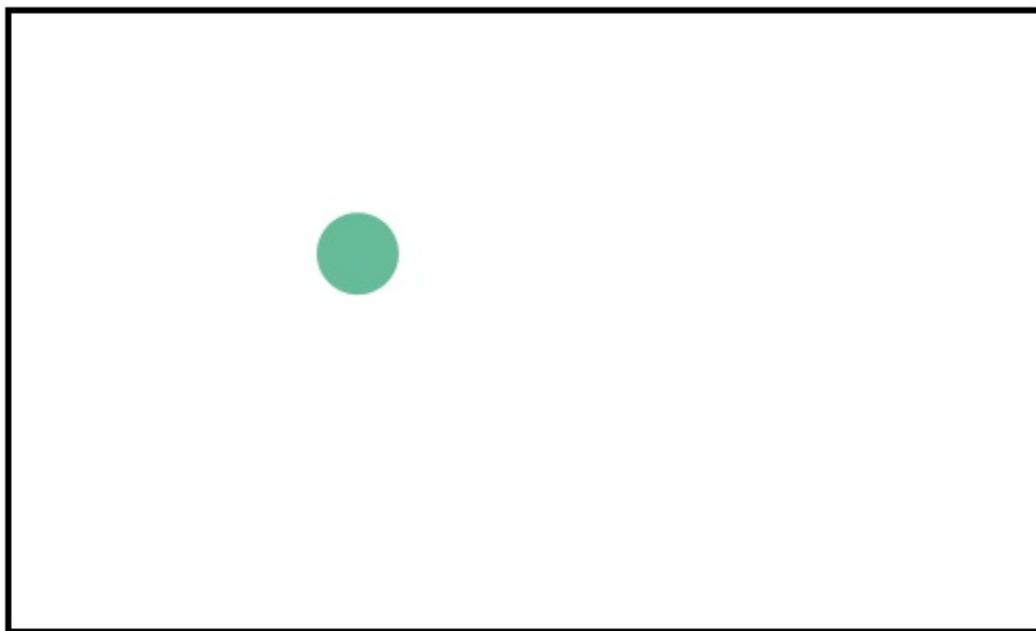
var calque = new Kinetic.Layer();

var balle = new Kinetic.Ellipse({
    x: 100,
    y: 100,
    radius: 20,
    fill: "#6b9"
});

calque.add(balle);
scène.add(calque);

animation();
};
```

Tester !



Les positions x et y ainsi que la durée sont générées de façon aléatoire et une fonction de rappel commence une nouvelle transition.

Cette façon de procéder est efficace, mais elle est évidemment assez rigide et devient vite obsolète pour des animations plus spécifiques.

## Déplacement sur une ellipse

Dans cet exemple on déplace une balle le long d'une ellipse :

Code : JavaScript

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

```
var scène = new Kinetic.Stage({
  container: "kinetic",
  width: 500,
  height: 300
});

var calque = new Kinetic.Layer();
var fond = new Kinetic.Layer();

var a = 150;
var b = 80;
var centre_x = scène.getWidth() / 2;
var centre_y = scène.getHeight() / 2;

var balle = new Kinetic.Ellipse({
  radius: 10,
  fill: "red"
});

