

# Comment créer un moteur d'affichage 2D en HTML5

Par Nicolas Legrand 

Date de publication : 11 mai 2014

Cet article a été publié avec l'aimable autorisation de **Nicolas Legrand**. L'article original (**Créer un moteur d'affichage 2D en HTML5**) peut être commenté sur **Google Documents**.

**Source** [Télécharger les sources de l'article.](#)

En complément sur Developpez.com

- [HTML Canvas pour les néophytes](#)
- [L'interactivité avec la balise HTML5 Canvas](#)
- [Réaliser une bannière animée en quelques lignes de code](#)
- [Le forum « Publications JavaScript et AJAX »](#)

I - Rapide Historique.....	3
I-A - JavaScript & ECMAScript.....	3
I-B - HTML5 et Canvas.....	3
II - Dessiner avec canvas.....	4
II-A - Fichier de base.....	4
II-B - Créer un canvas.....	4
II-C - Dessiner des primitives (lignes, cercles, carrés...).....	5
II-C-1 - Dessiner une ligne.....	5
II-C-2 - Dessiner un cercle.....	6
II-D - Alpha, Scale, Rotation et Translation.....	7
II-D-1 - L'alpha.....	7
II-D-2 - La translation.....	8
II-D-3 - Le scale.....	8
II-D-4 - La rotation.....	9
II-D-5 - Cumul des transformations, sauvegarde et restauration du contexte.....	10
III - Dessiner une image.....	11
III-A - Charger une image ( ou texture ).....	12
III-B - Dessiner une texture.....	12
III-C - Méthodes de dessin avancées.....	13
III-D - Dessiner à travers un masque.....	14
IV - Structure de base du moteur.....	15
IV-A - La POO en JavaScript, le prototypage.....	15
IV-B - Héritage et ordre d'inclusion.....	16
IV-C - Les Namespaces.....	17
V - Gestion des médias (ou assets).....	18
V-A - Introduction aux spritesheets.....	18
V-B - Gestion des Assets : AssetsLoader et AssetsManager.....	19
V-B-1 - La classe AssetsLoader.....	19
V-B-2 - La classe Texture.....	22
V-C - Regrouper toutes les textures, la classe TextureAtlas.....	23
VI - Les bases de l'affichage.....	26
VI-A - Structure arborescente et DisplayList.....	26
VI-B - Le point de départ : La classe DisplayObject.....	27
VI-C - Enfin des textures : La classe Bitmap.....	28
VI-D - Objets imbriqués : La classe DisplayObjectContainer.....	29
VI-E - Racine de la DisplayList : classe Stage.....	32
VII - Manipuler les objets : transformations et calculs matriciels.....	35
VII-A - Le problème des transformations imbriquées.....	35
VII-B - Les matrices, pour quoi faire ?.....	36
VII-C - Comment utiliser les matrices ?.....	36
VII-D - Implémentation des matrices dans le moteur.....	37
VIII - Modèle événementiel et design pattern.....	44
VIII-A - Pourquoi utiliser des événements ?.....	44
VIII-B - Comment gérer un modèle événementiel, le design pattern observer.....	44
VIII-C - Implémentation sur les objets d'affichage, la classe EventDispatcher et la classe Event.....	45
IX - Collisions et événements utilisateurs.....	48
IX-A - Comment détecter les événements souris ?.....	48
IX-B - Théorie des collisions, les formules.....	48
IX-C - Implémentation d'une méthode hitTest sur un DisplayObject.....	48
X - Les animations.....	51
X-A - Créer une classe MovieClip permettant de jouer une animation.....	52
XI - Les filtres.....	52
XI-A - Base des filtres avec la balise canvas.....	52
XI-B - Implémentation de la classe PixelFilter.....	54
XII - Conclusion et remerciements.....	55
XII-A - Formations HTML, HTML5, CSS & canvas.....	55
XII-B - Remerciements.....	56

## I - Rapide Historique

### I-A - JavaScript & ECMAScript

JavaScript (abrégé JS) est un **langage de programmation** de **scripts** principalement utilisé dans les **pages Web** interactives mais aussi côté serveur<sup>[1]</sup>. C'est un langage **orienté objet** à **prototype**, c'est-à-dire que les bases du langage et ses principales interfaces sont fournies par des **objets** qui ne sont pas des **instances** de **classes**, mais qui sont chacun équipés de **constructeurs** permettant de créer leurs propriétés, et notamment une propriété de prototypage qui permet d'en créer des objets **héritiers** personnalisés.

Le langage a été créé en 1995 par **Brendan Eich** (Brendan Eich étant membre du conseil d'administration de la fondation Mozilla) pour le compte de **Netscape Communications Corporation**. Le langage, actuellement à la version 1.8.2, est une implémentation de la 3e version de la norme **ECMA-262** qui intègre également des éléments inspirés du **langage Python**. La version 1.8.5 du langage est prévue pour intégrer la 5e version du standard ECMA<sup>[2]</sup>.

ECMAScript est un **langage de programmation** de type **script** standardisé par **ECMA International** dans le cadre de la spécification ECMA-262. Il s'agit donc d'un standard, dont les spécifications sont mises en œuvre dans différents langages script, comme **JavaScript** ou **ActionScript**, ainsi qu'en **C++** (norme 2011). C'est un langage de programmation orienté objet.

En décembre 1995, **Sun** et **Netscape** annoncent la sortie de JavaScript. En mars 1996, Netscape implémente le moteur JavaScript dans son navigateur Web **Netscape Navigator** 2.0. Le succès de ce navigateur contribue à l'adoption rapide de JavaScript dans le développement Web orienté client. **Microsoft** réagit alors en développant **JScript**, qu'il inclut ensuite dans **Internet Explorer** 3.0 en août 1996 pour la sortie de son navigateur.

Netscape soumet alors JavaScript à l'ECMA pour standardisation. Les travaux débutent en novembre 1996, et se terminent en juin 1997 par l'adoption du nouveau standard ECMAScript. Les spécifications sont rédigées dans le document Standard ECMA-262.

### I-B - HTML5 et Canvas

HTML5 (*HyperText Markup Language 5*) est la prochaine révision majeure d'**HTML (format de données** conçu pour représenter les **pages Web**). Cette version est en développement en 2013 et est appelée à devenir le prochain standard du web en matière de développement d'applications riches sur internet (RIA). Elle apporte certaines innovations qui manquaient au langage tout en se payant le luxe d'être intégrée au navigateur, offrant ainsi une alternative à certaines technologies propriétaires comme Flash ou Unity, tout en basées elles, sur des plugins. L'une des grandes innovations de cette nouvelle mouture est la mise à disposition de la balise `<canvas>` et de toute une API de dessin permettant au programmeur de réaliser du contenu interactif de type jeu vidéo, chat webcam, manipulation de vidéo, connexion à un serveur de sockets etc. Jusqu'ici, développer ce type de contenu avec les anciennes normes était extrêmement difficile sinon impossible.

En suivant ce cours vous apprendrez à vous construire un moteur d'affichage 2d que nommerons "Tomahawk#". La version présentée ici est une version pédagogique, par conséquent quelques erreurs ont pu se glisser ici et là, n'hésitez pas à le signaler dans les commentaires.

Sachez cependant que le moteur Tomahawk existe et qu'il est disponible sur internet, il a pour vocation à être amélioré par une communauté de développeurs toujours plus grande et compétente, peut être vous ?

Vous pourrez retrouver la version dédiée à la production du moteur Tomahawk en ligne sur le github du projet à l'adresse suivante : <https://github.com/thetinyspark/tomahawk>

## II - Dessiner avec canvas

### II-A - Fichier de base

Nous allons commencer par créer un fichier de base pour notre application sur la base du modèle suivant :

```
index.html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title></title>
  <script type="text/javascript" src="sample.js"></script>
  <style type="text/css">
    canvas{
      border: 2px solid black;
      background-color: black;
    }
  </style>
</head>
<body>
  <!-- La balise canvas que nous utiliserons dans le fichier sample.js-->
  <canvas id="tomahawk" width=#800# height=#600#></canvas>
</body>
</html>
```

```
sample.js
/* Point d'entrée de l'application */
function init(){
  //code de notre Application
}

/*
* Quand toutes les données sont chargées ( DOM, Images, Sons, Vidéos etc ... )
* On démarre l'application par la fonction init
*/
window.onload = init;
```

Le fichier `index.html` est un fichier HTML5 simple, les seuls éléments notables sont :

- l'inclusion d'un fichier JavaScript nommé `sample.js` ;
- la présence d'une balise `<canvas>` dont l'id est "tomahawk" (nom de notre moteur) ;
- le style CSS donné à la balise en question (à changer à votre convenance).

Sur une base de code simple, nous allons pouvoir passer à la suite, créer un canvas et l'utiliser.

### II-B - Créer un canvas

Créer un canvas peut se faire de deux façons :

- soit de la manière la plus simple qui soit c'est-à-dire en ajoutant une balise `<canvas>` au sein du DOM (dans le code HTML) ;
- soit en la créant par programmation et en l'ajoutant manuellement au DOM de la façon suivante :

```
var canvas = document.createElement("canvas");
document.getElementsByTagName("body")[0].appendChild(canvas);
```

Dans notre exemple, nous choisirons la première méthode, c'est-à-dire ajouter une balise `<canvas>` au code HTML.

Notre fichier `index.html` en contient déjà une, nous allons donc créer une fonction pour la retrouver.

```

/*
 * Retourne une référence à l'objet canvas créée à l'aide de la balise
 * placée dans le code html
 */
function getCanvas() {
    return document.getElementById("tomahawk");
}

```

À l'aide de cette méthode, nous avons accès à l'objet `canvas`, maintenant il nous faut pouvoir dessiner à l'intérieur. Pour ce faire nous allons d'abord devoir récupérer le contexte de l'objet `canvas`, pour de plus amples informations sur ce qu'est un contexte je vous invite à vous rendre sur le site du W3C.

Le contexte de l'objet `canvas` se récupère à l'aide de la méthode `getContext()` la façon suivante :

```

canvas.getContext(String contextMode);

```

Où `canvas` est une instance d'un objet de type `Canvas` et `contextMode` est un paramètre de type chaîne permettant de préciser quel genre de contexte on souhaite récupérer (par exemple un contexte 2d ou un contexte WebGL).

Nous allons donc créer une méthode nous permettant de récupérer le contexte 2d de notre canvas.

```

/*
 * Retourne le contexte d'exécution 2d du canvas
 */
function getContext() {
    return getCanvas().getContext("2d");
}

```

Voilà, à présent nous avons tous les outils nécessaires pour pouvoir dessiner dans notre canvas.

## II-C - Dessiner des primitives (lignes, cercles, carrés...)

HTML5 embarque toute une API dédiée au dessin, ce qui permet aux développeurs de créer du contenu 2d sans avoir systématiquement recours à des images. À l'aide cette API, on peut dessiner n'importe quelle forme géométrique, voire des dessins plus complexes, la seule vraie limite est votre imagination.

### II-C-1 - Dessiner une ligne

Commençons par le plus simple, dessiner une ligne de couleur verte faisant cinq pixels d'épaisseur.

En HTML5, toute opération de dessin se réalise à l'aide d'un objet de type `Context`, ça tombe plutôt bien, nous venons de créer une fonction nous permettant de récupérer ce contexte !

```

sample1.js
var context = getContext();

context.beginPath();
context.lineWidth = 5;
context.moveTo( 0, 0 );
context.lineTo( 100, 100 );
context.strokeStyle = '#003300';
context.stroke();

```

Avant toute opération de dessin en HTML5, il nous faudra commencer par un `beginPath()` qui permet, comme son nom l'indique, de commencer un « chemin », comprendre par là que l'on initialise un nouveau cycle de dessin, un peu comme si l'on prenait une nouvelle feuille vierge pour dessiner.

Faisons le point de ce que nous avons actuellement sur notre canvas.

Une ligne dont les propriétés sont :

- une épaisseur de 5px définie par la propriété `lineWidth = 5` ;
- une couleur définie par la propriété `strokeStyle = '#003300'` ;
- un point de départ situé à `x = 0px` et `y = 0px` que nous avons défini avec l'appel à la fonction `moveTo( 0, 0 )` ;
- un point d'arrivée situé à `x = 100px` et `y = 100px` que nous avons relié au point de départ en faisant appel à la fonction `lineTo( 100, 100 )`, qui relie le dernier point dessiné au point dont les coordonnées sont passées en paramètres.

Vous pouvez remarquer que la dernière ligne de notre code se termine par `context.stroke()`, cette méthode permet de dessiner l'ensemble du jeu d'instructions définies entre l'appel à `context.beginPath()` et `context.stroke()`, si vous commentez l'appel à cette méthode, rien ne sera dessiné.

Notez également que `context.stroke()` n'exécute que les jeux d'instructions relatifs aux **lignes** et pas aux formes pleines, ces dernières sont gérées de manière différente, ce qui nous amène à la suite, dessiner des primitives « pleines », en commençant par le cercle.

## II-C-2 - Dessiner un cercle

Si je souhaite dessiner un cercle, je peux utiliser la méthode suivante :

```
context.arc(x, y, radius, startAngle, endAngle, counterClockwise);
```

Où `x` et `y` représentent les coordonnées du centre de mon arc, `radius` le rayon ( en pixels ) de mon arc, `startAngle` et `endAngle` les angles de départ et d'arrivée (en radians) et `counterClockwise` un booléen qui sert à définir si l'arc est défini dans le sens antihoraire ou non. Étudions à présent le code suivant :

### sample2.js

```
var context = getContext();
var toRadians = Math.PI / 180;
var startAngle = 0 * toRadians;
var endAngle = 360 * toRadians;

context.beginPath();
context.fillStyle = "green";
context.moveTo(100,100);
context.arc(100, 100, 50, startAngle, endAngle, false);
context.lineTo(100,100);
context.fill();
```

Comme pour la ligne, faisons le point de ce que nous avons actuellement sur notre canvas.

Un cercle dont les propriétés sont :

- une couleur de remplissage définie par la propriété `fillStyle = 'green'` ;
- le centre de départ situé à `x = 100px` et `y = 100px` représentés par les deux premiers paramètres ;
- un rayon de 50px représenté par le troisième paramètre ;
- un angle de départ situé à 0 degrés et converti en radians représenté par le quatrième paramètre ;
- un angle de départ situé à 360 degrés et converti en radians représenté par le cinquième paramètre ;
- une direction dans le sens horaire car le sixième paramètre vaut `false`.

Comme vous pouvez le constater, nous utilisons toujours la méthode `context.beginPath()` pour créer un nouveau dessin. Afin de pouvoir exécuter le nouveau jeu d'instructions, relatif cette fois-ci à des formes pleines, nous utilisons la méthode `context.fill()` qui agit de la même façon que la méthode `context.stroke()`.

En regardant un peu plus en avant l'API de dessin, on peut s'apercevoir qu'il existe pas mal de méthodes pour dessiner d'autres primitives, ou d'autres types de lignes, qu'elles soient droites ou dessinées à l'aide de courbes de Bézier etc.

**Exemple :**

voici un exemple permettant de dessiner une ligne rouge puis un rectangle bleu et terminer par un cercle vert sur le même canvas.

#### sample3.js

```

var context = getContext();
var toRadians = Math.PI / 180;
var startAngle = 0 * toRadians;
var endAngle = 360 * toRadians;

context.beginPath();
context.strokeStyle = "#FF0000";
context.lineWidth = 5;
context.moveTo( 50, 50 );
context.lineTo( 150, 200 );
context.stroke();

context.beginPath();
context.fillStyle = "blue";
context.fillRect( 20, 20, 100, 100 );
context.fill();

context.beginPath();
context.fillStyle = "green";
context.moveTo(100,100);
context.arc(100, 100, 50, startAngle, endAngle, false);
context.lineTo(100,100);
context.fill();

```

Maintenant que nous savons comment dessiner des primitives, nous allons apprendre à les transformer, comprendre par là que nous allons leur appliquer un changement d'échelle, de rotation, d'alpha ou de translation.

## II-D - Alpha, Scale, Rotation et Translation

Appliquer une transformation en HTML5 est facile, en effet l'API met à notre disposition des méthodes simples, la seule difficulté réside dans le fait que ces méthodes sont cumulatives mais nous reviendrons là-dessus plus tard, pour l'instant nous allons nous contenter d'appliquer des transformations à un carré.

### II-D-1 - L'alpha

Pour dessiner quelque chose en transparence, on modifie la propriété :

```

// pour définir la transparence à 50%
context.globalAlpha = 0.5;

```

La valeur de la propriété `globalAlpha` du contexte se situe toujours entre 0 et 1, si, comme dans l'exemple ci-dessus, vous souhaitez obtenir un alpha de 50 %, il vous suffit de modifier la valeur de cette propriété à 0.5.

