

Alkemi Flash Bitmap Renderer, partie 1 : structure et utilisation basique

Il existe de nombreuses ressources sur le net pour le développement ActionScript, notamment dans le domaine du jeu. Comme tout le monde nous en avons pas mal profité et nous apportons aujourd'hui notre pierre à l'édifice en commençant une petite série d'articles qui présentent l'utilisation et l'implémentation du mini framework d'affichage que nous utilisons dans nos jeux Flash. Bien loin d'un outil complet comme Flixel qui assiste le développeur dans l'intégralité du fonctionnement d'un projet de jeu, notre petit package n'a que la prétention d'afficher de très nombreux sprites animés et de les afficher vite.

Avant tout choses, quelques précisions sur ce projet :

- Nous diffusons le code source ainsi que les articles qui l'accompagnent à des fins de partage et d'échange d'idées et vous êtes libres de l'utiliser ou de le modifier comme bon vous semble.
- Nous ne garantissons aucunement le fonctionnement du package et n'assureront de suivi, de correction de bugs ou d'ajout de fonctionnalités que si le cœur nous en dit !
- Nous vous encourageons par contre à nous faire part de vos commentaires, suggestions et modifications.

Etape 1 - Préparation des assets graphiques

Qui dit affichage rapide en Flash dit quasiment automatiquement bitmap. Le vectoriel, c'est sympa, redimensionnable, pas lourd à télécharger mais beaucoup trop coûteux à afficher dès qu'il y a un minimum de complexité. Bien entendu, notre système est également basé sur l'affichage de graphisme bitmap exclusivement. Quand on souhaite animer un sprite avec un outil 2D, il est souvent requis une étape de création de sprite sheet : une image regroupant tous les états du sprite lors de l'animation, bien rangés, bien alignés, souvent de même taille.