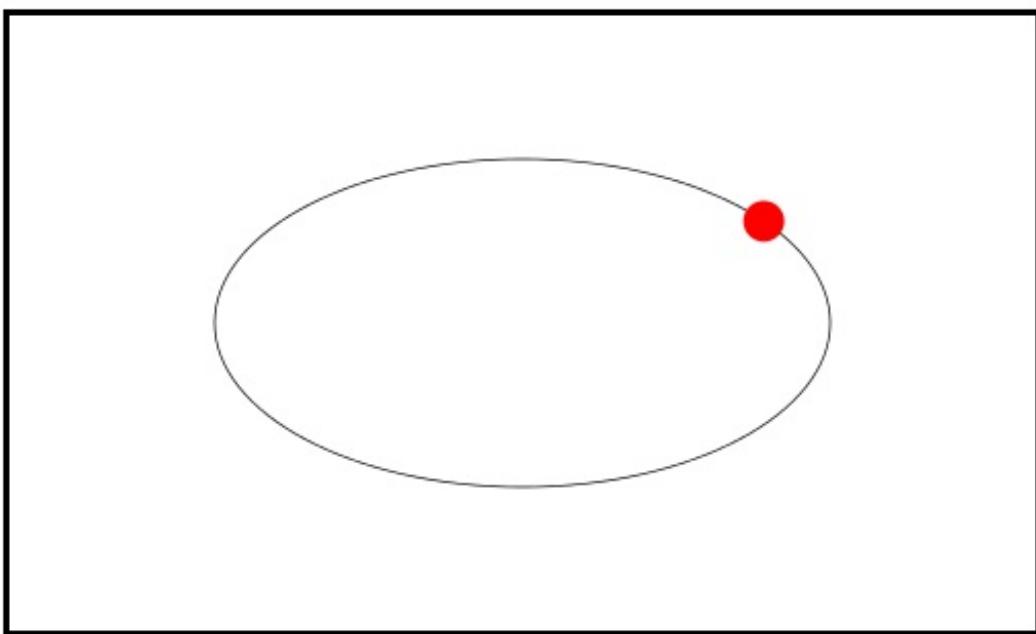
var ellipse = new Kinetic.Ellipse({
  x: centre_x,
  y: centre_y,
  radius: [a, b],
  stroke: "black",
  strokeWidth: .7
});

calque.add(balle);
fond.add(ellipse);
scène.add(fond);
scène.add(calque);

var animation = new Kinetic.Animation({
  func: function(frame) {
    var x = a * Math.cos(frame.time / 2000);
    var y = b * Math.sin (frame.time / 2000);
    balle.setX(centre_x + x);
    balle.setY(centre_y + y);
  },
  node: calque
});

animation.start();
};
```

Tester !



Nous allons nous intéresser à la vitesse de déplacement de cette balle. Vous remarquez dans le code l'utilisation de la propriété **time** du paramètre **frame**. Celui-ci nous donne le temps écoulé depuis le début de l'animation en millisecondes. Il existe une deuxième propriété qui nous donne le temps écoulé depuis le dernier rafraîchissement. Voici ces deux paramètres :

Paramètre	Valeur
<b>time</b>	Temps écoulé depuis le début de l'animation en millisecondes
<b>timeDiff</b>	Temps écoulé depuis le dernier rafraîchissement en millisecondes

Ce sont ces propriétés qui vont nous permettre de gérer efficacement le temps. Voyons cela avec notre balle, voici le code de calcul de la position :

#### Code : JavaScript

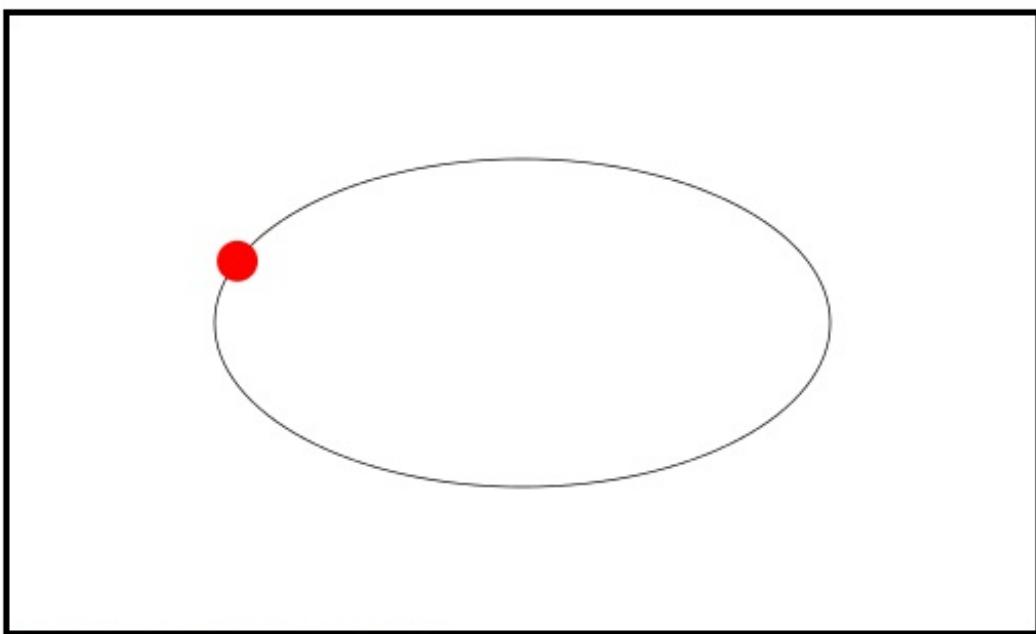
```
var x = a * Math.cos(frame.time / 2000);
var y = b * Math.sin (frame.time / 2000);
balle.setX(centre_x + x);
balle.setY(centre_y + y);
```

On sait que la fonction cosinus de **JavaScript** attend une valeur en radians. Autrement dit un tour complet est effectué quand on passe d'une valeur de 0 à  $2 \cdot \pi$ , autrement dit de 0 à 6,28. Comme je divise le temps par 2000 j'ai une progression de 0,5 à chaque seconde, il me faut donc  $6,28 / 0,5 = 12,56$  secondes pour effectuer un tour complet. A condition évidemment que le rafraîchissement s'effectue à une vitesse suffisante 😊. On va d'ailleurs vérifier cela en ajoutant un contrôle du temps :

#### Code : JavaScript