Facile n'est-ce pas ? Par exemple si je veux dessiner un cercle avec une transparence de 50 % par dessus un carré mon code ressemblera à ceci :

#### sample4.js

```

var context = getContext();
var toRadians = Math.PI / 180;
var startAngle = 0 * toRadians;
var endAngle = 360 * toRadians;

context.beginPath();
context.fillStyle = "blue";
context.fillRect( 25, 25, 100, 100 );
context.fill();

```

#### sample4.js

```
context.globalAlpha = 0.5;
context.beginPath();
context.fillStyle = "green";
context.arc(100, 100, 50, startAngle, endAngle, false);
context.fill();
```

On aperçoit bien le carré en transparence derrière le cercle. Essayez de jouer un peu avec les valeurs de l'alpha, vous pouvez également changer la valeur de l'alpha du carré et remettre l'alpha du cercle à 1 ( ou toute autre valeur ), c'est vraiment facile !

Vous avez sans doute remarqué que jusqu'ici nous avons défini les coordonnées de nos primitives à l'aide des paramètres fournis à cet effet, toutefois lors de vos futurs développements, vous verrez qu'il n'est pas forcément pratique de procéder de la sorte. Le mieux serait de pouvoir dessiner nos objets aux coordonnées 0, 0 et de les déplacer ensuite. Ça tombe plutôt bien, la prochaine transformation que je compte vous montrer est la translation.

## II-D-2 - La translation

Pour effectuer une translation, on utilise la méthode :

```
context.translate( translateX, translateY );
```

Où `translateX` est le déplacement sur l'axe des x (en pixels) que vous souhaitez obtenir et `translateY` la même chose mais sur l'axe des y. Ainsi, pour dessiner un carré rouge de 100 pixels de côtés prenant son point d'origine aux coordonnées x = 47 et y = 72, j'aurai à écrire le code suivant :

#### sample5.js

```
var context = getContext();

context.translate( 47, 72 );
context.beginPath();
context.fillStyle = "red";
context.fillRect( 0, 0, 100, 100 );
context.fill();
```

Notez que nous aurions tout aussi bien pu utiliser les deux premiers paramètres de la méthode `fillRect` ( ce que nous faisons jusqu'ici ), toutefois comme expliqué plus haut, il vous sera plus utile d'utiliser les méthodes de transformations par la suite plutôt que d'utiliser ce type de paramètre.

Passons maintenant au changement d'échelle.

## II-D-3 - Le scale

Pour effectuer un changement d'échelle, on utilise la méthode :

```
context.scale( scaleX, scaleY );
```

Où `scaleX` est l'échelle sur l'axe des x que vous souhaitez obtenir et `scaleY` la même chose mais sur l'axe des y. Ainsi, pour dessiner le même carré que dans l'exemple précédent mais à une échelle deux fois plus grande, nous aurons le code suivant :

#### sample6.js

```
var context = getContext();

context.translate( 47, 72 );
context.scale( 2, 2 );
context.beginPath();
context.fillStyle = "red";
```



**sample6.js**

```
context.fillRect( 0, 0, 100, 100 );  
context.fill();
```

Ainsi, nous obtenons un **carré de 100 pixels** de côtés mais dont **l'échelle est de 2**, ainsi visuellement, j'ai un carré de 200 pixels de côtés. Quel est l'intérêt de cette méthode ? Pourquoi ne pas directement un carré de 200 pixels de côtés ? En plus on code moins !

Et bien l'intérêt principal est de **ne pas avoir à recalculer la largeur et la hauteur d'un objet d'affichage** à chaque fois que l'on souhaite changer son échelle, de plus, ces calculs sont simples à réaliser lorsque l'objet en question n'est pas en rotation, mais dès qu'il s'agit de calculer une largeur et une hauteur avec une rotation par dessus le marché, ça devient plus compliqué et **plus coûteux en ressources**.

Passons maintenant à la dernière transformation, la rotation

**II-D-4 - La rotation**

Avant de commencer, il me faut éclaircir un point que nous avons omis de préciser jusque là : l'unité de mesure employée pour une rotation. En effet, alors que la plupart des gens calculent leurs angles en degrés, en programmation graphique il est de coutume d'employer le radian.

Vu que ce livre n'a pas vocation à être un cours de mathématiques, je vais tout simplement vous donner la formule de conversion degrés/radians et vous laisser approfondir ce point si vous le souhaitez (internet fourmille de ressources sur la question).

La formule de conversion degrés/radians est la suivante :

```
angle_radians = angle_degré * ( Math.PI / 180 );
```

Nous l'avons déjà utilisé plus haut pour définir les angles de départ et de fin de notre arc.

Maintenant, nous savons que lorsqu'on parlera d'un angle, on s'exprimera par défaut en radians et si l'on change d'unité de mesure, je vous le préciserai alors.

Bien, maintenant que tout le monde parle le même langage, laissez-moi vous présenter la méthode qui vous permettra d'appliquer une rotation à vos objets :

```
context.rotate( angle_radian );
```

Assez simple n'est-ce pas ? Ainsi, pour continuer sur l'exemple de notre carré rouge, nous allons reprendre le code de tout à l'heure et ajouter une rotation de 37° à notre carré :

**sample7.js**

```
var context = getContext();  
var toRadians = Math.PI / 180;  
  
context.translate( 47, 72 );  
context.scale( 2, 2 );  
context.rotate( 37 * toRadians );  
context.beginPath();  
context.fillStyle = "red";  
context.fillRect( 0, 0, 100, 100 );  
context.fill();
```

Notez que toutes les rotations s'effectuent dans le sens **horaire** ! Nous avons vu les transformations que nous voulions voir, nous y reviendrons plus tard. Il nous reste à voir le cumul des transformations, la sauvegarde et restauration du contexte et nous aurons terminé ce premier chapitre.

## II-D-5 - Cumul des transformations, sauvegarde et restauration du contexte

L'objet `context` utilise une **matrice** pour représenter et stocker le résultat de toutes les transformations qu'on lui applique. Nous ne nous étendrons pas pour l'instant sur ce qu'est une matrice ni comment l'utiliser, en revanche, sachez qu'une des lois basiques des calculs matriciels est la **commutativité**.

En clair, cela signifie que les transformations que l'on applique à une matrice se cumulent et que l'ordre dans lequel on les exécute influe sur le résultat obtenu.

Exemple :

sample8.js

```
var context = getContext();
// ici on applique une translation AVANT le scale
context.translate( 47, 72 );
context.scale( 2, 2 );

context.beginPath();
context.fillStyle = "red";
context.fillRect( 0, 0, 100, 100 );
context.fill();

var context = getContext();
// ici on applique une translation APRES le scale
context.scale( 2, 2 );
context.translate( 47, 72 );

context.beginPath();
context.fillStyle = "red";
context.fillRect( 0, 0, 100, 100 );
context.fill();
```

On peut voir que le résultat obtenu à l'écran est différent suivant que l'on applique **scale avant** ou **après** la translation, tout cela est normal, la loi de commutativité est en marche. Mais alors dans quel ordre appliquer mes transformations ? Et bien ça, c'est à vous de le décider, bien qu'en règle générale le résultat attendu nécessite que l'on applique dans l'ordre une translation, une rotation et enfin l'échelle.

Et vous pensiez que c'était fini ? Et bien non, en effet les transformations en html5 c'est pas de la tarte ( enfin uniquement quand on y est pas habitué, après je vous rassure ça roule tout seul ).

Si je veux par exemple définir une échelle à 0,5 après avoir l'avoir définie à 2, le code suivant ne fonctionne pas :

```
var context = getContext();

context.scale( 2, 2 ); // l'échelle est à 2
context.scale( 0.5, 0.5 );
// ici l'échelle ne vaut pas 0.5 MAIS 1 car j'ai MULTIPLIE la valeur
// de l'échelle courante par 0.5, donc le résultat est 1, pour avoir une valeur d'échelle à 2, il aurait
// fallu que j'applique un scale de 0,25
```

Le problème de cette commutativité, c'est que je ne suis pas forcément au courant de l'état actuel de ma matrice au moment où je l'utilise, donc cela peut me poser pas mal de problèmes pour obtenir l'état désiré.

Heureusement, il existe une parade à cela : la sauvegarde du contexte ! En effet, il est possible de stocker en mémoire l'état du contexte et de le restaurer par la suite. Cela fonctionne avec la paire de méthodes suivantes :

```
context.save()
context.restore()
```

La méthode, `context.save()` permet de sauvegarder l'état actuel du contexte, la méthode `context.restore()` permet quant à elle de restituer l'état du dernier contexte sauvegardé, c'est-à-dire que les données de transformations de la matrice ainsi que les données de dessins etc. seront exactement les mêmes que lors du dernier appel à `context.save()`.

Ces méthodes fonctionnent un peu à la manière d'une pile d'assiettes, c'est-à-dire que le dernier contexte sauvegardé ira « au-dessus » de la pile et donc, lors du prochain appel à `context.restore()` ce sera cette dernière « assiette » qui sera restituée.

Dans l'exemple suivant, je dessine quatre carrés, tous de la même taille mais avec des transformations différentes. L'utilisation de `context.save()` et `context.restore()` est indispensable pour pouvoir récupérer l'état de la précédente matrice.

#### sample9.js

```
var context = getContext();

context.save(); // sauvegarde 1
context.translate( 50, 50 );
context.scale( 2, 2 );

context.beginPath();
context.fillStyle = "red"
context.fillRect( 0, 0, 100, 100 );
context.fill();

context.save(); // sauvegarde 2, ici les transformations et les paramètres de dessins sont sauvés.

context.rotate( 60 * ( Math.PI / 180 ) );

context.beginPath();
context.fillStyle = "green"
context.fillRect( 0, 0, 100, 100 );
context.fill();

context.restore(); // restaure la 2ème sauvegarde

context.beginPath();
context.fillStyle = "blue"
context.fillRect( 0, 0, 50, 50 );
context.fill();

context.restore(); // restaure la 1ère sauvegarde

context.beginPath();
context.fillStyle = "purple"
context.fillRect( 0, 0, 100, 100 );
context.fill();
```

Voilà, les bases des transformations et du dessin avec canvas sont posées, nous venons de clôturer ce premier chapitre.

## III - Dessiner une image

Nous allons maintenant passer au dessin d'image en HTML5.

## III-A - Charger une image ( ou texture )

En JavaScript, il est très simple de charger une image, aussi appelée texture dans le domaine du jeu vidéo (d'ailleurs nous utiliserons ce terme dorénavant). Il y a de multiples façons de faire, pour le besoin de ce chapitre nous utiliserons la plus simple qui est d'inclure les images directement dans la structure HTML.

Il nous suffira alors de démarrer le script au chargement total de la page et pour ça nous utiliserons l'événement natif JavaScript `window.onload`.

Exemple:

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title></title>
  <script type="text/javascript" src="sample.js"></script>
  <style type="text/css">
    canvas{
      border: 2px solid black;
      background-color: black;
    }
  </style>
</head>
<body>
  <canvas id="tomahawk" width="800" height="600"></canvas>
  
</body>
</html>
```

sample.js

```
/*
 * Retourne une référence à l'objet canvas
 */
function getCanvas() {
  return document.getElementById("tomahawk");
}

/*
 * Retourne le contexte d'exécution 2d du canvas
 */
function getContext() {
  return getCanvas().getContext("2d");
}

/* Point d'entrée de l'application */
function init() {
  //code de notre Application
  var canvas = getCanvas();
  var context = getContext();
}

/* Quand toutes les données sont chargées ( DOM, Images, Sons, Vidéos etc ... )
 * On démarre l'application par la fonction init
 */
window.onload = init;
```

## III-B - Dessiner une texture.

Pour dessiner une texture, l'API HTML5 embarque une méthode dont voici la **première** signature :

```
context.drawImage(image, dx, dy)
```

- Le paramètre `image` représente la texture que l'on souhaite dessiner.
- `dx` et `dy` représentent les coordonnées auxquelles on souhaite dessiner l'image.

Cette méthode vous permettra de **dessiner directement la texture sur le canvas, sans aucune transformation**.

Exemple :

```
sample1.js
//code de notre Application
var canvas = getCanvas();
var context = getContext();
var texture = document.getElementById('persol');

context.save(); // sauvegarde 1
context.drawImage(texture,0,0);
context.restore();
```

### III-C - Méthodes de dessin avancées

À l'heure actuelle, nous avons vu la méthode **classique** pour dessiner une texture, nous allons à présent faire un tour du côté des méthodes de dessin **avancées** en commençant par dessiner une texture à la taille qu'on souhaite obtenir.

Pour ce faire, nous allons voir la deuxième signature de la méthode `context.drawImage()` :

```
context.drawImage(image, dx, dy, dw, dh)
```

- `image` représente la texture que l'on souhaite dessiner.
- `dx`, `dy`, `dw` et `dh` représentent le rectangle dans lequel on dessinera cette texture dans le canvas.

Exemple :

```
sample2.js
/* Point d'entrée de l'application */
function init(){
    //code de notre Application
    var canvas = getCanvas();
    var context = getContext();
    var texture = document.getElementById('persol');

    context.save(); // sauvegarde 1
    context.drawImage(texture,10,10,100,100);
    context.restore();
}
```

Ici, **on dessine notre texture** aux coordonnées `x = 10`, `y = 10` et **on applique une transformation à cette texture** de façon à ce qu'elle soit dessinée avec 100 pixels de largeur et 100 pixels de hauteur (`width = 100`, `height = 100`).

Nous allons maintenant voir comment dessiner une portion de notre texture et pour cela, nous allons étudier la **troisième** et dernière signature possible de la méthode `context.drawImage()`.

```
context.drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh)
```

- `image` représente la texture que l'on souhaite dessiner.
- `sx`, `sy`, `sw` et `sh` représentent la portion de texture que l'on souhaite dessiner.
- `dx`, `dy`, `dw` et `dh` représentent le rectangle dans lequel on dessinera cette fameuse portion de texture.

Exemple :

#### sample3.js

```

/* Point d'entrée de l'application */
function init(){
  //code de notre Application
  var canvas = getCanvas();
  var context = getContext();
  var texture = document.getElementById('persol');

  context.save(); // sauvegarde 1
  context.drawImage(texture,20,0,100,100,0,0,200,200);
  context.restore();
}

```

Ici, on dessine uniquement une portion de l'image, cette portion est comprise entre les coordonnées  $x = 20$ ,  $y = 20$  avec une largeur et une hauteur de 100 pixels, **cette portion d'image sera dessinée sur le canvas** dans un rectangle compris entre les coordonnées  $x = 0$ ,  $y = 0$  et  $x = 200$ ,  $y = 200$ .

Maintenant, nous allons pouvoir passer au dessin avec des ombres, nous allons donc nous intéresser aux propriétés : `shadowColor`, `shadowBlur`, `shadowOffsetX` et `shadowOffsetY` de l'objet `context`. Ces propriétés peuvent être utilisées avec les méthodes de dessin classiques, sans texture ou avec.

- La propriété `context.shadowOffsetX` sert à définir le décalage en x que l'ombre aura par rapport au dessin, le type de cette propriété est un entier.
- La propriété `context.shadowOffsetY` sert à définir le décalage en y que l'ombre aura par rapport au dessin, le type de cette propriété est un entier.
- La propriété `context.shadowColor`, comme son nom l'indique, définit la couleur de l'ombre, le type de cette propriété est une chaîne.
- La propriété `context.shadowBlur` elle, sert à spécifier la netteté ( ou plus spécifiquement le flou ) que l'on souhaite appliquer à cette ombre, le type de cette propriété est un entier.

Exemple :

#### sample4.js

```

/* Point d'entrée de l'application */
function init(){
  //code de notre Application
  var canvas = getCanvas();
  var context = getContext();
  var texture = document.getElementById('persol');
  context.save(); // sauvegarde 1

  // je souhaite dessiner une texture avec une ombre rouge, décalée de 20 pixels sur l'axe des x
  // et de 20 pixels sur l'axe des y avec une qualité de flou de 2.

  context.shadowColor = "#FF0000";
  context.shadowBlur = 2;
  context.shadowOffsetX = 20;
  context.shadowOffsetY = 20;

  // je dessine ma texture
  context.drawImage(texture,20,0,100,100,0,0,200,200);
  context.restore();
}

```

### III-D - Dessiner à travers un masque

Nous allons à présent apprendre à dessiner à travers un masque.

Pour cela, nous allons avoir besoin de la propriété `globalCompositeOperation` de l'objet `context`.

Cette propriété peut prendre plusieurs valeurs dont les résultats sont illustrés à l'aide du graphique ci-dessous (source: <http://www.html5canvastutorials.com/advanced/html5-canvas-global-composite-operations-tutorial/>) :

Le contenu dynamique ne peut pas être affiché dans ce support, veuillez consulter la page [en ligne](#) pour le visualiser.

Comment interpréter ce graphique ?

En premier lieu, il faut que je vous donne le code source qui va avec :