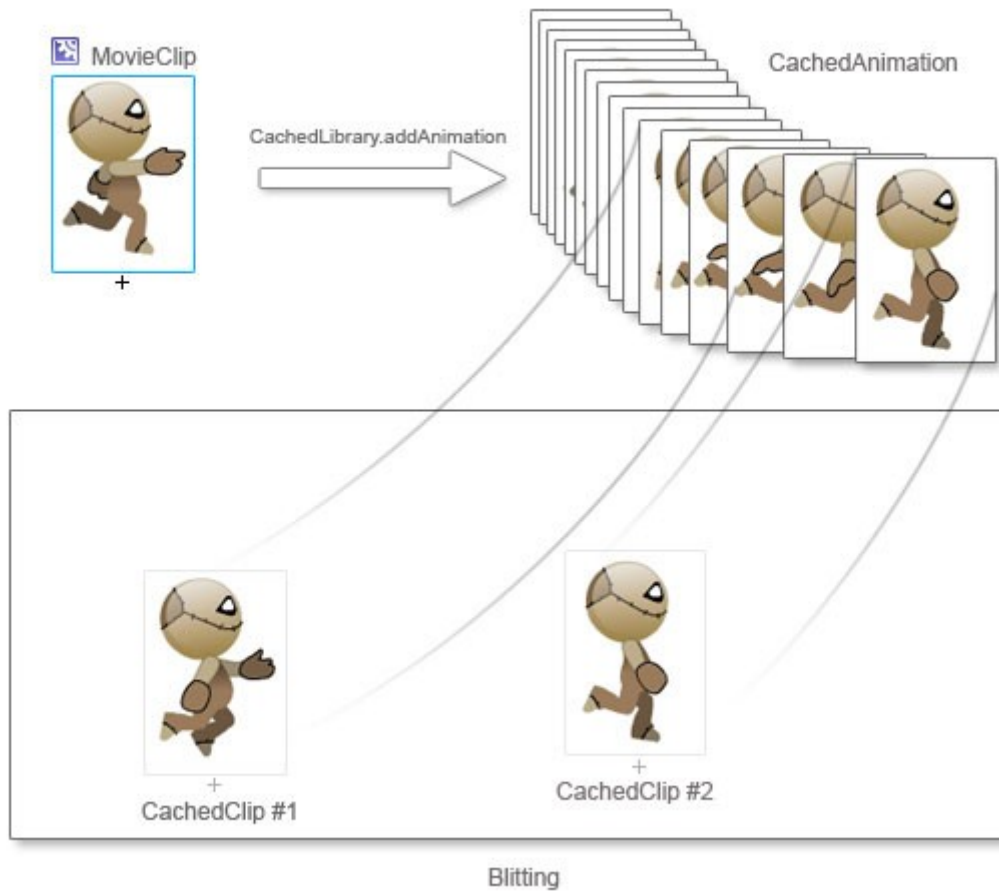
sprite sheet

Les sprite sheets c'est beaucoup moins drôle que le génialissime MovieClip de Flash pour travailler ou visualiser une animation. Par contre on fait difficilement plus lent que le MovieClip à afficher car il a été conçu pour permettre beaucoup trop de choses.

La première étape de l'utilisation de notre package est la conversion de MovieClip classiques en une ressource bitmap utilisable par le moteur de rendu. Le concept global est le suivant : on transforme un MovieClip en une CachedAnimation qui est stockée dans une CachedLibrary puis utilisée par une multitude de CachedClips.

- La CachedAnimation est essentiellement un tableau de BitmapData, d'images donc, associées aux informations de positionnement de pivot pour chaque frame.
- La CachedLibrary est juste un conteneur static de CachedAnimations qui mutualise les ressources graphiques pour l'ensemble du projet.

- Le `CachedClip` est un objet qui reproduit le comportement du `MovieClip` classique de Flash avec des méthodes `Play()`, `Stop()`, `GotoAndPlay()`, etc... Il est associé à une `CachedAnimation` de la `CachedLibrary` pour définir son aspect visuel. Le `CachedClip` est l'unique objet graphique reconnu par le renderer.



Quels sont les avantages de ce système :

- N'importe quel `MovieClip` peut être converti en `CachedClip` même si le `MovieClip` original est purement vectoriel. Cela permet de manipuler des graphismes avec les performances du bitmap mais avec les temps de chargement du vectoriel ainsi que d'utiliser des animations créées par interpolations de mouvements ou de formes.
- Il est possible de redimensionner, tourner, appliquer un filtre au `MovieClip` avant sa conversion pour pouvoir décliner celui-ci en une multitude de `CachedAnimations` différentes à moindre frais. L'affichage sera toujours aussi rapide, seule la taille en mémoire occupée par la `CachedLibrary` augmente.
- La `CachedAnimation` génère la plus petite `BitmapData` possible pour chaque frame, ce qui permet d'avoir un sprite de taille importante sur certaines étapes d'animations sans occuper inutilement de l'espace mémoire.
- La `CachedAnimation` stocke automatiquement pour chacun des frames la position du pivot du `MovieClip` par rapport au graphisme à afficher. Le `CachedClip` résultant se positionne exactement comme le `MovieClip` original.

Les limites du Système :

- Un MovieClip trop long et / ou trop grand va générer une CachedAnimation de taille considérable en mémoire. Un MovieClip de quelques dizaines de pixels de cotés peut aisément durer 50 frames ou plus. Avec un MovieClip dont les dimensions se comptent en centaines de pixels de cotés, il convient de rester très raisonnable sur le nombre de frames d'animation. Cela continuera de fonctionner, mais les configurations les plus limitées en mémoire risquent de souffrir.
- Même si plusieurs frames de l'animation sont identiques dans le MovieClip, rien n'est prévu pour le détecter. Chaque frame va générer un BitmapData supplémentaire quoi qu'il arrive.
- Actuellement le système refuse de convertir un MovieClip dont une frame serait vide en CachedAnimation. Pour contourner le problème il suffit d'ajouter un petit carré près du pivot dont l'alpha est à 0.
- Si vous souhaitez avoir un personnage qui court vers la gauche ou vers la droite, il vous faudra 2 CachedAnimations. Les performances optimales sont à ce prix. Il suffit par contre souvent d'un seul MovieClip ! La 2ème CachedAnimation est obtenue en ayant préalablement passé le scaleX du MovieClip de référence à -1.
- La mise en cache d'un MovieClip en CachedAnimation est une opération assez couteuse mais qui n'arrive souvent qu'une seule fois avant que le jeu ne commence à proprement parlé. Méfiez vous toutefois d'une accumulation trop importante de mises en cache successives pendant la même frame d'affichage du player. Notre système ne prévoit pas d'interrompre l'opération et de la reporter à la frame suivante si la limite de temps d'exécution d'un script est atteinte.