```
func: function(frame) {
  document.getElementById("texte").innerHTML = "Temps écoulé en
secondes : " + frame.time / 1000
+ "<br>fps : " + parseInt(1000 / frame.timeDiff);
  var x = a * Math.cos(frame.time / 2000);
  var y = b * Math.sin (frame.time / 2000);
  balle.setX(centre_x + x);
  balle.setY(centre_y + y);
},
```

Tester !



Temps écoulé en secondes : 57.322

fps : 30

J'ai ajouté un <div> pour afficher le temps écoulé et le fps (frames par secondes). Vous pouvez ainsi vérifier si je ne vous ai pas menti pour la durée de révolution 😊.

## Mettre en mouvement plusieurs formes

### *Un cas simple*

Nous avons fait déplacer une balle sur une ellipse, ajoutons maintenant un axe qui s'adapte automatiquement au mouvement :

#### Code : JavaScript

```
window.onload = function() {
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

    var calque = new Kinetic.Layer();
    var fond = new Kinetic.Layer();

    var a = 150;
    var b = 80;
    var centre_x = scène.getWidth() / 2;
    var centre_y = scène.getHeight() / 2;

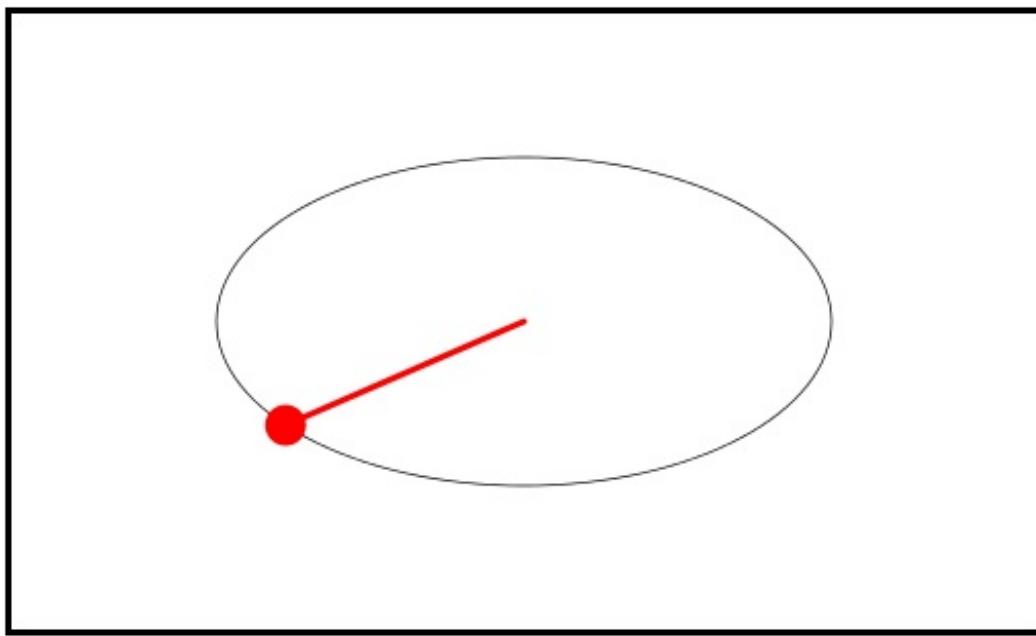
    var axe = new Kinetic.Line({
        points: [centre_x, centre_y],
        stroke: "red",
        strokeWidth: 3,
        lineCap: "round"
    });

    var balle = new Kinetic.Ellipse({
        radius: 10,
        fill: "red"
    });

    var ellipse = new Kinetic.Ellipse({
        x: centre_x,
```

```
y: centre_y,  
radius: [a, b],  
stroke: "black",  
strokeWidth: .7  
});  
  
calque.add(balle);  
calque.add(axe);  
fond.add(ellipse);  
scène.add(fond);  
scène.add(calque);  
  
var animation = new Kinetic.Animation({  
  func: function(frame) {  
    var x = a * Math.cos(frame.time / 2000);  
    var y = b * Math.sin (frame.time / 2000);  
    balle.setX(centre_x + x);  
    balle.setY(centre_y + y);  
    axe.setPoints([centre_x, centre_y, centre_x + x, centre_y + y]);  
  },  
  node: calque  
});  
animation.start();  
};
```

Tester !



Rien de bien compliqué là-dedans, on gère les deux formes.