```
context.beginPath();
context.fillStyle = "blue";
context.fillRect(0,0,55,55);
context.fill();
context.globalCompositeOperation = "source-in";
context.beginPath();
context.fillStyle = "red";
context.arc(50,50,35,0,2 * Math.PI,false);
context.fill();
```

Il s'agit d'une opération en deux étapes :

- tout d'abord, **on dessine le masque** sur l'objet `context` (ici le rectangle bleu) ;
- ensuite, **on spécifie la valeur** de la propriété `globalCompositeOperation` de l'objet `context` (ici `source-in`) ;
- puis **on dessine le contenu** que l'on souhaite voir apparaître à travers le masque (le cercle rouge)

Il suffit de changer la valeur de la propriété `globalCompositeOperation` pour pouvoir obtenir tous les résultats retranscrits sur le graphique. Pour dessiner un objet à travers un masque, la valeur qui nous intéresse est `source-in`. Voilà, les bases du dessin de texture ont été passées en revue, nous allons maintenant passer aux bases du moteur d'affichage lui-même.

## IV - Structure de base du moteur

### IV-A - La POO en JavaScript, le prototypage

Le JavaScript est un langage orienté objet par prototype, il ne donne donc pas la possibilité de créer ce que l'on appelle des classes ou même des espaces de noms (*namespaces*) et autres subtilités propres aux langages orientés objets par classes que l'on a l'habitude d'utiliser.

Cependant, c'est un langage à prototypes, c'est-à-dire que la notion d'objet existe mais qu'ils sont tous dynamiques (à part quelques rares exceptions) et que les méthodes et variables membres peuvent être ajoutées dynamiquement et à l'exécution, ce qui procure une extrême souplesse à ce langage.

Ce qui fait qu'en définissant une nouvelle fonction « Voiture » en JavaScript, je peux utiliser le mot clef `new` du langage et ainsi créer une nouvelle instance de ma pseudo classe `Voiture`, je dispose ainsi d'un objet `Voiture` tout neuf.

Chaque fonction/classe (dorénavant nous utiliserons le mot classe même s'il s'agit clairement d'un abus de langage) possède une propriété `prototype` qu'il nous est permis d'étendre à l'infini.

Ainsi à chaque nouvelle objet crée, celui-ci possèdera les propriétés définies sur le prototype de la classe associée.

Exemple :

```
// fonction constructeur
function Voiture(name,id){
    this.name = name;
    this.id = id;
```

```
}

Voiture.prototype.name = null;
Voiture.prototype.id = null;

var chevrolet = new Voiture("chevrolet",1);
var ferrari = new Voiture("ferrari",2);
```

Nous avons ici deux objets de type `Voiture` ( bien que la notion de typage soit vraiment implicite en JavaScript ) avec chacun des propriétés qui ont des valeurs différentes.

## IV-B - Héritage et ordre d'inclusion

Il existe plusieurs librairies JavaScript sur internet, chacune ayant sa propre façon de gérer l'héritage en JavaScript. En effet, le langage ne procure pas de moyen classique de le faire. Nous allons coder le notre, très simple, qui répondra à nos besoins.

Nous allons donc créer une classe nommée `Tomahawk` ( nom de notre moteur ) qui disposera de plusieurs méthodes statiques qui nous permettront « d'enregistrer » des classes auprès de la plateforme ainsi cette dernière pourra résoudre les problématiques d'héritage.

```
function Tomahawk() {}

Tomahawk._classes = {};
Tomahawk._extends = [];

Tomahawk.registerClass = function( classDef, className ){
    Tomahawk._classes[className] = classDef;
};
```

La méthode `registerClass` prend deux arguments :

- `classDef` qui correspond à la classe en elle-même ;
- `className`, un « alias » qui sera l'identifiant de la classe.

Maintenant, il nous faut une méthode pour « enregistrer » l'héritage d'une classe.

```
Tomahawk.extend = function( p_child, p_ancestor ){
    Tomahawk._extends.push({"child":p_child,"ancestor":p_ancestor});
};
```

Cette méthode prend elle aussi deux paramètres, chacun correspondant à une chaîne, le premier étant l'alias de la classe fille, le deuxième étant l'alias de la classe parente.

Maintenant nous devons résoudre cette problématique d'héritage, le principe en soit est très simple, il faut que les prototypes de chacune de nos classes disposent d'une copie de toutes les méthodes et propriétés de leur classe parente sans pour autant remplacer les surcharges de méthodes définies sur le prototype de la classe fille.

```
Tomahawk._getParentClass = function(child) {
    var i = 0;
    var max = Tomahawk._extends.length;

    for (i = 0; i < max; i++) {
        obj = Tomahawk._extends[i];
        if( obj["child"] == child )
            return obj;
    }

    return null;
};
```



```
Tomahawk._inherits = function( obj ){
  var child = null;
  var ancestor = null;
  var superParent = Tomahawk._getParentClass(obj["ancestor"]);

  if( superParent != null )
    Tomahawk._inherits(superParent);

  child = Tomahawk._classes[obj["child"]];
  ancestor = Tomahawk._classes[obj["ancestor"]];

  if( child != null && ancestor != null ){
    ancestor = new ancestor();
    for( var prop in ancestor ){
      if( !child.prototype[prop] ){
        child.prototype[prop] = ancestor[prop];
      }
    }
  }
};
```

Et enfin, il nous faut une méthode pour démarrer tout ça :

```
Tomahawk.run = function(){
  var obj = null;
  var i = 0;
  var max = Tomahawk._extends.length;

  for (i = 0; i < max; i++){
    Tomahawk._inherits( Tomahawk._extends[i] );
  }
}
```

## IV-C - Les Namespaces

Dans le domaine de la programmation orientée objet, un *namespace* est, comme son nom l'indique, un espace de nom. Cela sert à pouvoir définir autant de fonctions, classes etc. au sein d'un même domaine. Concrètement **cela évite les collisions de nom** que l'on peut retrouver par exemple dans le cas de l'utilisation de bibliothèques JavaScript différentes.

Exemple :

- la bibliothèque A contient une fonction nommée `google`, j'ai besoin de cette fonctionnalité, du coup j'utilise la bibliothèque associée ;
- La bibliothèque B contient également une fonction nommée `google`, elle ne fait pas la même chose, cependant j'ai besoin d'autres fonctionnalités de cette bibliothèque, du coup je l'utilise également dans ma page.

Le problème c'est que **ces deux bibliothèques vont entrer en collision** à cause de cette fonction nommée `google`. Du coup, la fonction `google` sera redéfinie, ce qui fait que la première bibliothèque sera en partie inutilisable.

Pour résoudre ce problème, on utilise les *namespaces*, le problème c'est qu'il **n'existe pas de mécanisme propre au langage JavaScript** permettant de déclarer ou de changer de namespace.

Heureusement, **le prototypage** peut encore venir à notre secours, en effet, il suffit de créer un objet (à qui l'on donnera le nom de notre *namespace*) et de créer nos classes en tant que propriété de cet objet. Concrètement ça donne cela :

```
// fonction message classique
function Message() {
  console.log("Message 1");
};

// fonction message incluse dans le namespace "tomahawk"
```

```
var tomahawk = {};
tomahawk.Message = function() {
    console.log("Message 2");
};

Message();
tomahawk.Message();
```

Nous avons passé en revue les *namespaces* et leur utilité, nous pouvons maintenant passer à la prochaine étape : la gestion des médias, ou *assets* (les sons et les images quoi).

**i** *Note : dans les chapitres suivants, nous n'utiliserons pas de namespaces afin de garder une certaine aisance, de plus, vous pourrez ainsi adapter le code source en utilisant votre propre namespace.*

## V - Gestion des médias (ou assets)

*Note.*

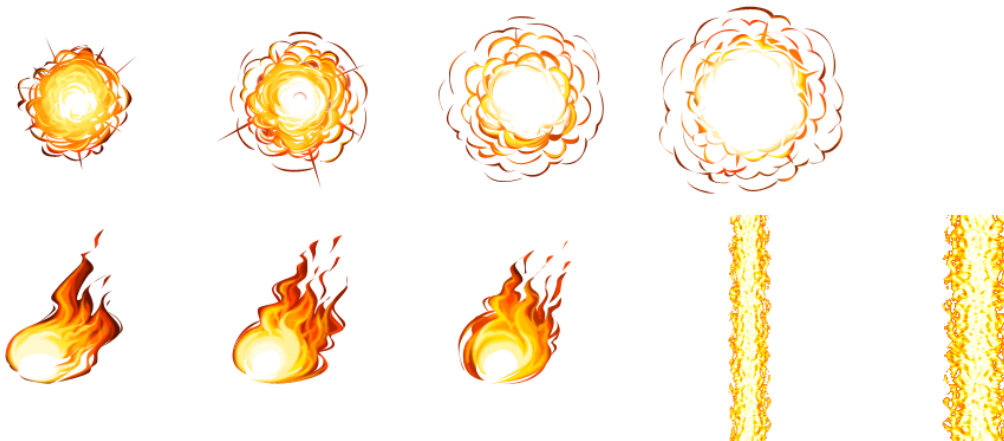
**i** *Précédemment, nous avons convenu que nous utiliserions le mot texture pour désigner une image, car cela correspond à un vocabulaire très utilisé dans le jeu vidéo. Pour les besoins de ce chapitre, nous allons coder une classe nommée `Texture` dont l'une des propriétés sera une image.*

*Par soucis de clarté, nous allons donc réutiliser le mot image pour parler d'une image, le mot texture désignera dorénavant un objet de type `Texture`.*

### V-A - Introduction aux spritesheets

Nous commencerons ce chapitre avec une introduction à la notion de *spritesheet*.

**Une *spritesheet* est une image qui contient les différentes étapes d'une même animation**, par exemple, l'image suivante est une *spritesheet* livrée avec le logiciel RPG Maker représentant une explosion :



À quoi peut-elle bien nous servir, pourquoi ne pas charger une multitude d'images plutôt qu'une seule et s'embarrasser à gérer cela ? Et bien tout simplement parce que charger une seule et même image permet *d'alléger le poids total des assets et d'optimiser les performances d'affichage* (mais ça nous le verrons plus tard).

Sachez juste que si l'on avait découpé l'image ci-dessus en de multiples images, **cela aurait pesé plus lourd** car chaque image possède ses propres **headers** c'est-à-dire quelques octets au tout début du fichier qui contiennent des informations à propos de celui-ci.

De plus, selon le type de fichier, **l'encodage, la compression peuvent représenter une contrainte de poids supplémentaire** que l'on multipliera si l'on a plusieurs fichiers.

Pour finir, sachez qu'un serveur HTTP préférera toujours vous envoyer un seul et même fichier plutôt qu'une multitude de petits fichiers, cela lui demande à chaque fois un temps d'accès au disque dur ainsi que la construction d'une requête HTTP de retour etc. Bref beaucoup de travail en trop pour rien.

Laissons maintenant les *spritesheets*, nous y reviendrons plus tard (au moment de créer des animations) et passons maintenant au chargement et au stockage des images.

## V-B - Gestion des Assets : AssetsLoader et AssetsManager

Nous allons maintenant passer à la gestion de ces images, pour cela nous allons **coder deux classes** très utiles, l'une nous permettra de **charger** une suite d'images, l'autre de **stocker** et de retrouver ces mêmes images (qui, plus tard, seront embarquée dans une classe `Texture`).

### V-B-1 - La classe AssetsLoader

Tout d'abord nous allons charger ces fameuses images et pour cela nous allons créer une classe que l'on nommera `AssetsLoader`, que nous mettrons dans un fichier à part du même nom.

Sa fonction sera de stocker les URL des images, d'y associer un alias, de charger toutes les images et d'appeler une fonction définie par l'utilisateur lorsque le chargement sera complet.

```
AssetsLoader.js
function AssetsLoader(){
    this._loadingList = [];
};

Tomahawk.registerClass( AssetsLoader, "AssetsLoader" );

AssetsLoader._instance = null;
AssetsLoader.getInstance = function(){ // singleton
    if( AssetsLoader._instance == null )
        AssetsLoader._instance = new AssetsLoader();

    return AssetsLoader._instance;
};

AssetsLoader.prototype.onComplete = null;
AssetsLoader.prototype._loadingList = null;
AssetsLoader.prototype._data = null;

AssetsLoader.prototype.getData = function(){
    return this._data;
};

AssetsLoader.prototype.addFile = function(fileURL, fileAlias){
    // on réinitialise les data
    this._data = {};

    // on stocke un objet contenant l'URL et l'alias du fichier que l'on
    // utilisera pour le retrouver
    this._loadingList.push({url:fileURL,alias:fileAlias});
};

AssetsLoader.prototype.load = function(){
    if( this._loadingList.length == 0 ){
        if( this.onComplete ){
            this.onComplete();
        }
    }
    else{
```

### AssetsLoader.js

```
var obj = this._loadingList.shift();
var scope = this;
var image = new Image();
image.onload = function() {
    scope._onLoadComplete(image, obj.alias);
};

image.src = obj.url;
}
};

AssetsLoader.prototype._onLoadComplete = function(image, alias) {
    this._data[alias] = image;
    this.load();
};
```

### sample1.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0, user-scalable=0">
<title></title>
<script type="text/javascript" src="tomahawk/Tomahawk.js"></script>
<script type="text/javascript" src="tomahawk/Utils/AssetsLoader.js"></script>
<script type="text/javascript" src="sample1.js"></script>
<style type="text">
    canvas{
        border: 2px solid black;
        background-color: black;
    }
</style>
</head>
<body>
<canvas id="tomahawk"></canvas>
</body>
</html>
```

### sample1.js

```
/* Point d'entrée de l'application */
function init(){
    Tomahawk.run(); // on démarre la plateforme Tomahawk

    var scope = this;

    AssetsLoader.getInstance().onComplete = onComplete;
    AssetsLoader.getInstance().addFile("perso1.png", "perso1");
    AssetsLoader.getInstance().addFile("perso2.png", "perso2");
    AssetsLoader.getInstance().addFile("perso3.png", "perso3");
    AssetsLoader.getInstance().addFile("ground.png", "ground");
    AssetsLoader.getInstance().load();
}

function onComplete() {
    console.log(AssetsLoader.getInstance().getData());
}

window.onload = init;
```

Nous avons le code source de trois fichiers :

- le fichier `sample1.html`, qui embarque les fichiers `Tomahawk.js`, `sample1.js` et `AssetsLoader.js` ;
- le fichier `AssetsLoader.js` qui contient le code de notre classe ;
- le fichier `sample1.js` qui se contente d'appeler une fonction `init` au chargement de la page.

Comme vous pouvez le constater, nous avons utilisé le **design pattern singleton**, qui consiste à retourner une seule et unique instance de la classe `AssetsLoader`. C'est un choix personnel, **ne vous sentez pas obligé d'y adhérer**, si vous souhaitez pouvoir créer plusieurs instances de la classe `AssetsLoader` ceci est votre droit.

Nous avons également utilisé notre fameuse classe `Tomahawk` ainsi que certaines de ses méthodes. Plus précisément nous avons enregistré la classe `AssetsLoader` à l'aide de la méthode statique `Tomahawk.registerClass()` puis nous avons démarré la plateforme à l'aide de l'appel à la méthode `Tomahawk.run()`.

Même si notre classe `AssetsLoader` stocke nos images, elle n'est pas vraiment faite pour ça, en effet, à chaque fois que nous l'utiliserons pour charger un nouveau groupe d'images, la propriété `_data` est réinitialisée, ce qui rend les données indisponibles.

Ce qu'il nous faudrait donc, c'est une classe dont la tâche serait de stocker et de restituer ces images, ce qui est précisément le cas de la classe `AssetsManager` dont voici le code source :

#### AssetsManager.js

```
function AssetsManager() {
    this._data = {};
};

Tomahawk.registerClass( AssetsManager, "AssetsManager" );

// singleton
AssetsManager._instance = null;
AssetsManager.getInstance = function() {
    if( AssetsManager._instance == null )
        AssetsManager._instance = new AssetsManager();

    return AssetsManager._instance;
};

AssetsManager.prototype._data = null;

AssetsManager.prototype.getData = function() {
    return this._data;
};

AssetsManager.prototype.getDataByAlias = function(alias) {
    if( this._data[alias] )
        return this._data[alias];

    return null;
};

AssetsManager.prototype.addImage = function(image, alias) {
    this._data[alias] = image;
};
```

Voilà, maintenant nous disposons d'une **classe de stockage des assets**, notez que j'ai fais là aussi le choix du singleton, encore une fois ceci est plus une question de goût qu'autre chose, essayez d'adapter le code à votre façon de faire.

