

Alkemi Flash Bitmap Renderer, partie 2 : les pools

Les procédés de bitmap caching et de blitting sont un facteur très important de performance. Il en est un autre qui peut faire une différence considérable : l'utilisation de pools d'objets. Les pools n'ont rien de graphique mais nous verrons dans un article ultérieur qu'ils sont utilisés dans notre renderer pour manipuler des objets graphiques en grandes quantités.

Qu'est ce qu'un 'pool' ?

Un pool est un groupe d'objets du même type instanciés massivement à un moment donné puis perpétuellement recyclés. Cette instanciation massive a lieu à un moment non critique de l'exécution du code, peu importe donc qu'elle soit lente. On dispose alors d'une liste d'objets libres ou disponibles. A chaque fois que l'on a besoin d'un objet, plutôt que de l'instancier, on va le piocher dans la liste des objets libres. Une fois que l'objet devient inutile, plutôt que de l'effacer (ou plus exactement de le rendre disponible pour le garbage collector) on le renvoie dans la liste des objets libres.

Le bénéfice du pool est double :

- L'opérateur 'new' est fort couteux, la réservation de l'espace mémoire nécessaire à la création d'un objet étant un processus lent. Avec un pool, l'instanciation est remplacée par un simple appel de fonction (ou 2 pour être tout à fait honnête, mais nous y reviendront).

- Les objets qui ne sont plus utilisés ne sont pas rendus disponibles pour le garbage collector. Mais quel intérêt me demanderez vous peut être ? S'il est là c'est pour servir ! Et bien oui, mais si possible aussi peu souvent que possible car cet animal là aussi est très très lent...

Quelques infos supplémentaires sur le garbage collector > [ici](#)

Voilà pour le principe général. Voyons un peu plus en détails la structure et l'utilisation d'un pool d'objet.

Les pools sont constitués de 2 listes : une liste d'objets libres ou disponibles à laquelle nous ferons désormais référence par le terme de 'free list' et une liste d'objet en cours d'utilisation ou liste active.

Toujours pour des questions de performances nous utilisons des listes chaînées plutôt que des Arrays ou des Vectors pour implémenter ces listes d'objets. Pour ceux qui ne seraient pas familier avec le concept de liste chaînée qui n'existe pas à l'origine en ActionScript, vous trouverez plus de détails et une présentation de notre implémentation [ici](#) !

A la création du pool, un grand nombre d'objets du type choisi sont instanciés. Cet opération est couteuse mais effectuée à un moment non critique du code avant que le jeu ne commence réellement, à l'initialisation d'un niveau par exemple. L'objectif pour le développeur est ici d'estimer de combien d'objets de ce type il aura besoin **au maximum simultanément** et de les créer immédiatement en une seule fois. Si le chiffre est surévalué, la seule 'perte' sera une occupation en mémoire inutile liée aux objets en surplus ce qui n'a vraiment d'importance que si l'objet en question est assez lourd. Si le chiffre est sous évalué, il faudra en cours d'exécution réinstancier une nouvelle série d'objets ce qui provoquera au pire un ralentissement ponctuel de l'exécution, en général rien de bien dramatique donc. Tous les objets initialement créés sont placées dans la free list en attente d'être utilisés.

Attention ! Comme tous les objets d'un pool sont créés au même moment avant leur utilisation réelle, la structure du pool n'a aucun intérêt à permettre de passer des paramètres au constructeur de l'objet. Il n'est en effet pas possible de savoir dans quel contexte précis sera utilisé tel ou tel objet. Les objets qui seront utilisés en pool devront donc avoir un constructeur qui ne prend pas de paramètre. Si les objets doivent être paramétrés avant leur utilisation, ce qui est plus que fréquent, il faudra utiliser une fonction d'initialisation qui jouera ce rôle juste après la mobilisation d'un objet depuis la free list.

Une fois le pool créé, sa manipulation se résume à l'utilisation de 2 méthodes : `create()` et `dispose()`.

1 - La méthode `create()`

Elle mobilise un objet disponible dans la free list et l'ajoute à la suite de la liste principale des objets en cours d'utilisation. La méthode retourne un pointeur non pas vers l'objet lui-même mais vers le node qui contient l'objet souhaité (cf les détails de notre implémentation des listes chaînées). Très souvent l'objet contenu dans le node retourné devra être immédiatement initialisé avec des paramètres qui correspondraient aux paramètres du constructeur d'un objet 'classique'. Chaque 'création' d'objet dans le cadre d'un pool se résume donc souvent à 2 appels de fonction : l'invocation de `create()` au niveau du pool puis d'une fonction `init()` au niveau de l'objet nouvellement 'créé' ou plutôt mobilisé.