### *Un cas un peu plus délicat*

Maintenant nous allons faire tourner des images le long de l'ellipse en créant une impression de profondeur en jouant sur la dimension et la transparence des images :

#### Code : JavaScript

```
// Chargement des images  
function load_images(sources, callback) {  
  var images = new Array();  
  var loadedImages = 0;  
  sources.forEach(function(value, index) {  
    images[index] = new Image();  
    images[index].src = value;  
    images[index].onload = function() {  
      loadedImages++;  
      if (loadedImages == sources.length) {  
        callback();  
      }  
    };  
  });  
}
```

```
images[index].onload = function() {
  if(++loadedImages >= sources.length) callback(images);
};

images[index].src = "images/" + value;
});

}

// Initialisation de la scène
function init_scène(images) {
  var scène = new Kinetic.Stage({
    container: "kinetic",
    width: 500,
    height: 300
  });
  var fond = new Kinetic.Layer();
  var calque = new Kinetic.Layer();
  var rectangle = new Kinetic.Rect({
    width: 500,
    height: 300,
    fill: "#bbb",
  });
  fond.add(rectangle);
  images.forEach(function(value, index) {
    var image = new Kinetic.Image({
      x: 82 + index * 80,
      y: 252,
      offset: [32, 32],
      image: value,
      width: 64,
      height: 64,
      name: "image"
    });
    calque.add(image);
  });
  scène.add(fond);
  scène.add(calque);

  var a = 170;
  var b = 80;
  var centre_x = scène.getWidth() / 2;
  var centre_y = scène.getHeight() / 2;

  var animation = new Kinetic.Animation({
    func: function(frame) {
      var images = scène.get(".image");
      var angle = frame.time / 2000;
      var offset = Math.PI * 2 / images.length;
      for(var i = 0; i < images.length; ++i) {
        var img = images[i];
        var x = a * Math.cos(angle + i * offset);
        var y = b * Math.sin (angle + i * offset);
        var posX = centre_x + x;
        var posY = centre_y + y;
        img.setX(posX);
        img.setY(posY);
        var scale = .5 + (posY - 80) / 160;
        img.setScale(scale);
        img.setOpacity(scale - .2);
      }
    },
    node: calque
  });
  animation.start()
}