Pour avoir un exemple d'utilisation de cette classe, il nous suffit de modifier légèrement le code du fichier `sample1.js`

#### sample.js

```
function onComplete() {
    var data = AssetsLoader.getInstance().getData();
    for( var alias in data ) {
        AssetsManager.getInstance().addImage(data[alias], alias);
    }

    console.log(AssetsManager.getInstance().getDataByAlias("ground"));
}
```

L'on voit que l'on stocke toutes les images au sein de l'`AssetsManager` et que dès que l'on souhaite retrouver une des images, il nous suffit de **passer le nom de son alias** à la méthode `getDataByAlias` de l'objet `AssetsManager`.

Bien, nous savons charger, stocker, **retrouver des images en fonction d'un alias**, il nous faut à présent apprendre à nous en servir. Pour ce faire, je vous propose de passer tout de suite à la création de notre classe `Texture`.

## V-B-2 - La classe Texture

Pourquoi créer une classe `Texture` ? Nous avons déjà des images, nous savons les dessiner de toutes les façons possibles, alors pourquoi s'ennuyer à coder une classe par-dessus ?

Et bien tout simplement parce que cette classe `Texture` contiendra des informations qui nous seront bien utiles plus tard, comme le nom de la texture, l'image associée (qui n'est pas forcément différente de celle d'une autre texture nous y reviendrons plus tard) etc...

C'est une classe dont on pourrait se passer. Concrètement, on pourrait se débrouiller sans elle, mais elle nous facilite beaucoup la vie et plus c'est simple mieux c'est non ?

Voici sans plus tarder le code la classe `Texture` que nous placerons dans un fichier à part :

```
Texture.js
function Texture() {}

Tomahawk.registerClass( Texture, "Texture" );

Texture.prototype.data = null;
Texture.prototype.name = null;
Texture.prototype.rect = null;
```

Comme je vous l'ai dit, **cette classe nous servira uniquement à stocker des données additionnelles**.

Pas de méthodes compliquées, rien que de la donnée. Parfois la simplification d'un code source tient à peu de choses...

Bien, voyons à présent comment nous pouvons nous en servir, reprenons le code de notre fichier `sample1.js` de tout à l'heure.

```
sample1.js
function onComplete() {
    var data = AssetsLoader.getInstance().getData();
    var canvas = document.getElementById('tomahawk');
    var context = canvas.getContext('2d');
    for( var alias in data ){
        AssetsManager.getInstance().addImage(data[alias],alias);
    }

    // on crée une nouvelle texture
    var texture = new Texture();

    // on lui associe l'image dont l'alias est ground
    texture.data = AssetsManager.getInstance().getDataByAlias("ground");

    // on lui donne un nom
    texture.name = "groundTexture";

    // on précise quelle est la portion d'image relatif à cette texture
    // un tableau dont les valeurs représentent les valeurs [x,y,width,height]
    // ici la portion d'image relatif à ma texture correspond à la moitié de
    texture.rect = [0,0,32,22]; // l'image

    context.save();
```

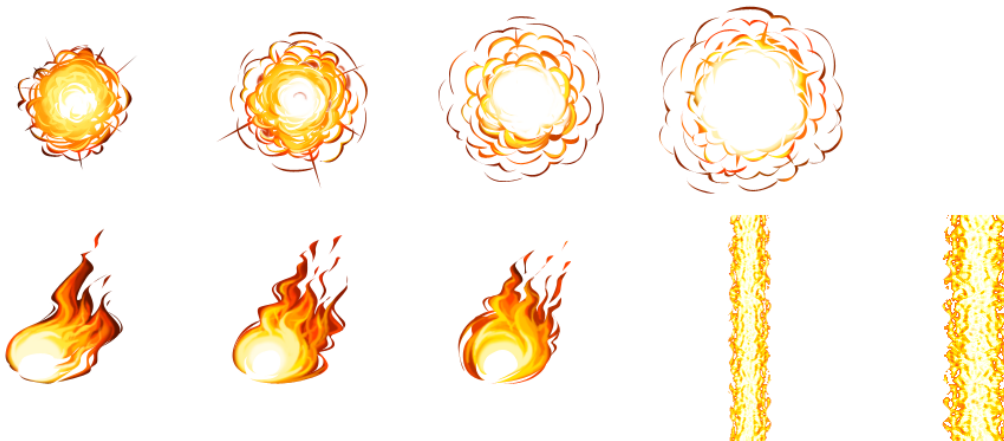
#### sample1.js

```
context.drawImage( texture.data,
    texture.rect[0],
    texture.rect[1],
    texture.rect[2],
    texture.rect[3],
    0,
    0,
    100,
    100 );
context.restore();
}
```

Attardons-nous un peu sur la propriété `rect` de la classe `Texture` voulez-vous. À quoi peut-elle bien nous servir ? Et bien tout l'intérêt de cette propriété réside dans le fait qu'elle nous permet de préciser la portion de l'image à dessiner.

« Oui mais ça, on pouvait déjà le savoir sans avoir à coder une classe pour cela, la preuve on l'a fait tout à l'heure sans utiliser tout ce code supplémentaire ».

Je vois qu'il y en a qui suivent, c'est bien. Et bien à cela je vous répondrais que oui, tout à l'heure c'était plus simple mais moins pratique, car avec la classe que nous avons là, deux `Textures` différentes peuvent partager la même image, il leur suffira de définir une portion d'image différente afin de se distinguer l'une de l'autre. Reprenons notre image de tout à l'heure :



Ici, chaque portion de l'image représente un état de l'explosion, ce qui veut dire que nous pouvons associer un objet `Texture` à chacun de ses états tout en chargeant une seule et même image.

Dans le cas de cette explosion, si je veux afficher le stade 0 de l'explosion, il me suffit de dessiner la texture associée, si je veux afficher le stade 1 puis 2 puis 3 il me suffira de dessiner l'objet `Texture` correspondant.

Nous reparlerons de l'animation plus loin, nous allons nous intéresser maintenant à un autre moyen d'exploiter cette fameuse classe `Texture`.

## V-C - Regrouper toutes les textures, la classe `TextureAtlas`

Comme nous venons de le voir, la classe `Texture` possède un vrai potentiel qu'il nous faut maintenant exploiter, c'est pour cela que nous allons créer une classe nommée `TextureAtlas` dont le rôle sera de créer, de stocker, de restituer des objets de type `Texture`.

#### TextureAtlas.js

```
function TextureAtlas() {
    this._textures = [];
}
```

## TextureAtlas.js

```
Tomahawk.registerClass( TextureAtlas, "TextureAtlas" );

TextureAtlas.prototype._textures = null;
TextureAtlas.prototype.data = null;
TextureAtlas.prototype.name = null;

TextureAtlas.prototype.createTexture = function( name, startX, startY, endX, endY ){
  var texture = new Texture();
  texture.name = name;
  texture.data = this.data;
  texture.rect = [startX, startY, endX, endY];

  this._textures.push(texture);
};

TextureAtlas.prototype.getTextureByName = function( name ){
  var i = this._textures.length;
  var currentTexture = null;
  while( --i > -1 ){
    currentTexture = this._textures[i];
    if( currentTexture.name == name )
      return currentTexture;
  }

  return null;
};

TextureAtlas.prototype.removeTexture = function( name ){
  var texture = this.getTextureByName( name );

  if( texture == null )
    return;

  var index = this._textures.indexOf(texture);
  this._textures.splice(index,1);
};
```

Modifions maintenant le code notre fichier `sample1.js` :

```
function onComplete(){
  var data = AssetsLoader.getInstance().getData();
  var canvas = document.getElementById('tomahawk');
  var context = canvas.getContext('2d');
  for( var alias in data ){
    AssetsManager.getInstance().addImage( data[alias], alias );
  }

  // on crée un nouvel atlas
  var atlas = new TextureAtlas();

  // on lui associe une image qui sera celle partagée par toutes les textures stockée en son sein
  atlas.data = AssetsManager.getInstance().getDataByAlias( "ground" );

  // on crée deux textures différentes, portant un nom différent, ayant chacune la même image
  // mais pas les mêmes portions d'image associées

  atlas.createTexture( "texture_1", 0,0,32,43);
  atlas.createTexture( "texture_2", 32,0,32,43);

  var texture = atlas.getTextureByName("texture_1");

  context.save();

  // on dessine la première texture
  context.drawImage( texture.data,
    texture.rect[0],
    texture.rect[1],
    texture.rect[2],
    texture.rect[3],
```



```

        0,
        0,
        100,
        100 );

texture = atlas.getTextureByName("texture_2");

// puis la deuxième
context.drawImage(    texture.data,
                      texture.rect[0],
                      texture.rect[1],
                      texture.rect[2],
                      texture.rect[3],
                      110,
                      0,
                      100,
                      100 );

// en gros, nous avons dessiné les deux moitiés de l'image
// sur une largeur et une hauteur de 100 pixels
// les deux moitiés étant séparées de 10 pixels
context.restore();
}

```

Dans l'ordre, nous avons d'abord créé un objet de type `TextureAtlas`, nous avons associé une image à sa propriété `data` puis nous avons créé deux textures à l'aide la méthode `createTexture()` de la classe `TextureAtlas`.

Cette méthode crée un nouvel objet de type `Texture` et se charge d'initialiser ses propriétés avec les bonnes informations, l'objet `Texture` est ensuite stocké et pourra être retrouvé à l'aide de la méthode `getTextureByName()` de la classe `TextureAtlas`.

Pour finir, nous dessinons les deux textures sur le canvas.

Il ne nous reste plus qu'à implémenter le code qui nous permettra de stocker des objets de type `Texture` et `TextureAtlas` au sein d'un objet de type `AssetsManager` et nous aurons terminé ce chapitre.

Voici le code de la classe `AssetsManager`. Notez que nous avons cette fois-ci bien fait la distinction entre tous les types de données différents :

```

function AssetsManager() {
    this._images = {};
    this._atlases = {};
    this._textures = {};
};

Tomahawk.registerClass( AssetsManager, "AssetsManager" );

// singleton
AssetsManager._instance = null;
AssetsManager.getInstance = function() {
    if( AssetsManager._instance == null )
        AssetsManager._instance = new AssetsManager();

    return AssetsManager._instance;
};

AssetsManager.prototype._images = null;
AssetsManager.prototype._atlases = null;
AssetsManager.prototype._textures = null;

// images
AssetsManager.prototype.getImages = function() {
    return this._images;
};

AssetsManager.prototype.getImageByAlias = function(alias) {

```

```

    if( this._images[alias] )
        return this._images[alias];

    return null;
};

AssetsManager.prototype.addImage = function(image, alias){
    this._images[alias] = image;
};

//atlases
AssetsManager.prototype.addAtlas = function(atlas, alias){
    this._atlases[alias] = atlas;
};

AssetsManager.prototype.getAtlases = function(){
    return this._atlases;
};

AssetsManager.prototype.getAtlasByAlias = function(alias){
    if( this._atlases[alias] )
        return this._atlases[alias];

    return null;
};

//textures
AssetsManager.prototype.addTexture = function(texture, alias){
    this._textures[alias] = texture;
};

AssetsManager.prototype.getTextures = function(){
    return this._textures;
};

AssetsManager.prototype.getTextureByAlias = function(alias){
    if( this._textures[alias] )
        return this._textures[alias];

    return null;
};

```

Maintenant que nous savons comment gérer les *assets*, nous allons apprendre à créer une structure pour les manipuler.

## VI - Les bases de l'affichage

### VI-A - Structure arborescente et DisplayList

Commençons par introduire la notion de DisplayList, qu'est-ce qu'une DisplayList et à quoi cela sert-il ?

La DisplayList (traduire liste d'affichage) est, comme son nom l'indique, une structure de données contenant l'ensemble des informations à afficher. Pour les développeurs *ActionScript*, cette notion est évidente car intrinsèque au langage. Pour les développeurs Web, cela se rapproche beaucoup du DOM.

Il s'agit d'une structure arborescente possédant, de par ce fait, une racine et un nombre non fini de feuilles ou ramifications. Si on prend l'exemple du DOM, le code source suivant est un bel exemple de structure arborescente :

```

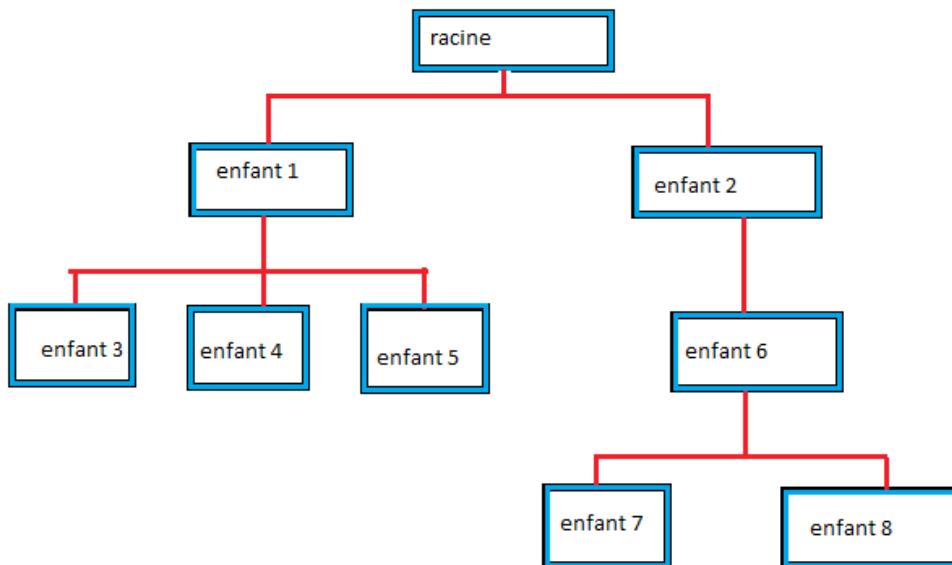
<div>
  <p>enfant numero 1</p>
  <p>enfant numero 2</p>
</div>

```

Ici, la balise `<div>` est le parent des balises `<p>` contenues en son sein, dans un vocabulaire algorithmique, et plus particulièrement dans le cas des structures arborescentes, on parle de **nœuds** ou de **parents** et de **feuilles** ou **d'enfants**.

L'exemple plus haut est assez simple, mais une structure arborescente, de par sa nature, n'est jamais finie, comprendre par là que l'on peut toujours ajouter des enfants à parent.

Voici un schéma d'une autre structure, celle-ci part d'un nœud racine pour ensuite déployer ses enfants qui eux-mêmes auront des enfants etc.



Pour en revenir à notre `DisplayList`, sa structure sera similaire, c'est-à-dire que nous partirons d'un objet d'affichage de départ pour ensuite progresser récursivement afin de parcourir chaque enfant de l'arbre.

Nous allons donc commencer par créer la classe d'affichage de base, la classe `DisplayObject`.

## VI-B - Le point de départ : La classe `DisplayObject`

Nous avons vu précédemment les différentes techniques de dessin possibles, puis nous avons créé une classe `Texture` destinée à être manipulée par un objet de type `Context`.

Notre classe `DisplayObject` doit reprendre l'ensemble de ces éléments tout en gardant un esprit « objet », c'est-à-dire qu'il nous faudra pouvoir la manipuler aisément. Le modèle de l'API du langage ActionScript 3 est plutôt clair, nous allons donc nous caler dessus. Notre classe `DisplayObject` doit pouvoir gérer l'affichage et le stockage des données relatives à l'affichage de ce que l'on souhaite dessiner.

Voici une première version de notre classe `DisplayObject`, notez que sa méthode `draw()` dessine un carré rouge, ce carré rouge n'est présent que dans le but d'avoir un code prêt à tester, gardez à l'esprit qu'il ne devrait pas être présent, notre classe `DisplayObject` étant destinée à être une classe de base pour tous les objets d'affichage qui hériteront de ses fonctionnalités.

```

function DisplayObject() {}

Tomahawk.registerClass( DisplayObject, "DisplayObject" );

DisplayObject.prototype.name = null;
  
```

```

DisplayObject.prototype.parent = null;
DisplayObject.prototype.x = 0;
DisplayObject.prototype.y = 0;
DisplayObject.prototype.scaleX = 1;
DisplayObject.prototype.scaleY = 1;
DisplayObject.prototype.width = 0;
DisplayObject.prototype.height = 0;
DisplayObject.prototype.rotation = 0;
DisplayObject.prototype.alpha = 1;

DisplayObject._toRadians = Math.PI / 180;

DisplayObject.prototype.render = function( context ){
  context.save(); // d'abord on sauvegarde le contexte

  //puis on applique les transformations, comme nous avons
  // dans les chapitres précédents
  context.translate(this.x, this.y);
  context.rotate(this.rotation * DisplayObject._toRadians);
  context.scale( this.scaleX, this.scaleY );
  context.globalAlpha = this.alpha;

  // puis on dessine
  this.draw(context);

  // et enfin on restaure le contexte sauvegardé plus haut
  context.restore();
};

DisplayObject.prototype.draw = function(context){
  // nous dessinons un rectangle rouge
  context.beginPath();
  context.fillStyle = "red";
  context.fillRect(0, 0, this.width, this.height);
  context.fill();
}

```

Voici une première version de notre classe `DisplayObject`, si vous souhaitez la tester, c'est assez simple : il vous suffit de créer un objet de type `DisplayObject` et d'appeler sa méthode `render()` en lui passant le contexte en paramètre. Expérimentez, changez les propriétés de votre objet, son alpha, son `scaleX`, sa rotation etc. Vous verrez, on s'y fait rapidement.