Voilà pour le principe, voyons maintenant l'utilisation :

```
// Conversion d'un MovieClip en CachedAnimation
var myClipInstance : MyClip = new MyClip () ;
CachedLibrary.addAnimation ( myClipInstance, "MyAnimation" ) ;
// Création d'une CachedAnimation symétrique horizontalement à la première
myClipInstance.scaleX = -1 ;
CachedLibrary.addAnimation ( myClipInstance, "MyAnimationTowardLeft" )
// Création d'une CachedAnimation avec un filtre de flou appliqué au visuel
myClipInstance.filters = [ new BlurFilter ( 4, 4, 1 ) ] ;
CachedLibrary.addAnimation ( myClipInstance, "MyAnimationBlurred" ) ;
// Accès à une animation de la CachedLibrary
var myAnimation : CachedAnimation = CachedAnimation
( CachedLibrary.animations["MyAnimation"] ) ;
// Création d'un CachedClip
var myCachedClip : CachedClip = new CachedClip ( myAnimation ) ;
// A tout moment il est possible de modifier l'animation d'un CachedClip
myCachedClip.animation = CachedLibrary.animations["MyAnimationBlurred"] ;
```

La différence majeure entre un MovieClip et un CachedClip est que l'apparence visuelle de ce dernier n'est qu'une propriété. Elle est modifiable à tout moment car ne fait pas partie de la définition intrinsèque de l'objet. Notez qu'à chaque modification de l'animation d'un CachedClip, celle-ci repart à la frame 1. Il est toutefois possible de récupérer le numéro de frame courante avec la propriété currentFrame et d'enchaîner la nouvelle animation à cette frame avec un simple gotoAndPlay().

Etape 2 - Le Renderer

Le moteur de rendu ou renderer fonctionne sur le principe du Blitting qui consiste à imprimer successivement tous les sprites présents à l'écran sur la même image. Flash effectue également cette opération quand le player rend la scène à chaque frame, on peut donc se demander l'inérêt d'avoir recours à de telles méthodes. Il est clair que l'utilisation des MovieClips coute cher, mais même en remplaçant tout ceux-ci par des objets Bitmaps, le Blitting 'manuel' continue de donner des performances au pire similaires et dans bien des cas largement supérieures.

Voici la structure globale du système de rendu :

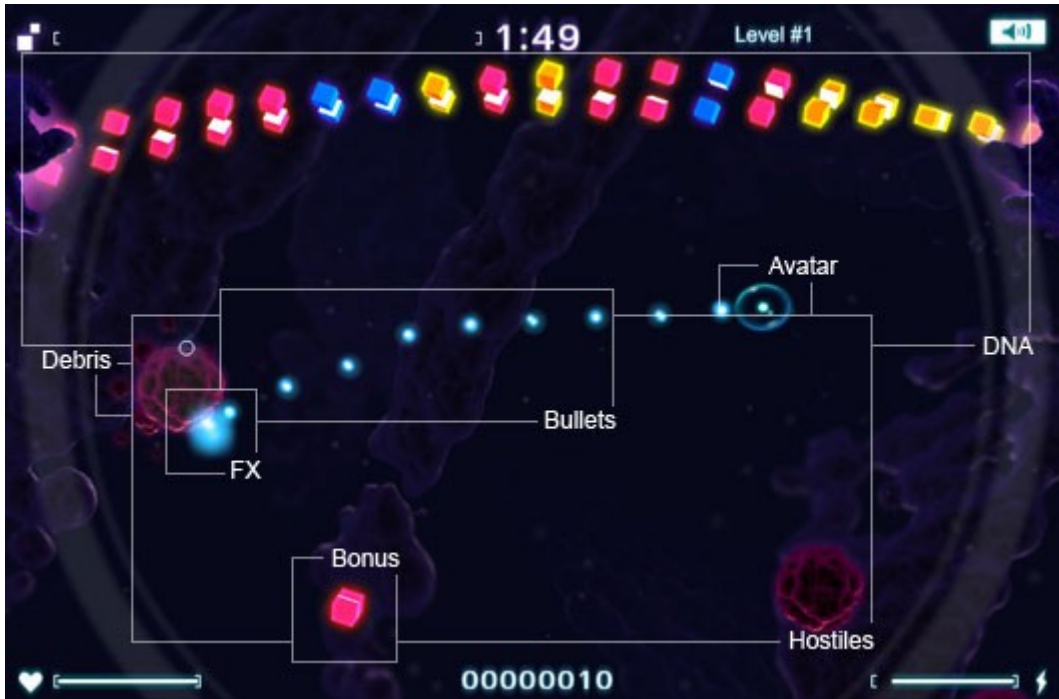
- Le BlittingRenderer est un conteneur pour les différentes couches à rendre à chaque frame. Ce n'est pas un objet graphique en tant que tel, mais juste un moyen de centraliser les commandes de rendus et de gestion des couches.
- La BlittingLayer est le coeur réel du système, c'est là que réside l'implémentation du Blitting. La BlittingLayer qui dérive avant tout de la classe Bitmap de Flash est un objet graphique qui devra être ajouté à la scène et pourra être manipulé comme n'importe quel autre Bitmap.
- Chaque BlittingLayer possède un tableau de listes de rendus, des listes de CachedClip. Ces listes peuvent être de 3 types :
 - BListBase
 - BListPool qui dérive BListBase
 - BOrderedListPool qui dérive également BListBase