Si la free list est vide quand `create()` est invoquée, le pool procède à une opération d'augmentation du nombre d'objets total. Plutôt que d'instancier un seul objet et de se retrouver de nouveau confronté au problème très rapidement, une nouvelle série d'objets dont la taille est fixée par le développeur est créée. En général plus petite que la série initiale, la création de ce nouveau lot peut au pire provoquer un léger ac-coup dans l'exécution du programme, mais si la taille initiale était bien choisie cela devrait rester très rare et ponctuel. Une fois l'opération terminée, la méthode `create()` peut retourner l'un des éléments nouvellement créés.

2 - La méthode `dispose ()`

La méthode `dispose()` a pour rôle de renvoyer un objet désormais inutile vers la free list. Il est extrêmement important de ne pas oublier de l'utiliser sinon le pool n'a aucun intérêt. La méthode `dispose()` prend en paramètre non pas l'objet à éliminer mais le node qui le contient (cf les détails de notre implémentation des listes chaînées).

Attention ! Comme les objets ne sont pas réellement effacés mais juste renvoyés dans la free list, s'ils possèdent des pointeurs vers d'autres objets qui devraient normalement être emportés par le garbage collector ils ne le seront pas puisque les références persistent. Pensez à rendre ces références null si nécessaire.

N'oubliez pas qu'un objet retourné par `create()` n'en est probablement pas à sa première utilisation ! Les différentes propriétés de l'objet devront être consciencieusement réinitialisées au moment de la création ou du renvoi vers la free list sous peine de se retrouver face à des comportements incohérents extrêmement difficiles à debugger !!!

Et maintenant un peu de pratique

```
// Création d'un pool d'objet du type MyType
// Le 2ème paramètre correspond au nombre d'objets initialement créés
// Le 3ème paramètre correspond au nombre d'objets créés quand create() est invoquée
// alors que la free list est vide
var myPool : Pool = new Pool ( MyType, 2000, 200 ) ;

// Création d'un nouvel objet de type MyType et stockage du node retourné
var newNode : LinkedListNode = myPool.create() ;

// Récupération du nouvel objet depuis la propriété data du node
var newObj : MyType = MyType ( newNode.data ) ;

// initialisation du nouvel objet ;
newObj.init ( params ) ;

// Parsing de tous les éléments actifs d'un pool en partant du début
var aNode : LinkedListNode = myPool.head,
    nextNode : LinkedListNode,
    myObj: MyType ;

while ( aNode )
{
    // pensez à stocker une référence vers le prochain node avant toute chose
    // si votre parsing implique un possible dispose sur le node, il sera trop tard
    nextNode = aNode.next ;

    // Malheureusement le Pool et ses nodes ne connaissent
    //pas le type de l'objet stocké comme pour un Array,
    // vous devrez effectuer un cast de l'objet vers le type correct
    //à chaque fois que vous le récupérez du pool !
    myObj = MyType ( aNode.data ) ;

    // update de l'objet
    myObj.update () ;

    // une propriété interne de l'objet définie s'il doit être éliminé ou non,
    //si oui on dispose le node qui le contient et non l'objet lui même
    if ( myObj.isOver )
        myPool.dispose ( aNode ) ;

    aNode = nextNode ;
}
```

Que faut il retenir de l'utilisation des pools :

- un objet nouvellement mobilisé de la free list doit systématiquement être initialisé ou plutôt réinitialiser immédiatement après l'appel à create() pour s'assurer que toutes traces de ses anciennes utilisations aient bien été effacées.

- le pool et les nodes du pools ne connaissent pas le type des objets stockés ce qui oblige à effectuer un cast avant chaque utilisation d'un objet récupéré sous peine de perdre énormément en performance (exactement comme pour un Array)

- les objets stockés dans les nodes du pool n'ont pas de référence vers le node qui les stocke. Ils n'ont donc aucun moyen d'appeler un dispose() sur eux même car c'est le node qui doit être passé en paramètre. Pour éliminer les objets désormais inutiles, lors d'un parsing complet du pool (souvent nécessaire de toutes façons pour mettre à jour les objets) utilisez un Booléen propre à l'objet pour savoir s'il doit être écarté. Une alternative est de créer vous-même une référence vers le node dans vos objets en faisant quelque chose comme ça :

```
var newNode : LinkedListNode = myPool.create() ;
var newObj : MyType = MyType ( newNode.data ) ;
newObj.node = newNode ;
```

Il existe de multiples façons d'aborder l'implémentation du concept de pool ou de liste chaînée en ActionScript. La notre a ses avantages et ses inconvénients. Certaines lourdeurs dans l'utilisation sont compensés par des performances supérieures ou bien une plus grande flexibilité dans les choix de design.