Nous allons maintenant passer à une classe un peu plus intéressante, car elle sera destinée à dessiner des textures : la classe `Bitmap`.

## VI-C - Enfin des textures : La classe Bitmap

Nous allons à présent afficher des textures, pour cela nous allons coder une classe que nous nommerons `Bitmap`. Cette classe héritera de la classe `DisplayObject`, ce qui fait qu'elle disposera de toutes les fonctionnalités de celle-ci en plus des siennes propres.

Sans plus tarder, voici le code :

```

function Bitmap() {}

Tomahawk.registerClass( Bitmap, "Bitmap" );
Tomahawk.extend( "Bitmap", "DisplayObject" );

Bitmap.prototype.texture = null;

Bitmap.prototype.draw = function( context ){
  var rect = this.texture.rect;
  var data = this.texture.data;

  context.drawImage( data, rect[0], rect[1], rect[2], rect[3], 0, 0, this.width, this.height );
};

```

Notez que l'on utilise bien les méthodes statiques de notre classe `Tomahawk` pour résoudre cette problématique d'héritage, nous avons donc bien hérité des propriétés et des méthodes de la classe `DisplayObject`.

Nous avons également ajouté une propriété à notre classe `Bitmap`, que nous avons nommée `texture()`.

Celle-ci, comme on peut s'en douter, représente un objet de type `Texture`.

La seule méthode à avoir été redéfinie est la méthode `draw()`, ici, elle nous permet de dessiner notre texture, de la même façon que dans le chapitre précédent.

Voici un exemple de code qui nous permettra d'afficher un objet de type `Bitmap`.

#### sample1.js

```

/* Point d'entrée de l'application */
function init() {
    // on démarre la plateforme Tomahawk puis on charge les fichiers
    Tomahawk.run();
    AssetsLoader.getInstance().onComplete = onComplete;
    AssetsLoader.getInstance().addFile("ground.png", "ground");
    AssetsLoader.getInstance().load();
}

function onComplete() {
    var data = AssetsLoader.getInstance().getData();
    var canvas = document.getElementById('tomahawk');
    var context = canvas.getContext('2d');

    for( var alias in data ) {
        AssetsManager.getInstance().addImage(data[alias], alias);
    }

    // on crée un nouvel atlas
    var atlas = new TextureAtlas();

    // on lui associe une image qui sera celle partagée par toutes les textures stockée en son sein
    atlas.data = AssetsManager.getInstance().getImageByAlias("ground");

    // on crée deux textures différentes, portant un nom différent, ayant chacune la même image
    // mais pas les mêmes portions d'image associées
    atlas.createTexture( "texture_1", 0,0,64,43);

    var texture = atlas.getTextureByName("texture_1"); // on retrouve notre texture
    var bmp = new Bitmap(); // on crée un nouvel objet de type Bitmap
    bmp.texture = texture; // on y associe la texture
    bmp.width = 64; // on définit la largeur
    bmp.height = 43; //... puis la hauteur
    bmp.render(context); // et enfin on dessine le tout
}

```

Nous pouvons à présent manipuler des objets de type `Bitmap`, nous allons donc passer à la prochaine étape, le côté « imbrication » des objets d'affichage.

## VI-D - Objets imbriqués : La classe `DisplayObjectContainer`

Comme nous avons pu le voir précédemment, la notion de `DisplayList` implique une certaine récursivité, une « imbrication » en théorie infinie. L'implémentation d'un tel concept peut sembler compliquée mais il n'en est rien, en effet il nous « suffira » de créer un `DisplayObject` qui possède la capacité d'avoir des enfants.

Ainsi, lorsqu'on appellera la méthode `render()` de ce `DisplayObject`, l'ensemble de ses enfants seront également dessinés et si nous choisissons de ne pas dessiner cet objet, ses enfants ne le seront pas non plus.

Cette classe, nous l'appellerons `DisplayObjectContainer`, voici son code source :

```
function DisplayObjectContainer() {this._construct();}

Tomahawk.registerClass( DisplayObjectContainer, "DisplayObjectContainer" );
Tomahawk.extend( "DisplayObjectContainer", "DisplayObject" );

DisplayObjectContainer.prototype.children = null;

DisplayObjectContainer.prototype._construct = function() {
  this.children = new Array();
};

DisplayObjectContainer.prototype.addChild = function(child) {
  if( child.parent ) {
    child.parent.removeChild(child);
  }

  child.parent = this;
  this.children.push(child);
};

DisplayObjectContainer.prototype.getChildAt = function (index) {
  return this.children[index];
};

DisplayObjectContainer.prototype.getChildByName = function(name) {
  var children = this.children;
  var i = children.length;

  while( --i > -1 ) {
    if( children[i].name == name )
      return children[i];
  }

  return null;
};

DisplayObjectContainer.prototype.addChildAt = function(child, index) {
  var children = this.children;
  var tab1 = this.children.slice(0,index);
  var tab2 = this.children.slice(index);
  this.children = tab1.concat([child]).concat(tab2);

  child.parent = this;
};

DisplayObjectContainer.prototype.removeChildAt = function(index) {
  var child = this.children[index];
  if( child )
    child.parent = null;
  this.children.splice(index,1);
};

DisplayObjectContainer.prototype.removeChild = function(child) {
  var index = this.children.indexOf(child);

  if( index > -1 )
    this.children.splice(index,1);

  child.parent = null;
};

DisplayObjectContainer.prototype.draw = function( context ) {
  var children = this.children;
  var i = 0;
  var max = children.length;
  var child = null;

  for( ; i < max; i++ ) {
    child = children[i];
    child.render(context);
  }
}
```

```
};
```

Comme vous pouvez le constater, cette classe hérite de `DisplayObject`, elle possède donc les propriétés et les méthodes d'un objet de type `DisplayObject`, avec cependant quelques petites choses en plus.

Notamment cette propriété `children` qui, comme son nom l'indique, va servir à stocker des enfants et à boucler dessus au sein de la méthode `draw()` qui pour le coup a été redéfinie.

Les méthodes `addChild()` et `addChildAt()` sont là pour ajouter un enfant, soit à la fin de la liste des enfants, soit à un index précis, les enfants étant dessinés dans l'ordre.

Les méthodes `removeChild()` et `removeChildAt()` sont là pour enlever un enfant à notre `DisplayObjectContainer`, soit un enfant précis dans le cas de `removeChild()` soit un enfant se situant à un index précis dans le cas de `removeChildAt()`.

La méthode `getChildAt()` nous renvoie l'enfant situé à l'index passé en paramètre.

La méthode `getChildByName()` nous renvoie le premier enfant dont le nom est égal à celui passé en paramètre.

Voici un exemple de code mettant en pratique notre `DisplayObjectContainer` :

#### sample1.js

```
/* Point d'entrée de l'application */
function init(){
    // on démarre la plateforme Tomahawk puis on charge les fichiers
    Tomahawk.run();
    AssetsLoader.getInstance().onComplete = onComplete;
    AssetsLoader.getInstance().addFile("ground.png","ground");
    AssetsLoader.getInstance().load();
}

function onComplete(){
    var data = AssetsLoader.getInstance().getData();
    var canvas = document.getElementById('tomahawk');
    var context = canvas.getContext('2d');

    for( var alias in data ){
        AssetsManager.getInstance().addImage(data[alias],alias);
    }

    // on crée un nouvel atlas
    var atlas = new TextureAtlas();

    // on lui associe une image qui sera celle partagée par toutes les textures stockée en son sein
    atlas.data = AssetsManager.getInstance().getImageByAlias("ground");

    // on crée deux textures différentes, portant un nom différent, ayant chacune la même image
    // mais pas les mêmes portions d'image associées
    atlas.createTexture( "texture_1", 0,0,64,43);

    var container = new DisplayObjectContainer(); // on crée un objet de type DisplayObjectContainer
    var texture = atlas.getTextureByName("texture_1"); // on retrouve notre texture
    var bmp = new Bitmap(); // on crée un nouvel objet de type Bitmap
    bmp.texture = texture; // on y associe la texture
    bmp.width = 64; // on définit la largeur
    bmp.height = 43; //... puis la hauteur

    container.addChild(bmp); // et on l'ajoute à la liste des enfants du container

    // on recommence l'opération tout en changeant les coordonnées du deuxième enfant
    bmp = new Bitmap();
    bmp.texture = texture;
    bmp.width = 64;
```

### sample1.js

```

bmp.height = 43;
bmp.x = 100;
bmp.y = 100;

container.addChild(bmp);

// et on appelle la méthode render du DisplayObjectContainer
container.render(context);
}

```

Comme vous pouvez le constater, les deux enfants, des objets de type `Bitmap`, sont dessinés à l'écran et tout ça en appelant la méthode `render()` de leur parent.

## VI-E - Racine de la DisplayList : classe Stage

Nous allons à présent passer à la dernière étape essentielle de la `DisplayList` : **la racine**.

Dans le graphique de tout à l'heure, on peut voir que toute structure arborescente possède une racine, un point de départ.

Par définition, ce point de départ ne possède pas de parent et si nous explorons chacun de ses enfants de manière récursive, nous parcourons l'ensemble de l'arbre.

À l'heure actuelle, nous pourrions déjà nous contenter de ce que nous avons, dans l'exemple précédent, la variable `container` possédait deux enfants, des objets de type `Bitmap`, mais rien ne nous aurait empêché de lui ajouter un enfant (ou plusieurs) de type `DisplayObjectContainer` qui lui-même aurait contenu des enfants etc. Cette fameuse variable `container` devenait la racine de notre arbre.

Cependant, dans le cadre de notre moteur, il serait bon que notre racine possède quelques fonctionnalités supplémentaires, comme le fait d'appeler elle-même sa méthode `render()` à un intervalle de temps régulier, déclenchant ainsi la première étape vers l'animation de nos objets en implémentant la notion de *frame*.

Il serait bon que notre racine gère également d'autres petites choses, comme le fait d'effacer le canvas à chaque *frame*, nous donner le nombre de frames par seconde etc. C'est ce que nous allons faire en introduisant la classe `Stage`.

```

function Stage() {

    // on se sert de la fonction de base "webkitRequestAnimationFrame"
    // fourni par l'API html5
    window.requestAnimationFrame = (function() {

        return window.requestAnimationFrame || //Chromium
               window.webkitRequestAnimationFrame || //Webkit
               window.mozRequestAnimationFrame || //Mozilla Geko
               window.oRequestAnimationFrame || //Opera Presto
               window.msRequestAnimationFrame || //IE Trident?
               function(callback, element) { //Fallback function
                   window.setTimeout(callback, 10);
               }

    })();

    this._construct();
}

Tomahawk.registerClass( Stage, "Stage" );
Tomahawk.extend( "Stage", "DisplayObjectContainer" );

Stage._instance = null;
Stage.getInstance = function() {
    if( Stage._instance == null )

```



```

    Stage._instance = new Stage();

    return Stage._instance;
};

Stage.prototype._lastTime = 0;
Stage.prototype._frameCount = 0;
Stage.prototype._fps = 0;
Stage.prototype._canvas = null;
Stage.prototype._context = null;
Stage.prototype._debug = false;

Stage.prototype.init = function(canvas) {
    this._canvas = canvas;
    this._context = canvas.getContext("2d");
    this._enterFrame();
};

Stage.prototype._enterFrame = function() {

    var curTime = new Date().getTime();
    var scope = this;

    this._frameCount++;

    if( curTime - this._lastTime >= 1000 ) {
        this._fps = this._frameCount;
        this._frameCount = 0;
        this._lastTime = curTime;
    }

    this._context.clearRect(0,0,this._canvas.width,this._canvas.height);
    this._context.save();
    this.render(this._context);
    this._context.restore();

    if( this._debug == true ) {
        this._context.save();
        this._context.beginPath();
        this._context.fillStyle = "black";
        this._context.fillRect(0,0,100,30);
        this._context.fill();
        this._context.fillStyle = "red";
        this._context.font = 'italic 20pt Calibri';
        this._context.fillText("fps: "+this._fps, 0,30);
        this._context.restore();
    }

    window.requestAnimationFrame(
        function() {
            scope._enterFrame();
        }
    );
};

Stage.prototype.getCanvas = function() {
    return this._canvas;
};

Stage.prototype.getContext = function() {
    return this._context;
};

Stage.prototype.getFPS = function() {
    return this._fps;
};

Stage.prototype.setDebug = function( debug ) {
    this._debug = debug;
};

```

Comme vous pouvez le voir, notre classe `Stage` hérite de `DisplayObjectContainer`, ainsi nous pourrons lui ajouter des enfants. On, peut également constater quelques ajouts de fonctionnalités comme un compteur de FPS ou l'utilisation d'une méthode de l'API HTML5 `requestAnimationFrame()`.

Cette dernière est utilisée de préférence à la méthode `setTimeout()`. En effet, le navigateur va la déclencher de lui-même après que toutes les opérations de dessin sont terminées, optimisant ainsi l'affichage et donc la fluidité de notre application.

J'ai encore fait ici le choix du *singleton*, en effet, la racine est unique par définition, il ne sert donc à rien de créer de multiples instances de `Stage`, mais si vous trouvez une utilité quelconque à le faire, n'hésitez pas, expérimentez par vous-mêmes.

Comme d'habitude, voici le code d'exemple :

```
/* Point d'entrée de l'application */
function init(){
    // on démarre la plateforme Tomahawk puis on charge les fichiers
    Tomahawk.run();
    AssetsLoader.getInstance().onComplete = onComplete;
    AssetsLoader.getInstance().addFile("ground.png", "ground");
    AssetsLoader.getInstance().load();
}

function onComplete(){
    var data = AssetsLoader.getInstance().getData();
    var canvas = document.getElementById('tomahawk');

    // on initialise la racine en lui envoyant la référence vers le canvas
    Stage.getInstance().init(canvas);

    for( var alias in data ){
        AssetsManager.getInstance().addImage(data[alias], alias);
    }

    // on crée un nouvel atlas
    var atlas = new TextureAtlas();

    // on lui associe une image qui sera celle partagée par toutes les textures stockée en son sein
    atlas.data = AssetsManager.getInstance().getImageByAlias("ground");

    // on crée deux textures différentes, portant un nom différent, ayant chacune la même image
    // mais pas les mêmes portions d'image associées
    atlas.createTexture( "texture_1", 0,0,64,43);

    var texture = atlas.getTextureByName("texture_1"); // on retrouve notre texture
    var bmp = new Bitmap(); // on crée un nouvel objet de type Bitmap
    bmp.texture = texture; // on y associe la texture
    bmp.width = 64; // on définit la largeur
    bmp.height = 43; //... puis la hauteur

    Stage.getInstance().addChild(bmp); // on ajoute l'enfant à la racine

    // on recommence l'opération tout en changeant les coordonnées du deuxième enfant
    bmp = new Bitmap();
    bmp.texture = texture;
    bmp.width = 64;
    bmp.height = 43;
    bmp.x = 100;
    bmp.y = 100;

    Stage.getInstance().addChild(bmp); // on l'ajoute aussi
    Stage.getInstance().setDebug(true); // on souhaite voir le fps
}
```

Beaucoup plus pratique n'est-ce pas ?

À l'avenir, nous devrons seulement initialiser la racine en lui passant le canvas en paramètre. Nous n'avons plus à retrouver le contexte et à l'envoyer à toutes les instances de `DisplayObject` ou `DisplayObjectContainer`, la racine se charge de tout, et à chaque frame s'il vous plaît !