Nous ne parlerons pour l'instant que des BListBases. Les 2 autres types, essentiels pour obtenir les meilleures performances, feront l'objet d'un article ultérieur spécifique.

La séquence de rendu est la suivante :

- La méthode render() du BlittingRenderer invoque render() dans chacune des BlittingLayer. L'ordre de rendu des couches n'a strictement aucune importance puisque chaque couche est un objet Bitmap distinct et c'est le dernier ajouté à la scène qui apparaîtra au dessus.
- Pour chaque BlittingLayer les listes de rendus, les BListBase, sont traitées dans l'ordre dans lequel elles ont été ajoutées : les dernières sont écrites au dessus des précédentes. De même, à l'intérieur d'une liste, chaque CachedClip est affiché par dessus tous les précédents.
- Les BlittingLayer et les listes de rendus sont des outils de tri pour votre affichage. En séparant les éléments graphiques d'un jeu en différentes listes de rendus et en les ajoutant aux couches de rendus dans un ordre convenable, il est possible d'organiser l'affichage de son jeu sans jamais avoir à se préoccuper de faire un couteux tri actif en cours d'exécution.

A titre d'exemple, voici l'architecture adoptée pour notre jeu Flash Transcribed :



L'arrière plan étant complètement statique, il est exclu du système de Blitting. L'image est incluse dans un objet Sprite de Flash ajouté en premier à la scène.

Une première BlittingLayer "couche1" est créée et ajoutée à la scène puis lui sont ajoutés dans cet ordre les listes suivantes :

- "DNA" qui contient la chaîne de cubes à détruire.
- "avatar" qui contient uniquement le visuel de l'entité contrôlée par le joueur ainsi que le visuel du cube qu'il transporte éventuellement
- "debris" qui contient les particules émises sous les monstres quand ceux-ci sont touchés
- "hostiles" qui contient l'ensemble des clips des agents biologiques hostiles
- "bonus" qui contient les power ups à ramasser

Une deuxième BlittingLayer "couche2" est ensuite créée et également ajoutée à la scène. Les listes suivantes lui sont ensuite ajoutées :

- "bullets" qui contient l'ensemble des tirs ennemis et alliés
- "FXs" qui contient l'ensemble des effets spéciaux (étincelles, explosion de cubes, textes de power ups, etc.)

Cette architecture permet d'afficher tous ces éléments distincts dans un ordre bien précis. Il n'est pas possible de dire si tel ou tel ennemi passera au-dessus ou au-dessous de tel autre, mais peu importe. Ce qui importe c'est que tous les bonus apparaîtront au-dessus des ennemis et ne seront donc jamais masqués, que les tirs également ne pourront être masqués par quasiment rien d'autres, etc. Il n'est pas rare d'avoir recours à une liste comme "avatar" qui ne contient que très peu voire un seul élément. Il n'y a aucun problème à multiplier ce genre de cas.