// Chargement de la page
window.onload = function() {
  var sources_icônes = [
    "img01.jpg",
    "img02.jpg",
    "img03.jpg",
  ];
}
```

```
"img04.jpg",
"img05.jpg",
"img06.jpg",
"img07.jpg",
"img08.jpg",
"img09.jpg",
"img10.jpg"
];
load_images(sources_icônes, init_scène);
}
```

Tester !



La seule astuce du code consiste à calculer un offset d'angle pour répartir les images de façon équilibrée sur toute la longueur de l'ellipse. J'ai aussi nommé les images pour pouvoir facilement les référencer par la suite. Comme il n'y a pas d'autres formes sur ce calque, j'aurais aussi pu utiliser la méthode `getChildren` et ne plus avoir à nommer les images :

#### Code : JavaScript

```
func: function(frame) {
  var images = calque.getChildren();
  var angle = frame.time / 2000;
  var offset = Math.PI * 2 / images.length;
  images.forEach(function(value, index) {
    var img = value;
    var x = a * Math.cos(angle + index * offset);
    var y = b * Math.sin (angle + index * offset);
    var posX = centre_x + x;
    var posY = centre_y + y;
    img.setX(posX);
    img.setY(posY);
    var scale = .5 + (posY - 80) / 160;
    img.setScale(scale);
    img.setOpacity(scale - .2);
  });
},
```

Tester !

## Modifier l'ordre de dessin des formes

Les formes se dessinent dans l'ordre du script, ce qui est en général satisfaisant. Mais lors d'une animation, il peut s'avérer judicieux de modifier cet ordre pour changer la superposition des formes. **KineticJS** prévoit cette possibilité avec quelques méthodes :

Méthode	Effet
<b>moveToTop()</b>	Passe au premier plan
<b>moveToBottom()</b>	Passe au dernier plan
<b>moveUp()</b>	Monte sur le plan au-dessus
<b>moveDown()</b>	Descend au plan inférieur
<b>setZIndex(x)</b>	Passe sur le plan x

Pour ceux qui ont besoin de visualiser ça :



Pour illustrer cette possibilité sur un cas pratique nous allons poursuivre l'exemple des photos tournoyantes en donnant la possibilité de cliquer sur une d'elles pour l'envoyer élégamment au premier plan grâce à la méthode **moveToTop** et une transition élastique pour centrer la photo et changer sa taille, on arrête aussi l'animation le temps de la transition :

### Code : JavaScript

```
// Chargement des images
function load_images(sources, callback) {
    var images = new Array();
    var loadedImages = 0;
    sources.forEach(function(value, index) {
        images[index] = new Image();
        images[index].onload = function() {
            if(++loadedImages >= sources.length) callback(images);
        };
        images[index].src = "images/" + value;
    });
}
// Initialisation de la scène
function init_scène(images) {
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });
    var fond = new Kinetic.Layer();
    var calque = new Kinetic.Layer();
    var rectangle = new Kinetic.Rect({
        width: 500,
        height: 300,
        fill: "#bbb",
    });
    fond.add(rectangle);
    images.forEach(function(value, index) {
        var image = new Kinetic.Image({
            x: 82 + index * 80,
            y: 252,
            offset: [32, 32],
        });
        calque.add(image);
    });
}
```