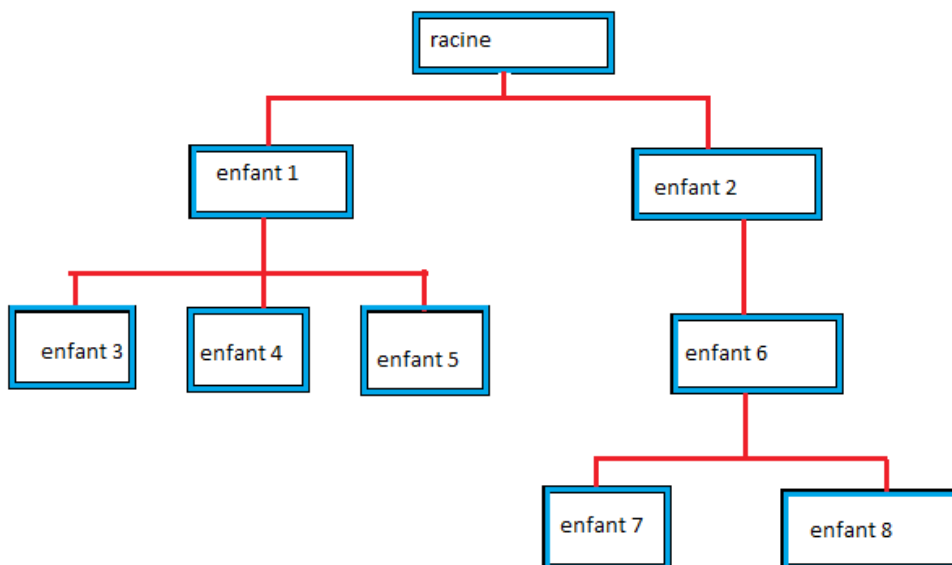
Nous allons maintenant enchaîner sur les transformations imbriquées, comment répercuter la rotation, la translation, le *scale* d'un parent sur l'ensemble de ses enfants et ainsi de suite ?

## VII - Manipuler les objets : transformations et calculs matriciels.

### VII-A - Le problème des transformations imbriquées.

Le problème des transformations imbriquées est très simple : il faut que les transformations d'un parent soient répercutées sur ses enfants et ce, de manière récursive et infinie. Concrètement, cela demande de connaître en permanence l'état actuel de l'ensemble des transformations appliquées sur un objet ET ajouter les siennes propres.

Exemple :



Admettons que dans le schéma suivant :

```

racine.x = 0
  enfant 1.x = 10
    enfant 3.x = 0
    enfant 4.x = 10
    enfant 5.x = 20
  
```

Si l'on part du principe que l'on doit répercuter les transformations du parent sur l'enfant, alors les coordonnées **réelles** en x de chacun de ces `DisplayObject` seront :

```

racine.x = 0
  enfant 1.x = 10
    enfant 3.x = 10
    enfant 4.x = 20
    enfant 5.x = 30
  
```

Cela paraît simple dit comme cela, mais si on ne sait pas comment le gérer proprement, cela devient vite un enfer, surtout qu'il faut prendre en compte les autres types de transformations comme le *scale* ou la rotation.

Et si je vous disais que nous avons déjà la solution, que le code que nous avons écrit règle ce problème ?

Si je vous disais que sans le savoir, vous avez utilisé l'outil mathématique adéquat ?

Que nous avons rejoints Morpheus et Néo..., que nous avons utilisé des matrices ?

Mais qu'est-ce qu'une matrice ?

## VII-B - Les matrices, pour quoi faire ?

Une matrice est un outil de mathématique permettant de résoudre certaines situations, elles sont, avec les quaternions, l'un des pans essentiels de la programmation graphique, bien que ces derniers soient surtout utilisés pour la 3D.

En notation mathématique, schématiquement cela ressemble à un tableau à deux dimensions pouvant contenir le nombre de colonnes et de lignes dont on a besoin.

Voici la représentation usuelle d'une matrice identité possédant quatre lignes et quatre colonnes :

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Sachez seulement que l'une des propriétés principales de la matrice est sa commutativité, c'est-à-dire que les transformations qu'on lui applique se cumulent et que le résultat des opérations dépend de l'ordre dans lequel on les exécute.

Ça ne vous rappelle rien ? Les méthodes de transformation de l'objet `context` bien sûr !

## VII-C - Comment utiliser les matrices ?

Les méthodes de transformation de l'objet `context` nous permettent de manipuler, à notre insu jusque là, une matrice 3x3 (trois lignes, trois colonnes) et de lui appliquer des transformations.

Ce qui est génial, c'est que comme les matrices sont commutatives, les transformations que l'on applique se cumulent et que l'on peut à tout moment revenir à un état antérieur d'une matrice.

Les méthodes `context.save()` et `context.restore()`, sauvegardent et restituent (entre autres) l'état de la matrice de transformations à un instant `t`.

Qui fait que lorsque j'appelle la méthode `render()` de mon `DisplayObjectContainer`, je réalise l'opération suivante :

- je sauvegarde l'état actuel de la matrice ;
- j'applique mes transformations au `context` ;
- je boucle sur mes enfants, j'appelle la méthode `render()` de chacun et j'applique les transformations des enfants au contexte etc., ces transformations sont cumulées à celle de mon parent ;
- je restaure le contexte sauvegardé à l'étape 1.

L'opération est récursive et est déjà implémentée. Essayez, vous verrez que tout fonctionne.

## VII-D - Implémentation des matrices dans le moteur.

Les matrices sont présentes nativement dans le moteur, le seul défaut de l'API, c'est qu'elle ne nous permet pas de récupérer l'état actuel de la matrice et de le stocker quelque part, ce qui nous sera utile par la suite.

Je vous propose donc d'implémenter les matrices d'une autre façon, à l'aide d'une classe `Matrix2D` dont voici le code source.

Cette classe `Matrix2D` est une petite adaptation de la classe `Matrix2D` utilisée par le moteur `CreateJs` (<http://www.createjs.com>).

```
function Matrix2D(a, b, c, d, tx, ty){
    this.initialize(a, b, c, d, tx, ty);
}

// static public properties:

/**
 * An identity matrix, representing a null transformation.
 * @property identity
 * @static
 * @type Matrix2D
 * @readonly
 */
Matrix2D.prototype.identity = null; // set at bottom of class definition.

/**
 * Multiplier for converting degrees to radians. Used internally by Matrix2D.
 * @property DEG_TO_RAD
 * @static
 * @final
 * @type Number
 * @readonly
 */
Matrix2D.DEG_TO_RAD = Math.PI/180;

// public properties:
/**
 * Position (0, 0) in a 3x3 affine transformation matrix.
 * @property a
 * @type Number
 */
Matrix2D.prototype.a = 1;

/**
 * Position (0, 1) in a 3x3 affine transformation matrix.
 * @property b
 * @type Number
 */
Matrix2D.prototype.b = 0;

/**
 * Position (1, 0) in a 3x3 affine transformation matrix.
 * @property c
 * @type Number
 */
Matrix2D.prototype.c = 0;

/**
 * Position (1, 1) in a 3x3 affine transformation matrix.
 * @property d
 * @type Number
 */
Matrix2D.prototype.d = 1;
```

```

/**
 * Position (2, 0) in a 3x3 affine transformation matrix.
 * @property tx
 * @type Number
 */
Matrix2D.prototype.tx = 0;

/**
 * Position (2, 1) in a 3x3 affine transformation matrix.
 * @property ty
 * @type Number
 */
Matrix2D.prototype.ty = 0;

// constructor:
/**
 * Initialization method. Can also be used to reinitialize the instance.
 * @method initialize
 * @param {Number} [a=1] Specifies the a property for the new matrix.
 * @param {Number} [b=0] Specifies the b property for the new matrix.
 * @param {Number} [c=0] Specifies the c property for the new matrix.
 * @param {Number} [d=1] Specifies the d property for the new matrix.
 * @param {Number} [tx=0] Specifies the tx property for the new matrix.
 * @param {Number} [ty=0] Specifies the ty property for the new matrix.
 * @return {Matrix2D} This instance. Useful for chaining method calls.
 */
Matrix2D.prototype.initialize = function(a, b, c, d, tx, ty) {
  this.a = (a == null) ? 1 : a;
  this.b = b || 0;
  this.c = c || 0;
  this.d = (d == null) ? 1 : d;
  this.tx = tx || 0;
  this.ty = ty || 0;
  return this;
};

// public methods:
/**
 * Concatenates the specified matrix properties with this matrix. All parameters are required.
 * @method prepend
 * @param {Number} a
 * @param {Number} b
 * @param {Number} c
 * @param {Number} d
 * @param {Number} tx
 * @param {Number} ty
 * @return {Matrix2D} This matrix. Useful for chaining method calls.
 */
Matrix2D.prototype.prepend = function(a, b, c, d, tx, ty) {
  var tx1 = this.tx;
  if (a != 1 || b != 0 || c != 0 || d != 1) {
    var a1 = this.a;
    var c1 = this.c;
    this.a = a1*a+this.b*c;
    this.b = a1*b+this.b*d;
    this.c = c1*a+this.d*c;
    this.d = c1*b+this.d*d;
  }
  this.tx = tx1*a+this.ty*c+tx;
  this.ty = tx1*b+this.ty*d+ty;
  return this;
};

/**
 * Appends the specified matrix properties with this matrix. All parameters are required.
 * @method append
 * @param {Number} a
 * @param {Number} b
 * @param {Number} c
 * @param {Number} d

```

```

* @param {Number} tx
* @param {Number} ty
* @return {Matrix2D} This matrix. Useful for chaining method calls.
**/
Matrix2D.prototype.append = function(a, b, c, d, tx, ty) {
  var a1 = this.a;
  var b1 = this.b;
  var c1 = this.c;
  var d1 = this.d;

  this.a = a*a1+b*c1;
  this.b = a*b1+b*d1;
  this.c = c*a1+d*c1;
  this.d = c*b1+d*d1;
  this.tx = tx*a1+ty*c1+this.tx;
  this.ty = tx*b1+ty*d1+this.ty;
  return this;
};

/**
* Prepends the specified matrix with this matrix.
* @method prependMatrix
* @param {Matrix2D} matrix
* @return {Matrix2D} This matrix. Useful for chaining method calls.
**/
Matrix2D.prototype.prependMatrix = function(matrix) {
  this.prepend(matrix.a, matrix.b, matrix.c, matrix.d, matrix.tx, matrix.ty);
  return this;
};

/**
* Appends the specified matrix with this matrix.
* @method appendMatrix
* @param {Matrix2D} matrix
* @return {Matrix2D} This matrix. Useful for chaining method calls.
**/
Matrix2D.prototype.appendMatrix = function(matrix) {
  this.append(matrix.a, matrix.b, matrix.c, matrix.d, matrix.tx, matrix.ty);
  return this;
};

/**
* Generates matrix properties from the specified display object transform properties, and prepends them with this matrix.
* For example, you can use this to generate a matrix from a display object: var mtx = new Matrix2D();
* mtx.prependTransform(o.x, o.y, o.scaleX, o.scaleY, o.rotation);
* @method prependTransform
* @param {Number} x
* @param {Number} y
* @param {Number} scaleX
* @param {Number} scaleY
* @param {Number} rotation
* @param {Number} skewX
* @param {Number} skewY
* @param {Number} regX Optional.
* @param {Number} regY Optional.
* @return {Matrix2D} This matrix. Useful for chaining method calls.
**/
Matrix2D.prototype.prependTransform = function(x, y, scaleX, scaleY, rotation, skewX, skewY, regX, regY) {
  if (rotation%360) {
    var r = rotation*Matrix2D.DEG_TO_RAD;
    var cos = Math.cos(r);
    var sin = Math.sin(r);
  } else {
    cos = 1;
    sin = 0;
  }

  if (regX || regY) {
    // append the registration offset:
    this.tx -= regX; this.ty -= regY;
  }
}

```

```

    if (skewX || skewY) {
      // TODO: can this be combined into a single prepend operation?
      skewX *= Matrix2D.DEG_TO_RAD;
      skewY *= Matrix2D.DEG_TO_RAD;
      this.prepend(cos*scaleX, sin*scaleX, -sin*scaleY, cos*scaleY, 0, 0);
      this.prepend(Math.cos(skewY), Math.sin(skewY), -Math.sin(skewX), Math.cos(skewX), x, y);
    } else {
      this.prepend(cos*scaleX, sin*scaleX, -sin*scaleY, cos*scaleY, x, y);
    }
    return this;
  };

  /**
   * Generates matrix properties from the specified display object transform properties, and appends them with this
   * For example, you can use this to generate a matrix from a display object: var mtx = new Matrix2D();
   * mtx.appendTransform(o.x, o.y, o.scaleX, o.scaleY, o.rotation);
   * @method appendTransform
   * @param {Number} x
   * @param {Number} y
   * @param {Number} scaleX
   * @param {Number} scaleY
   * @param {Number} rotation
   * @param {Number} skewX
   * @param {Number} skewY
   * @param {Number} regX Optional.
   * @param {Number} regY Optional.
   * @return {Matrix2D} This matrix. Useful for chaining method calls.
   */
  Matrix2D.prototype.appendTransform = function(x, y, scaleX, scaleY, rotation, skewX, skewY, regX,
    regY) {

    if (rotation%360) {
      var r = rotation*Matrix2D.DEG_TO_RAD;
      var cos = Math.cos(r);
      var sin = Math.sin(r);
    } else {
      cos = 1;
      sin = 0;
    }

    if (skewX || skewY) {
      // TODO: can this be combined into a single append?
      skewX *= Matrix2D.DEG_TO_RAD;
      skewY *= Matrix2D.DEG_TO_RAD;
      this.append(Math.cos(skewY), Math.sin(skewY), -Math.sin(skewX), Math.cos(skewX), x, y);
      this.append(cos*scaleX, sin*scaleX, -sin*scaleY, cos*scaleY, 0, 0);
    } else {
      this.append(cos*scaleX, sin*scaleX, -sin*scaleY, cos*scaleY, x, y);
    }

    if (regX || regY) {
      // prepend the registration offset:
      this.tx -= regX*this.a+regY*this.c;
      this.ty -= regX*this.b+regY*this.d;
    }
    return this;
  };

  /**
   * Applies a rotation transformation to the matrix.
   * @method rotate
   * @param {Number} angle The angle in radians. To use degrees, multiply by <code>Math.PI/180</code>.
   * @return {Matrix2D} This matrix. Useful for chaining method calls.
   */
  Matrix2D.prototype.rotate = function(angle) {
    var cos = Math.cos(angle);
    var sin = Math.sin(angle);

    var a1 = this.a;
    var c1 = this.c;
    var tx1 = this.tx;

```



```

    this.a = a1*cos-this.b*sin;
    this.b = a1*sin+this.b*cos;
    this.c = c1*cos-this.d*sin;
    this.d = c1*sin+this.d*cos;
    this.tx = tx1*cos-this.ty*sin;
    this.ty = tx1*sin+this.ty*cos;
    return this;
  };

  /**
   * Applies a skew transformation to the matrix.
   * @method skew
   * @param {Number} skewX The amount to skew horizontally in degrees.
   * @param {Number} skewY The amount to skew vertically in degrees.
   * @return {Matrix2D} This matrix. Useful for chaining method calls.
   */
  Matrix2D.prototype.skew = function(skewX, skewY) {
    skewX = skewX*Matrix2D.DEG_TO_RAD;
    skewY = skewY*Matrix2D.DEG_TO_RAD;
    this.append(Math.cos(skewY), Math.sin(skewY), -Math.sin(skewX), Math.cos(skewX), 0, 0);
    return this;
  };

  /**
   * Applies a scale transformation to the matrix.
   * @method scale
   * @param {Number} x The amount to scale horizontally
   * @param {Number} y The amount to scale vertically
   * @return {Matrix2D} This matrix. Useful for chaining method calls.
   */
  Matrix2D.prototype.scale = function(x, y) {
    this.a *= x;
    this.d *= y;
    this.c *= x;
    this.b *= y;
    this.tx *= x;
    this.ty *= y;
    return this;
  };

  /**
   * Translates the matrix on the x and y axes.
   * @method translate
   * @param {Number} x
   * @param {Number} y
   * @return {Matrix2D} This matrix. Useful for chaining method calls.
   */
  Matrix2D.prototype.translate = function(x, y) {
    this.tx += x;
    this.ty += y;
    return this;
  };

  /**
   * Sets the properties of the matrix to those of an identity matrix (one that applies a null transformation).
   * @method identity
   * @return {Matrix2D} This matrix. Useful for chaining method calls.
   */
  Matrix2D.prototype.identity = function() {
    this.a = this.d = 1;
    this.b = this.c = this.tx = this.ty = 0;
    return this;
  };

  /**
   * Inverts the matrix, causing it to perform the opposite transformation.
   * @method invert
   * @return {Matrix2D} This matrix. Useful for chaining method calls.
   */
  Matrix2D.prototype.invert = function() {
    var a1 = this.a;
    var b1 = this.b;