Pourquoi scinder les éléments graphiques en 2 BlittingLayer s? Il aurait été possible de continuer à empiler les listes de rendus dans le même ordre sur une seule et même couche. Comme nous l'avons vu, la BlittingLayer est avant tout dérivée de l'objet Bitmap de Flash. Cette séparation nous permet donc d'appliquer un aspect lumineux aux éléments de la couche2 en utilisant un blendMode additif pour celle-ci. L'application de cet effet additif individuellement à chacun des tirs et FXs est possible mais beaucoup plus coûteux que d'appliquer le rendu additif à un Bitmap plus large qui contient déjà tous les éléments. Certes l'effet n'est pas tout à fait identique, car une superposition importante de tirs ne tendra pas vers le blanc mais pour afficher un très grand nombre de projectiles le coût à payer était trop grand pour beaucoup de machines. On peut imaginer tout un tas d'effets basés sur l'utilisation des filtres appliqués aux BlittingLayers. Bien entendu le filtre concerne un bitmap assez grand donc il convient de ne pas en abuser, mais cela reste toujours moins coûteux que d'appliquer ce genre de filtre à un très grand nombre d'éléments.

Maintenant, un peu de pratique !

```
// Création du renderer
myRenderer = new BlittingRenderer () ;
// Création d'une BlittingLayer de 600 x 400 px
// Ajout de la BlittingLayer à la display list de la scène
var layer1 : BlittingLayer = myRenderer.addLayer ( "layer_1", 600, 400 ) ;
addChild ( layer1 ) ;
// Ajout de listes de rendus à layer1
var bList1 : BListBase = new BListBase () ;
myRenderer.addListToLayer ( "layer_1" , bList1 ) ;
var bList2 : BListBase = new BListBase () ;
myRenderer.addListToLayer ( "layer_1" , bList2 ) ;

// ==> bList2 sera rendu au dessus de bList1
// Ajout d'un CachedClip à une liste de rendu à la suite
// et donc au dessus des éléments déjà présents
bList1.append ( myCachedClip1 ) ;
bList1.append ( myCachedClip2 ) ;
// Ajout d'un CachedClip à une liste de rendu
// au début et donc au dessous des éléments déjà présents
bList1.prepend ( myCachedClip3 ) ;

// Retrait d'un CachedClip d'une liste de rendu
bList1.remove ( myCachedClip2 ) ;
// Mise en place du rendu à chaque frame du player
addEventListener ( Event.ENTER_FRAME, renderUpdate ) ;
function renderUpdate ( evt : Event )
{
    myRenderer.render() ;
}
```

A ce stade vous devriez déjà pouvoir afficher des clips animés.

Etape 3 - Choix de la méthode de rendu

Avant de conclure reste à aborder le dernier point très important de cette première partie qui est capital pour l'obtention de bonnes performances : la méthode de rendu. En matière de manipulation de bitmaps, Flash offre aux développeurs 2 méthodes distinctes qui peuvent être utilisées pour effectuer du Blitting. Ces 2 méthodes de la classe BitmapData sont copyPixels() et draw().

- copyPixels() est une routine extrêmement rapide de copie de blocs d'image rectangulaires d'un bitmap à un autre. Sa très grande rapidité est contrebalancée par l'impossibilité de faire autre chose que de copier un rectangle d'une image à une autre : aucune rotation possible, aucun scale possible, pas le moindre effet de manipulation de couleur ou de blending pour la zone copiée. Un autre point très important est l'obligation de copier les rectangles à des valeurs entières de pixels sur l'image de destination. Ceci à pour conséquence de rendre saccadés les mouvements lents (dont la vitesse par frame est de l'ordre du pixel ou inférieure).
- draw() par contre permet de faire tout ce qu'on a l'habitude d'attendre de Flash en matière d'affichage : rotation, scale, manipulation des couleurs via un colorTransform, modification du mode de blending, lissage des bitmaps redimensionnés et tournés et enfin affichage à des coordonnées précises à la fraction de pixel. Bien entend tout cela à un coût et la méthode draw() est significativement plus couteuse que copyPixels()... par un facteur de 2 à beaucoup plus suivant les modifications appliquées à l'image.