```
        image: value,
        width: 64,
        height: 64,
        name: "image"
    );
    image.on("click", function() {
        animation.stop();
        with(this) {
            setScale(5);
            moveToTop();
            setOpacity(1);
            transitionTo({
                x: centre_x,
                y: centre_y,
                duration: 2,
                easing: 'elastic-ease-out',
                callback: function() {
                    animation.start();
                }
            });
        }
    });
    calque.add(image);
});
scène.add(fond);
scène.add(calque);

var a = 170;
var b = 80;
var centre_x = scène.getWidth() / 2;
var centre_y = scène.getHeight() / 2;

var animation = new Kinetic.Animation({
    func: function(frame) {
        var images = scène.get(".image");
        var angle = frame.time / 2000;
        var offset = Math.PI * 2 / images.length;
        images.forEach(function(value, index) {
            var img = value;
            var x = a * Math.cos(angle + index * offset);
            var y = b * Math.sin (angle + index * offset);
            var posX = centre_x + x;
            var posY = centre_y + y;
            img.setX(posX);
            img.setY(posY);
            var scale = .5 + (posY - 80) / 160;
            img.setScale(scale);
            img.setOpacity(scale - .2);
        });
    },
    node: calque
});
animation.start()
}

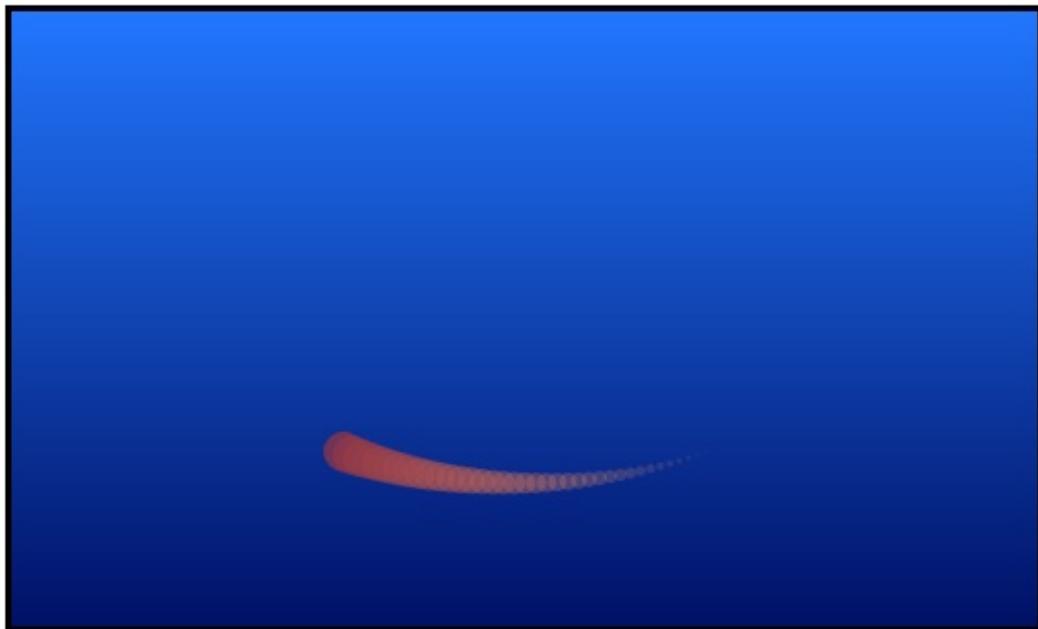
// Chargement de la page
window.onload = function() {
    var sources_icônes = [
        "img01.jpg",
        "img02.jpg",
        "img03.jpg",
        "img04.jpg",
        "img05.jpg",
        "img06.jpg",
        "img07.jpg",
        "img08.jpg",
        "img09.jpg",
        "img10.jpg"
    ];
    load images(sources icônes, init scène);
```

{

[Tester !](#)

## TP

Pour faire le point je vous propose un petit TP. Le but est d'obtenir cet effet :

[Tester !](#)

C'est une forme allongée qui va en rétrécissant en formant une queue qui suit une ellipse. Pour constituer ce genre d'effet le plus simple est d'empiler des disques colorés. A vous de jouer 😊.

**Secret** ([cliquez pour afficher](#))

Code : **JavaScript**