```

```

    var c1 = this.c;
    var d1 = this.d;
    var tx1 = this.tx;
    var n = a1*d1-b1*c1;

    this.a = d1/n;
    this.b = -b1/n;
    this.c = -c1/n;
    this.d = a1/n;
    this.tx = (c1*this.ty-d1*tx1)/n;
    this.ty = -(a1*this.ty-b1*tx1)/n;
    return this;
  };

  /**
   * Returns true if the matrix is an identity matrix.
   * @method
   * isIdentity
   * @return {Boolean}
   */
  Matrix2D.prototype.isIdentity = function() {
    return this.tx == 0 && this.ty == 0 && this.a == 1 && this.b == 0 && this.c == 0 && this.d == 1;
  };

  /**
   * Transforms a point according to this matrix.
   * @method transformPoint
   * @param {Number} x The x component of the point to transform.
   * @param {Number} y The y component of the point to transform.
   * @param {Point | Object} [pt] An object to copy the result into. If omitted a generic object with x/
   y properties will be returned.
   * @return {Point} This matrix. Useful for chaining method calls.
   */
  Matrix2D.prototype.transformPoint = function(x, y, pt) {
    pt = pt || {};
    pt.x = x*this.a+y*this.c+this.tx;
    pt.y = x*this.b+y*this.d+this.ty;
    return pt;
  };

  /**
   * Decomposes the matrix into transform properties (x, y, scaleX, scaleY, and rotation). Note that this these val
   * may not match the transform properties you used to generate the matrix, though they will produce the same visu
   * results.
   * @method decompose
   * @param {Object} target The object to apply the transform properties to. If null, then a new object will be ret
   * @return {Matrix2D} This matrix. Useful for chaining method calls.
   */
  Matrix2D.prototype.decompose = function(target) {

    if (target == null) { target = {}; }
    target.x = this.tx;
    target.y = this.ty;
    target.scaleX = Math.sqrt(this.a * this.a + this.b * this.b);
    target.scaleY = Math.sqrt(this.c * this.c + this.d * this.d);

    var skewX = Math.atan2(-this.c, this.d);
    var skewY = Math.atan2(this.b, this.a);

    if (skewX == skewY) {
      target.rotation = skewY/Matrix2D.DEG_TO_RAD;
      if (this.a < 0 && this.d >= 0) {
        target.rotation += (target.rotation <= 0) ? 180 : -180;
      }
      target.skewX = target.skewY = 0;
    } else {
      target.skewX = skewX/Matrix2D.DEG_TO_RAD;
      target.skewY = skewY/Matrix2D.DEG_TO_RAD;
    }
    return target;
  };

```

```

/**
 * Reinitializes all matrix properties to those specified.
 * @method reinitialize
 * @param {Number} [a=1] Specifies the a property for the new matrix.
 * @param {Number} [b=0] Specifies the b property for the new matrix.
 * @param {Number} [c=0] Specifies the c property for the new matrix.
 * @param {Number} [d=1] Specifies the d property for the new matrix.
 * @param {Number} [tx=0] Specifies the tx property for the new matrix.
 * @param {Number} [ty=0] Specifies the ty property for the new matrix.
 * @return {Matrix2D} This matrix. Useful for chaining method calls.
 */
Matrix2D.prototype.reinitialize = function(a, b, c, d, tx, ty) {
  this.initialize(a,b,c,d,tx,ty);
  return this;
};

/**
 * Copies all properties from the specified matrix to this matrix.
 * @method copy
 * @param {Matrix2D} matrix The matrix to copy properties from.
 * @return {Matrix2D} This matrix. Useful for chaining method calls.
 */
Matrix2D.prototype.copy = function(matrix) {
  return this.reinitialize(matrix.a, matrix.b, matrix.c, matrix.d, matrix.tx, matrix.ty);
};

/**
 * Returns a clone of the Matrix2D instance.
 * @method clone
 * @return {Matrix2D} a clone of the Matrix2D instance.
 */
Matrix2D.prototype.clone = function() {
  return (new Matrix2D()).copy(this);
};

/**
 * Returns a string representation of this object.
 * @method toString
 * @return {String} a string representation of the instance.
 */
Matrix2D.prototype.toString = function() {
  return "[Matrix2D (a="+this.a+" b="+this.b+" c="+this.c+" d="+this.d+" tx="+this.tx+" ty="+this.ty+")]";
};
Matrix2D.identity = new Matrix2D();

```

Maintenant nous disposons de notre propre classe pour manipuler des matrices, voici son implémentation au sein de la classe `DisplayObject` :

```

// nouvelle méthode render
DisplayObject.prototype.render = function( context ){
  this.update();

  if( this.visible == false )
    return;

  var mat = this.matrix;
  context.save();
  context.globalAlpha = this.alpha;
  context.transform(mat.a,mat.b,mat.c,mat.d,mat.tx,mat.ty);
  this.draw(context);
  context.restore();
};

//Méthode update, qui nous permet d'actualiser la propriété this.matrix de la classe DisplayObject;
DisplayObject.prototype.update = function() {
  var current = this;
  var mat = this.matrix || new Matrix2D();

  mat.appendTransform(

```

```
this.x,  
    this.y,  
    this.scaleX,  
    this.scaleY,  
    this.rotation,  
    this.skewX,  
    this.skewY,  
    this.pivotX,  
    this.pivotY  
);  
};
```

À présent, nous disposons d'une classe nous permettant de calculer l'état réel de l'ensemble des transformations appliquées à un `DisplayObject`, nous utilisons la méthode `transform()` de l'objet `context` qui prend justement le nombre de paramètre suffisant pour redéfinir les composantes d'une matrice 3x3.

Ainsi, nous allons pouvoir gérer pas mal de choses, notamment plus tard les collisions, mais pour l'heure nous allons faire une pause et passer à un chapitre plus léger : le modèle événementiel.

## VIII - Modèle événementiel et design pattern

### VIII-A - Pourquoi utiliser des événements ?

Commençons déjà par définir ce que nous entendons par événement. Dans notre moteur il s'agira d'un objet, vecteur d'information, que l'on pourra lancer et attraper en fonction de qui est intéressé par cet événement.

Cela fonctionne un peu comme une *newsletter* : si vous êtes intéressé par l'activité d'un site Web, il est fort probable que vous soyez abonné à leur *newsletter*, votre email est alors stocké quelque part et un mail vous est envoyé au moment où une nouvelle est publiée sort avec des informations à son sujet.

Un événement dans notre moteur fonctionne exactement de la même manière : je notifie quelque part mon intérêt pour un événement précis, et lorsque cet événement se produit, je l'attrape et je m'en sers (ou non).

Pour ceux qui connaissent déjà, c'est exactement la même chose que le modèle événementiel natif du JavaScript ou de l'AS3.

Quel est l'intérêt d'utiliser un modèle événementiel, de dispatcher, d'écouter des événements ?

Quelques exemples :

- déclencher une action au clic sur un `DisplayObject` ;
- appliquer un effet au *rollover* ou au *rollout* sur un `DisplayObject` ;
- détecter quand un `DisplayObject` a été ajouté ou enlevé de la `DisplayList` ;
- déclencher une fonction à chaque *frame*.

En gros, cela nous permet de gérer facilement l'interactivité de notre application.

Mais comment implémenter cela dans notre moteur ?

### VIII-B - Comment gérer un modèle événementiel, le design pattern observer

Pour ceux qui ne connaissent pas, le design pattern *Observer* consiste à ajouter/enlever des « écouteurs » (*listener* en anglais) à un objet. Ces écouteurs attendent un type d'événement précis et chaque fois que l'objet sur lequel on a posé l'écouteur « lance » (ou *dispatch*) un événement, l'ensemble des écouteurs attendant un événement de ce type sont notifiés.

Concrètement, lorsqu'un écouteur est notifié, une fonction de rappel (*callback*) définie au moment de l'ajout de l'écouteur est appelée, en règle générale l'événement lui est passé en paramètre.

## VIII-C - Implémentation sur les objets d'affichage, la classe EventDispatcher et la classe Event

### EventDispatcher.js

```
function EventDispatcher() {}

Tomahawk.registerClass( EventDispatcher, "EventDispatcher" );

EventDispatcher.prototype.parent = null;
EventDispatcher.prototype._listeners = null;

EventDispatcher.prototype.addEventListener = function( type, scope, callback, useCapture ){
  this._listeners = this._listeners || [];
  var obj = {};
  obj.type = type;
  obj.scope = scope;
  obj.callback = callback;
  obj.useCapture = useCapture;
  this._listeners.push(obj);
};

EventDispatcher.prototype.hasEventListener = function(type) {
  if( this._listeners == null )
    return false;

  var obj = {};
  var i = this._listeners.length;
  while( --i > -1 )
  {
    obj = this._listeners[i];
    if( obj.type == type )
      return true;
  }
};

EventDispatcher.prototype.dispatchEvent = function( event ){
  this._listeners = this._listeners || [];
  var obj = {};
  var i = this._listeners.length;

  if( event.target == null )
    event.target = this;

  event.currentTarget = this;

  while( --i > -1 ){
    obj = this._listeners[i];

    if( obj.type == event.type ){
      if( event.target != this && obj.useCapture == false ){
        continue;
      }

      obj.callback.apply( obj.scope, [event] );
    }
  }

  if( event.bubbles == true && this.parent != null && this.parent.dispatchEvent ){
    this.parent.dispatchEvent(event);
  }
};

EventDispatcher.prototype.removeEventListener = function( type, scope, callback, useCapture ){
  var listener = this.getEventListener(type);

  while( listener != null ){
    var obj = {};
```

### EventDispatcher.js

```

var i = this._listeners.length;
var arr = [];

while( --i > -1 ){
    obj = this._listeners[i];
    if( obj.type != listener.type || obj.scope != scope || obj.callback != callback ||
obj.useCapture != useCapture )
        return arr.push(obj);
}

this._listeners = arr;
listener = this.getEventListener(type);
}
};

```

### Event.js

```

function Event(type, bubbles, cancelable){
    this.type = type;
    this.cancelable = cancelable;
    this.bubbles = bubbles;
}

Tomahawk.registerClass( Event, "Event" );

Event.prototype.type = null;
Event.prototype.bubbles = false;
Event.prototype.cancelable = true;
Event.prototype.data = null;
Event.prototype.target = null;
Event.prototype.currentTarget = null;

Event.prototype.stopPropagation = function(){
    if( this.cancelable == true )
        this.bubbles = false;
};

//constantes

Event.ADDED = "added";
Event.ADDED_TO_STAGE = "addedToStage";
Event.ENTER_FRAME = "enterFrame";
Event.REMOVED = "removed";
Event.REMOVED_FROM_STAGE = "removedFromStage";

```

Maintenant, nous pouvons faire hériter notre classe `DisplayObject` de la classe `EventDispatcher`, par conséquent tous nos `DisplayObject` seront désormais des `EventDispatcher`.

Pour ce faire, on doit ajouter la ligne suivante à notre classe `DisplayObject` :

```
Tomahawk.extend( "DisplayObject", "EventDispatcher" );
```

Comme vous pouvez le constater, notre classe `Event` possède quelques constantes prédéfinies.

`Event.ADDED` par exemple. Cette constante définit le type de l'événement, ce type d'événement sera dispatché par un `DisplayObject` au moment où il est ajouté à un `DisplayObjectContainer`.

Les quatre méthodes suivantes de notre classe `DisplayObjectContainer` seront alors modifiées :

```

DisplayObjectContainer.prototype.addChild = function(child){
    if( child.parent ){
        child.parent.removeChild(child);
    }

    child.parent = this;
    this.children.push(child);
}

```

```

        child.dispatchEvent( new Event(Event.ADDED, true, true) );
    };

    DisplayObjectContainer.prototype.addChildAt = function(child, index) {
        var children = this.children;
        var tab1 = this.children.slice(0, index);
        var tab2 = this.children.slice(index);
        this.children = tab1.concat([child]).concat(tab2);

        child.parent = this;
        child.dispatchEvent( new Event(Event.ADDED, true, true) );
    };

    DisplayObjectContainer.prototype.removeChildAt = function(index) {
        var child = this.children[index];
        if( child )
            child.parent = null;
        this.children.splice(index, 1);
        child.dispatchEvent( new Event(Event.REMOVED, true, true) );
    };

    DisplayObjectContainer.prototype.removeChild = function(child) {
        var index = this.children.indexOf(child);

        if( index > -1 )
            this.children.splice(index, 1);

        child.parent = null;
        child.dispatchEvent( new Event(Event.REMOVED, true, true) );
    };

```

Voilà, maintenant vous pouvez déclencher une action dès qu'un `DisplayObjectContainer` ajoute ou enlève un enfant.

Exemple :

```

function onAdded(event) {
    console.log(event.type);
}

var container = new DisplayObjectContainer();
var disp = new DisplayObject();

disp.addEventListener(Event.ADDED, null, onAdded);
container.addChild(disp);

```

Comme vous avez pu le constater, la méthode `addEventListener` possède un paramètre `useCapture`. Défini à `true`, ce paramètre vous donne la possibilité d'attraper les événements dispatchés par les enfants de l'objet sur lequel on a posé l'écouteur. En gros, cela veut dire que nous aurions pu reproduire l'exemple ci-dessus en posant l'écouteur directement sur le conteneur.

Je vous invite à essayer en guise d'exercice. Observez également le code de la classe `EventDispatcher` et regardez comment cette fonctionnalité est gérée.

Maintenant que nous sommes capables de manipuler les événements, nous pouvons nous intéresser à la détection des actions de l'utilisateur, par exemple le clic.

## IX - Collisions et événements utilisateurs

### IX-A - Comment détecter les événements souris ?

La détection des événements souris est indispensable dès que l'on souhaite interagir avec l'utilisateur. Le fait de bouger sa souris, de cliquer ou double cliquer ou même de passer sa souris par dessus un objet sont des événements qui méritent d'être notifiés. Quoi de mieux pour ça que notre modèle événementiel ?

Mais comment savoir sur quel objet l'utilisateur a cliqué ? Existe-t-il une méthode efficace nous permettant de savoir cela et si oui est-ce compliqué ? En premier lieu, sachez qu'il existe bien une méthode et que cette dernière, pour peu que l'on ait la théorie suffisante, est assez simple à implémenter. Nous allons donc commencer par la théorie.

### IX-B - Théorie des collisions, les formules

Quand un utilisateur clique, nous obtenons un point dont nous souhaitons savoir si les coordonnées `x` et `y` se situent bien « à l'intérieur » d'un `DisplayObject` quelconque. Mathématiquement parlant, un `DisplayObject` est un rectangle dont les coordonnées des quatre coins sont aisément calculables à l'aide des propriétés `x`, `y`, `width` et `height` de l'objet en question.

La formule pour savoir si un point se situe à l'intérieur ou non d'un rectangle se présente sous cette forme :

```
function isInRect(x,y, rectX, rectY, rectWidth, rectHeight){
    if(
        x > rectX + rectWidth ||
        y > rectY + rectHeight ||
        x < rectX ||
        y < rectY
    ){
        return false;
    }
    else{
        return true;
    }
}
```

Dans le cas présent, cela reste simple, mais dans le cas de notre classe `DisplayObject`, cela est un peu moins évident.

### IX-C - Implémentation d'une méthode `hitTest` sur un `DisplayObject`

Nous allons implémenter une méthode nommée `hitTest` sur notre classe `DisplayObject`, celle-ci aura pour rôle de nous dire si le point qu'on lui passe en paramètre entre en collision avec l'objet en question.

Seulement voilà, le point que l'on va passer à l'origine aura pour coordonnées celles du `Stage`, c'est-à-dire qu'elles ne correspondront pas au repère de données locales, car nos `DisplayObject` sont potentiellement transformés.

C'est pour pallier ce soucis que nous avons implémenté des matrices !

Dans l'utilisation que l'on en fait, les matrices nous servent à cumuler des transformations (celles des multiples parents et de l'enfant) pour arriver au résultat sur le stage, celui que l'on voit !

Et bien l'opération inverse est possible, partir d'un résultat sur le stage et arriver à une transformation dans l'espace local ! Pour cela, il vous suffira d'utiliser la matrice inverse de celle que vous avez utilisé et il se trouve que justement, la classe `Matrix2D` que nous avons utilisée dispose d'une méthode qui lui permet de s'inverser elle-même !

Nous allons donc pouvoir d'ores et déjà implémenter deux nouvelles méthodes très utiles sur notre classe `DisplayObject`.



- Une méthode `localToGlobal` qui prend en paramètre un point dans le système de coordonnées local du `DisplayObject` et qui le convertit en point dans le système de coordonnées global du `Stage`.
- Une méthode `globalToLocal`, qui fait exactement l'inverse de la fonction précédente, c'est celle-ci que nous allons utiliser dans notre méthode `hitTest` pour convertir le point passé en paramètre. Ainsi nous pourrions vérifier si ce point est compris dans le rectangle du `DisplayObject`.