Bien évidemment notre moteur donne accès à ces 2 méthodes et laisse l'utilisateur libre de choisir la plus adaptée pour tout ou partie de son affichage. C'est à la création de chaque BlittingLayer que l'utilisateur peut choisir quelle sera la méthode d'affichage pour celle-ci :

```
// création d'une blittingLayer utilisant la méthode copyPixels()
var layer1 : BlittingLayer = myRenderer.addLayer ( "layer_1", 600 , 400, true,
false ) ;
// création d'une blittingLayer utilisant la méthode draw()
// mais sans lissage pour les bitmaps redimenssionnés ou tournés
var layer1 : BlittingLayer = myRenderer.addLayer ( "layer_1", 600 , 400, false, false
) ;
// création d'une blittingLayer utilisant la méthode draw()
// avec lissage pour les bitmaps redimenssionnés ou tournés
var layer1 : BlittingLayer = myRenderer.addLayer ( "layer_1", 600 , 400, false,
true ) ;
```

Par défaut, la méthode copyPixels() est utilisée et le lissage est désactivé.

Conseils :

- Pour les jeux Flash, l'utilisation de la méthode draw() est à éviter au maximum, ne l'utilisez que si vous en avez vraiment besoin et pour le minimum d'objets (créez d'autres Layers en copyPixels pour tous les autres éléments).
- Il est souvent beaucoup plus optimal de créer une CachedAnimation pour différent degrés de rotation d'un visuel animé, plutôt que d'user du paramètre rotation d'un CachedClip.
- L'une des meilleures justifications de l'utilisation de la méthode draw() reste la nécessité d'avoir des clips qui se déplacent très lentement et fluidement ce qui est impossible en copyPixels().
- Si vous avez de nombreux clips à animer avec un mouvement lent et que la méthode draw() est trop couteuse pour les machines que vous ciblez, votre meilleure atout pour masquer les saccades est l'animation propre du clip ! Dans Transcribed, les cubes d'ADN sont affichés en copyPixels() mais les saccades de la progression de la chaine sont atténués par le mouvement de rotation des 2 cubes l'un par rapport à l'autre.
- Si vous prévoyez un jeu qui utilisent certains éléments en masse (tirs, particules, etc.) arrangez vous dans le design visuel pour ne pas avoir besoin de la méthode draw pour ceux-ci.

Attention !

Si vous utilisez une BlittingLayer avec la méthode copyPixels(), tous les CachedClip qui lui sont assignés ignoreront au moment du rendu scaleX, scaleY, rotation et blendMode. Par contre si vous assignez un colorTransform à un CachedClip qui est assigné à une BlittingLayer en copyPixels(), il sera rendu en mode draw() ! Il continuera d'être rendu en mode draw() tant que le colorTransform ne sera pas passé à null. Les autres clips de cette layer ne sont pas affectés.

Ce choix d'ignorer la méthode de la BlittingLayer uniquement pour le colorTransform est totalement arbitraire ! Il a été mis au point dans Transcribed pour pouvoir appliquer un Flash animé blanc aux monstres percutés par un projectile. Pendant la courte période du Flash, les ennemis sont rendus en draw() et le reste du temps en copyPixels(). Nous aurions tout aussi bien pu le faire également pour basculer temporairement en mode draw() tout élément qui possède une rotation différente de 0, un scale différent de 1 ou bien un blendMode différent de 'normal' ou null. Nous n'avons pas fait ce choix pour éviter de multiples tests avant le rendu de la totalité des clips à chaque frame. S'il vous venait l'envie de modifier les conditions de bascule automatique de copyPixels() vers draw(), rien de plus facile ! Allez faire un tour dans alkemiTools.blitting.BlittingLayer au niveau de la méthode render(). La modification devrait être élémentaire.

C'est tout pour l'instant ! Bientôt, la présentation des Pools et BlistPools des concepts complètement nécessaires pour obtenir de bonnes performances avec notre moteur.

Fichiers sources et exemple [ici](#)