```
window.onload = function() {
    // Scène
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

    // Constitution du fond dégradé
    var fond = new Kinetic.Layer();
    var calque = new Kinetic.Layer();
    var rectangle = new Kinetic.Rect({
        width: 500,
        height: 300,
        fill: {
            start: {
                x: scène.getWidth() / 2,
                y: 0
            },
            end: {
                x: scène.getWidth() / 2,
                y: scène.getHeight()
            },
            colorStops: [0, "#27f", 1, "#016"]
        },
    });
    fond.add(rectangle);

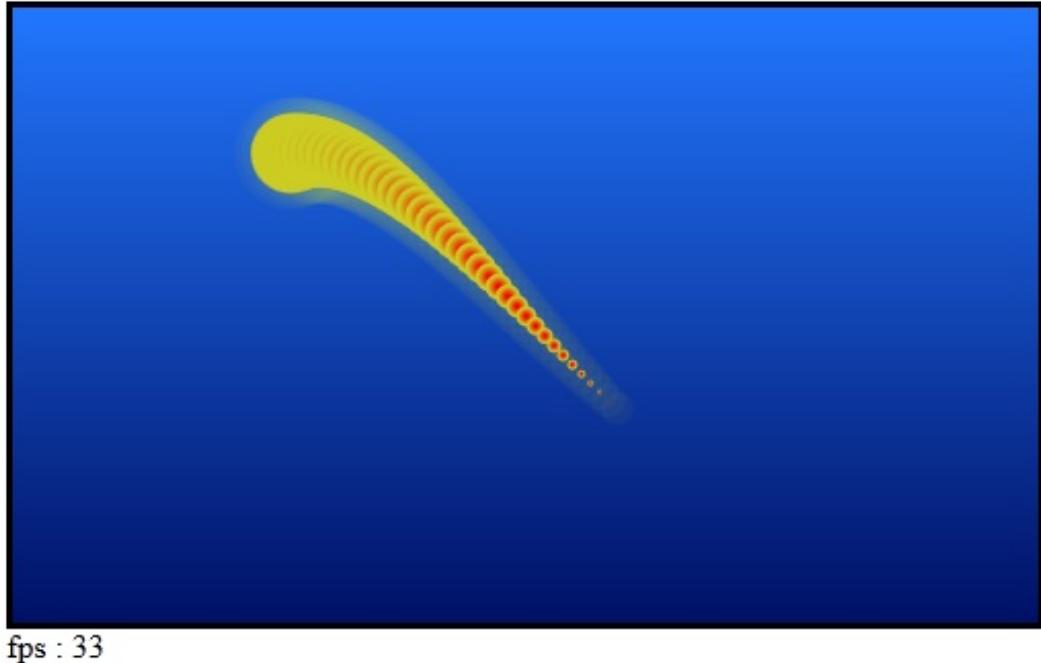
    // Constitution des balles
    var calque = new Kinetic.Layer();
    var a = 150;
    var b = 80;
    var centre_x = scène.getWidth() / 2;
    var centre_y = scène.getHeight() / 2;
    var ball_number = 40;
    var offset = .035;
    var red = 140;
    var green = 50;
    var blue = 70;
    for(var i = 1; i <= ball_number; ++i) {
        var balle = new Kinetic.Ellipse({
            radius: 10,
            fill: "rgb(" + (red + 3 * i) + ", " + (green + 3 * i) + ", " +
            (blue + i) + ")",
            opacity: .8 - (i / ball_number) * .8,
            scale: 1 - i / ball_number
        });
        calque.add(balle);
    }

    scène.add(fond);
    scène.add(calque);