La classe `Point` :

```
function Point(x,y) {
    this.x = x, this.y = y
}

Tomahawk.registerClass( Point, "Point" );

Point.prototype.x = 0;
Point.prototype.y = 0;
```

Voici les méthodes à ajouter à notre classe `DisplayObject` :

```
DisplayObject.prototype.getConcatenedMatrix = function() {
    var current = this;
    var mat = new Matrix2D();

    while( current != null ){
        current.update();
        mat.prependMatrix(current.matrix );
        current = current.parent;
    }

    this._concatenedMatrix = mat;
    return this._concatenedMatrix;
};

DisplayObject.prototype.localToGlobal = function(x,y) {
    var matrix = this.getConcatenedMatrix();
    var pt = matrix.transformPoint(x,y);
    return new Point(pt.x,pt.y);
};

DisplayObject.prototype.globalToLocal = function(x,y) {
    var matrix = this.getConcatenedMatrix().clone().invert();
    var pt = matrix.transformPoint(x,y);
    return new Point(pt.x,pt.y);
};

DisplayObject.prototype.hitTest = function(x,y) {
    if( this.matrix == null )
        this.update();

    var pt1 = this.globalToLocal(x,y);

    if( pt1.x < 0 || pt1.x > this.width || pt1.y < 0 || pt1.y > this.height )
        return false;
    else
        return true;
};
```

Maintenant, nous pouvons détecter si un `DisplayObject` est cliqué ou non. Encore faut-il déterminer si de `DisplayObject` ne se situe pas « en dessous » d'un autre. Ce dernier prendra alors automatiquement l'événement souris et l'objet se situant en dessous ne devra rien dispatcher.

Pour arriver à nos fins, nous devons repartir dans notre bonne vieille classe `Stage`.

Le `Stage` étant la racine de notre `DisplayList`, nous avons donc accès à l'ensemble des `DisplayObject`.

Il suffit alors de parcourir l'ensemble de l'arbre de manière récursive à rebours et en partant du dernier objet dans la liste des enfants.

Au premier objet répondant au `hitTest` et possédant un écouteur pour le type d'événement en cours, on stoppe le parcours de l'arbre et on fait en sorte que l'objet dispatche cet événement.

Nous pouvons procéder ainsi pour tous les événements souris.

Voici le code à ajouter à la classe `Stage` :

```
Stage.prototype.init = function(canvas) {
    var scope = this;
    var callback = function(event) {
        scope._mouseHandler(event);
    };

    this._canvas = canvas;
    this._context = canvas.getContext("2d");
    this._canvas.addEventListener("click", callback);
    this._canvas.addEventListener("mousemove", callback);
    this._canvas.addEventListener("mousedown", callback);
    this._canvas.addEventListener("mouseup", callback);
    this._enterFrame();
};

Stage.prototype._mouseHandler = function(event) {
    var bounds = this._canvas.getBoundingClientRect();
    var x = event.clientX - bounds.left;
    var y = event.clientY - bounds.top;
    var activeChild = this._getMouseObjectUnder(x, y, this);
    var mouseEvent = null;
    var i = 0;
    this.mouseX = x;
    this.mouseY = y;

    if( event.type == "mousemove" && this._lastActiveChild != activeChild ) {
        if( activeChild != null ) {
            mouseEvent = MouseEvent.fromNativeMouseEvent(event, true, true, x, y);
            mouseEvent.type = MouseEvent.ROLL_OVER;
            activeChild.dispatchEvent(mouseEvent);
        }

        if( this._lastActiveChild != null ) {
            mouseEvent = MouseEvent.fromNativeMouseEvent(event, true, true, x, y);
            mouseEvent.type = MouseEvent.ROLL_OUT;
            this._lastActiveChild.dispatchEvent(mouseEvent);
        }
    }
    else {
        if( activeChild != null ) {
            mouseEvent = MouseEvent.fromNativeMouseEvent(event, true, true, x, y);
            activeChild.dispatchEvent(mouseEvent);
        }
    }

    this._lastActiveChild = activeChild;
};

Stage.prototype._getMouseObjectUnder = function(x, y, container) {
    var under = null;
    var children = container.children;
    var i = children.length;
    var child = null;

    while( --i > -1 ) {
        child = children[i];
```

```

        if( child.children ){
            under = this._getMouseEventUnder(x,y,child);
            if( under != null )
                return under;
        }
        else if( child.mouseEnabled == true && child.hitTest(x,y) == true ){
            return child;
        }
    }

    return null;
};

```

Et le code la classe `MouseEvent`, qui étend notre classe `Event` de base :

```

function MouseEvent() {}

Tomahawk.registerClass( MouseEvent, "MouseEvent" );
Tomahawk.extend( "MouseEvent", "Event" );

function MouseEvent(type, bubbles, cancelable){
    this.type = type;
    this.cancelable = cancelable;
    this.bubbles = bubbles;
}

MouseEvent.fromNativeMouseEvent = function(event,bubbles,cancelable,x,y){
    var type = "";
    var msevent = null;

    switch( event.type ){
        case "click": type = MouseEvent.CLICK; break;
        case "mousemove": type = MouseEvent.MOUSE_MOVE; break;
        case "mouseup": type = MouseEvent.MOUSE_UP; break;
        case "mousedown": type = MouseEvent.MOUSE_DOWN; break;
    }

    msevent = new MouseEvent( type,bubbles,cancelable );
    msevent.stageX = x;
    msevent.stageY = y;
    return msevent;
};

MouseEvent.CLICK          = "click";
MouseEvent.ROLL_OVER     = "rollOver";
MouseEvent.ROLL_OUT      = "rollOut";
MouseEvent.MOUSE_MOVE    = "mouseMove";
MouseEvent.MOUSE_DOWN    = "mouseUp";
MouseEvent.MOUSE_UP      = "mouseDown";

```

Nous sommes donc capables de dispatcher des événements souris. Je vous laisse découvrir les détails de l'implémentation pour certains types d'événements (comme le *rollover* et le *rollout*). Maintenant nous allons passer aux animations.

## X - Les animations

Comment animer ? Nous avons déjà abordé brièvement le sujet en parlant des *spritesheets*, nous allons maintenant approfondir le sujet. Le but est d'arriver à dessiner une succession d'images avec une fluidité de 24 images par seconde minimum (seuil à partir duquel l'œil humain ne distingue plus d'effet de saccade).

Nous avons déjà une classe `Bitmap`, qui nous permet de dessiner une texture.

Ce qu'il nous faudrait, c'est une classe comme celle-ci mais qui changerait de texture à chaque frame, déclenchant l'effet d'animation désiré.

## X-A - Créer une classe MovieClip permettant de jouer une animation.

La classe `Stage` nous permet déjà d'écouter un événement qui est dispatché à chaque frame, il nous suffit de souscrire un écouteur et de boucler sur toutes les textures que l'on a ajoutées à notre objet.

Voici le code de la classe `MovieClip` :

```
function MovieClip(){
    this._frames = new Array();
}

Tomahawk.registerClass( MovieClip, "MovieClip" );
Tomahawk.extend( "MovieClip", "Bitmap" );

MovieClip.prototype._frames = null;
MovieClip.prototype.currentFrame = 0;
MovieClip.prototype._enterFrameHandler = null;

MovieClip.prototype._enterFrameHandler = function(event){
    this.currentFrame++;
    if( this.currentFrame >= this._frames.length )
        this.currentFrame = 0;

    if( this._frames[this.currentFrame] ){
        this.texture = this._frames[this.currentFrame];
    }
};

MovieClip.prototype.setFrame = function( frameIndex, texture ){
    this._frames[frameIndex] = texture;
};

MovieClip.prototype.play = function(){
    Stage.getInstance().addEventListener(Event.ENTER_FRAME, this, this._enterFrameHandler);
};

MovieClip.prototype.stop = function(){
    Stage.getInstance().removeEventListener(Event.ENTER_FRAME, this, this._enterFrameHandler);
};
```

Comme vous pouvez le constater, nous disposons :

- d'une méthode `setFrame()`, qui prend en paramètre l'index de la frame et la texture correspondant à la frame en question ;
- d'une méthode `play()`, qui nous permet de lancer l'animation ;
- d'une méthode `stop()`, qui nous permet de stopper l'animation
- d'une propriété `currentFrame`, qui indique l'index de la frame en cours ;
- d'une méthode privée `_enterFrameHandler()`, qui boucle sur toutes les textures dont on dispose.

Voici une idée d'exercice simple : maîtriser le fps propre à chaque `MovieClip` c'est-à-dire que vous pourrez animer un `MovieClip` à 24 fps et un autre à 60 fps par exemple.

## XI - Les filtres

### XI-A - Base des filtres avec la balise canvas

Appliquer un filtre est une opération plutôt triviale avec canvas, en effet les étapes sont assez simples et peu nombreuses :

- on récupère les pixels du canvas à l'aide de la méthode `context.getImageData()` ;
- on manipule les pixels en bouclant dessus ;

- enfin, on réinjecte les pixels à l'aide de la méthode `context.putImageData()`.

Ainsi, le code suivant vous permettra d'appliquer une filtre de nuance de gris sur votre objet canvas :

```
function grayscale(canvas,context){
    var pixels = context.getImageData(0,0,canvas.width,canvas.height);
    var data = pixels.data;

    for (var i=0; i<data.length; i+=4){
        var r = data[i];
        var g = data[i+1];
        var b = data[i+2];
        var v = 0.2126*r + 0.7152*g + 0.0722*b;
        data[i] = data[i+1] = data[i+2] = v;
    }

    context.putImageData (canvas,context);
}
```

Essayez le code suivant, tentez vos propres variations, je vous y encourage vivement !

Maintenant, voyons comment implémenter à un `DisplayObject` en particulier.

Le problème est qu'un filtre s'applique à qui a déjà été dessiné et donc, par défaut, sur tous les `DisplayObject` dessinés avant celui sur lequel on souhaite appliquer le filtre.

Qu'à cela ne tienne, il existe une technique très simple pour contourner le problème :

- créer une nouvelle instance de canvas qui servira de tampon ainsi que le contexte associé ;
- dessiner notre `DisplayObject` dessus, sans aucune transformation ;
- appliquer les filtres ;
- puis dessiner le tampon sur le canvas originel.

Voici l'implémentation de ces étapes sur notre classe `DisplayObject` :

```
DisplayObject.prototype.filters = null; // ajout d'une nouvelle propriété, un tableau de filtres

// code de la méthode qui implémente nos différentes étapes pour l'application des filtres

DisplayObject.prototype._applyFilters = function(){
    var canvas = document.createElement("canvas");
    var context = canvas.getContext("2d");
    canvas.width = this.width;
    canvas.height = this.height;

    this.draw(context);
    var i = 0;
    var max = this.filters.length;
    var filter = null;
    for( ; i < max; i++){
        filter = this.filters[i];
        filter.apply(canvas,context);
    }

    return canvas;
};

DisplayObject.prototype.render = function( context ){
    this.update();

    if( this.visible == false )
        return;

    var mat = this.matrix;
```

```
context.save();
context.globalAlpha = this.alpha;
context.transform(mat.a,mat.b,mat.c,mat.d,mat.tx,mat.ty);

if( this.filters != null ){
    // on appelle une nouvelle méthode _applyFilters
    var buffer = this._applyFilters();
    context.drawImage(canvas, 0, 0, buffer.width, buffer.height );
}
else{
    this.draw(context);
}

context.restore();
};
```

Nous avons vu comment implémenter le code sur la classe `DisplayObject`, mais si vous faites bien attention, vous verrez que la méthode `_applyFilters()` boucle sur un tableau d'objets.

Chacun de ces objets est supposé disposer d'une méthode `apply()`, nous allons dès à présent implémenter la classe qui nous permettra d'obtenir ces objets, la classe `PixelFilter`.

## XI-B - Implémentation de la classe `PixelFilter`

La classe `PixelFilter` sera notre classe de base pour les filtres, elle devra disposer :

- d'une méthode `apply()` qui prend deux paramètres, une référence vers le canvas et une référence vers le contexte de ce canvas ;
- d'une méthode `process()` qui sera appelée par la méthode `apply()`, c'est cette méthode que les classes filles surchargeront et dans laquelle le traitement des pixels se fera ;
- d'une méthode `getPixels()` qui nous permettra de récupérer les pixels du canvas ;
- d'une méthode `setPixels()` qui nous permettra d'éditer les pixels du canvas.

### `PixelFilter`

```
function PixelFilter() {}
Tomahawk.registerClass( PixelFilter, "PixelFilter" );

PixelFilter.prototype._canvas = null;
PixelFilter.prototype._context = null;

PixelFilter.prototype.getPixels = function() {
    return this._context.getImageData(0,0,this._canvas.width,this._canvas.height);
};

PixelFilter.prototype.setPixels = function(pixels) {
    context.putImageData(pixels,0,0);
};

PixelFilter.prototype.process = function() {
    //code de notre filtre ici
};

PixelFilter.prototype.apply = function(canvas,context) {
    this._canvas = canvas;
    this._context = context;
    this.process();
};
```

Et en cadeau, voici une classe nommée `GrayScaleFilter`, qui hérite de notre classe `PixelFilter` :

### `GrayScaleFilter`

```
function GrayScaleFilter() {}
Tomahawk.registerClass( GrayScaleFilter, "GrayScaleFilter" );
Tomahawk.extend( GrayScaleFilter, "PixelFilter" );
```

## GrayScaleFilter

```
GrayScaleFilter.prototype.process = function() {  
    var pixels = this.getPixels();  
    var data = pixels.data;  
  
    for (var i=0; i<data.length; i+=4) {  
        var r = data[i];  
        var g = data[i+1];  
        var b = data[i+2];  
        var v = 0.2126*r + 0.7152*g + 0.0722*b;  
        data[i] = data[i+1] = data[i+2] = v;  
    }  
  
    this.setPixels(pixels);  
};
```

## XII - Conclusion et remerciements

Voilà, nous avons passé en revue l'ensemble des notions qui vous seront nécessaires pour construire un moteur d'affichage « *Flash Like* » avec l'API HTML5 de la balise `<canvas>`, je vous encourage à tester vos propres implémentations, à vous créer vous-mêmes vos propres exercices.

Les connaissances que vous avez acquises dans ce cours sont transversales, c'est-à-dire que vous pourrez les réutiliser dans d'autres langages.

En tout cas, c'était un réel plaisir de passer ce temps en votre compagnie et j'espère sincèrement que vous avez apprécié cette première expérience dans le monde de la programmation des moteurs graphiques.

C'est à Paris que je vous retrouverai lors de mes sessions de formation chez Mediabox, dont les programmes sont les suivants.

### XII-A - Formations HTML, HTML5, CSS & canvas

- **Formation HTML & CSS, des bases à la maîtrise.**

Découvrez le programme d'initiation HTML et CSS et créez des sites internet aux normes du Web.

- **Formation HTML5 / CSS3.**

Découvrez toutes les nouveautés du HTML5 (nouvelles balises, microdonnées, meilleur référencement, vidéo et audio sans Flash, ...) et du CSS3 (sélecteurs, graphisme avancé, gestion des médias, transformations, animations, media queries, responsive design, ...)

- **Formation HTML5 & Canvas.**

Apprenez à construire votre propre moteur d'affichage à l'aide des toutes nouvelles fonctionnalités du HTML5 et de la balise `<canvas>`.

- **Formation Responsive Web Design.**

Maîtrisez le Responsive Web Design et préparez vos sites pour qu'ils soient véritablement multiplateformes : iPhone, iPad, tablettes, télé, PC, Mac.

- **Formation JavaScript & HTML5 développeur RIA : niveau 1.**

Maîtrisez le HTML5 et développez des applications riches en HTML5, JavaScript, CSS3. Créez des applications Web pour terminaux mobiles iPhone, Android, etc.

- **Formation JavaScript & HTML5 développeur RIA : niveau 2.**

Devenez expert HTML5 et développez des applications riches en HTML5, JavaScript, CSS3 avec la formation perfectionnement HTML5. Créez des applications Web pour terminaux mobiles iPhone, Android, etc.

## XII-B - Remerciements

Je tiens à remercier les personnes suivantes pour leur travail de relecture, correction, suggestions et leur soutien.

- **Céline Moulard**, développeuse PHP/MySQL JavaScript chez **The Tiny Spark** pour son soutien, la relecture et la correction des fautes.
- **Pierre Girard**, gérant de la société **Mediabox**, centre de formation parisien, pour m'avoir fait confiance concernant l'intérêt de cette formation et de la rédaction d'un support de cours associé sous ce format.
- **Gaëtan Boishue**, Directeur technique de la société **Mediabox**, centre de formation parisien, pour son soutien et travail de relecture.
- **Didier Mouronval**, Responsable de publication des rubriques Développement Web pour **développez.com** pour sa relecture et son travail d'adaptation au format **développez.com**.
- **Xavier Lecomte**, responsable des rubriques JavaScript et AJAX de **développez.com** pour sa relecture et sa participation concernant la publication de cet ouvrage sur la plateforme **développez.com**.