    // Animation
    var animation = new Kinetic.Animation({
        func: function(frame) {
            var balles = calque.getChildren();
            var angle = frame.time / 500;
            for(var i = 0; i < balles.length; ++i) {
                var balle = balles[i];
                var angle_ball = angle - i * offset;
                var x = a * Math.cos(angle_ball);
                var y = b * Math.sin (angle_ball);
                var posX = centre_x + x;
                var posY = centre_y + y;
                balle.setX(posX);
                balle.setY(posY);
            }
        },
    });
}
```

```
    node: calque
});
animation.start()
};
```

Comme je vous sens encore en forme, on va améliorer l'effet :



Tester !

Cette fois la forme est plus volumineuse, avec un dégradé, un halo et elle suit un huit (un peu de maths en passant). J'ai aussi ajouté le contrôle du rafraîchissement pour juger les performances de la librairie.

**Secret** ([cliquez pour afficher](#))

Code : JavaScript

```
window.onload = function() {
    // Scène
    var scène = new Kinetic.Stage({
        container: "kinetic",
        width: 500,
        height: 300
    });

    // Constitution du fond dégradé
    var fond = new Kinetic.Layer();
    var calque = new Kinetic.Layer();
    var rectangle = new Kinetic.Rect({
        width: 500,
        height: 300,
        fill: {
            start: {
                x: scène.getWidth() / 2,
                y: 0
            },
            end: {
                x: scène.getWidth() / 2,
                y: scène.getHeight()
            }
        }
    });

    // Ajout des couches
    scène.add(fond);
    scène.add(calque);
    scène.add(rectangle);

    // Animation
    var animation = new Kinetic.Tween({
        node: calque,
        duration: 1000
    });
    animation.to({x: 0, y: 300}, 1000);
    animation.start();
}
```

```
        },
        colorStops: [0, "#27f", 1, "#016"]
    });
});
fond.add(rectangle);

// Constitution des balles
var calque = new Kinetic.Layer();
var a = 150;
var b = 80;
var centre_x = scène.getWidth() / 2;
var centre_y = scène.getHeight() / 2;
var ball_number = 40;
var offset = .03;
// Halo
for(var i = 1; i <= ball_number; ++i) {
    var balle = new Kinetic.Ellipse({
        radius: 20,
        fill: "yellow",
        opacity: .04,
        scale: (1 - i / ball_number) + .4
    });
    calque.add(balle);
}
// Corps
for(var i = 1; i <= ball_number; ++i) {
    var balle = new Kinetic.Ellipse({
        radius: 20,
        fill: {
            start: {
                x: 0,
                y: 0,
                radius: 3
            },
            end: {
                x: 0,
                y: 0,
                radius: 20
            },
            colorStops: [.6, "red", .8, "#cc2"]
        },
        opacity: .8 - (i / ball_number) * .8,
        scale: 1 - i / ball_number
    });
    calque.add(balle);
}

scène.add(fond);
scène.add(calque);

// Animation
var k = 2;
var animation = new Kinetic.Animation({
    func: function(frame) {
        document.getElementById("texte").innerHTML = "fps : " +
parseInt(1000 / frame.timeDiff);
        var balles = calque.getChildren();
        var angle = frame.time / 500;
        for(var i = 0; i < balles.length / 2; ++i) {
            var balle1 = balles[i];
            var balle2 = balles[i + balles.length / 2];
            var angle_ball = angle - i * offset;
            var x = a * Math.cos(angle_ball);
            var y = b * Math.sin (k * angle_ball);
            var posX = centre_x + x;
            var posY = centre_y + y;
            balle1.setX(posX);
            balle1.setY(posY);
            balle2.setX(posX);
            balle2.setY(posY);
        }
    }
});
```

```
        }
    },
    node: calque
});
animation.start()
};
```

Nous voici arrivés au bout de ce cours. Je suis loin d'avoir été exhaustif, mais vous avez déjà les bases pour créer de belles choses. Si vous voulez vous perfectionner, le mieux est d'analyser [les exemples](#) sur le site de l'auteur.