Les nodes peuvent être perçus comme superflus et il est tout à fait possible d'imaginer des listes chaînées où les objets eux mêmes jouent le rôle de nodes avec des propriétés prev et next directement inclus dans la classe. Comme nous le verrons plus tard, c'est un choix que nous avons fait pour le concept de BListBase et de BListPool dont les éléments constitutifs, les CachedClips, intègrent les fonctionnalités de node. Pour le faire proprement tous les objets destinés à être utilisés en pool devrait dériver d'une classe de base (avec les propriétés prev et next) ce qui est contraignant ou bien implémenter une interface (avec des getter / setter prev() et next()) ce qui est contre performant au possible, du moins en AS...

Toujours concernant les nodes des listes chaînées, l'un des soucis qu'ils posent est la nécessité d'utiliser théoriquement un opérateur new à chaque fois qu'on veut ajouter un élément à une liste pour créer le node conteneur. Pour contourner ce problème nous avons intégré un concept de pool de nodes pour l'ensemble de nos listes chaînées. Ce sont les méthodes de la classe LinkedList qui s'occupent de recycler les mêmes objets LinkedListNode sans avoir besoin de perpétuellement eninstancier de nouveaux.

Un autre point noir est la nécessité d'effectuer un cast de l'objet créé dans le type correct systématiquement après avoir été récupéré dans le pool. La seule option pour ne pas avoir à effectuer cette opération serait de redévelopper une classe Pool et une classe LinkedListNode spécifique pour chaque type mis en pool. La méthode create() pourrait alors directement renvoyer l'objet dans le type correct. C'est pénible mais faisable. Si vous aimez les belles choses, ce genre de cas est exactement ce pour quoi on a inventé les types génériques ! Malheureusement il n'y a pas de 'génériques' en ActionScript mais son très très proche cousin Haxe en possède lui.

Le Garbage Collector

L'Actionscript est un langage 'managé' ce qui implique entre autre qu'il s'exécute en association avec ce qu'on appelle un garbage collector (un ramasse miette en français apparemment mais c'est vraiment trop laid !).

Le garbage collector (gc) est chargé de libérer automatiquement l'espace mémoire occupé par les objets qui ne sont plus utilisés par le programme. Sauf cas particuliers, il n'y a pas d'opérateur delete en Actionscript pour supprimer un objet. Un objet est effacé de la mémoire quand il n'est plus référencé nulle part ailleurs dans le code. La non suppression des références aux objets inutilisés conduit à l'un des défauts les plus fréquents dans les applications Flash : les fuites mémoires. Mais les objets non référencés ne sont en fait pas effacés immédiatement, ils sont juste marqués comme prêts à être détruits par le gc ce qui est notablement différent. Pour le développeur il n'est pas possible, ni même normalement utile de savoir quand sera invoqué le gc la prochaine fois. Tout dépend de la quantité d'objets à éliminer, de leur taille, et surement d'une foule de paramètres tous plus pertinents les uns que les autres choisis par les ingénieurs de chez Adobe.

Pour le développeur d'un jeu Flash, il n'est pas utile de savoir quand sera invoqué le gc mais il est utile de savoir qu'il le sera le moins souvent possible ! En effet le 'passage' du garbage collector est encore une fois une opération extrêmement couteuse et qui provoque souvent une petite saccade qui se ressent aisément dans l'exécution d'un jeu. Quand votre jeu génère des dizaines d'objets graphiques à chaque cycle d'exécution et qu'il en élimine autant comme par exemples des particules, le gc est sollicité intensément et **fréquemment**. Quand les petites saccades dont je parlais précédemment interviennent à quelques secondes d'intervalle voire même moins, jouer peut devenir fort désagréable...

Avec un pool, les objets en fin de vie ne sont pas emportés par le gc, ils sont renvoyés dans la liste des objets libres et sont donc constamment référencés par votre code. C'est lors de la destruction du pool, à un moment non critique choisi par le développeur, que seront libérés tous les objets qui pourront alors être emportés par le gc.

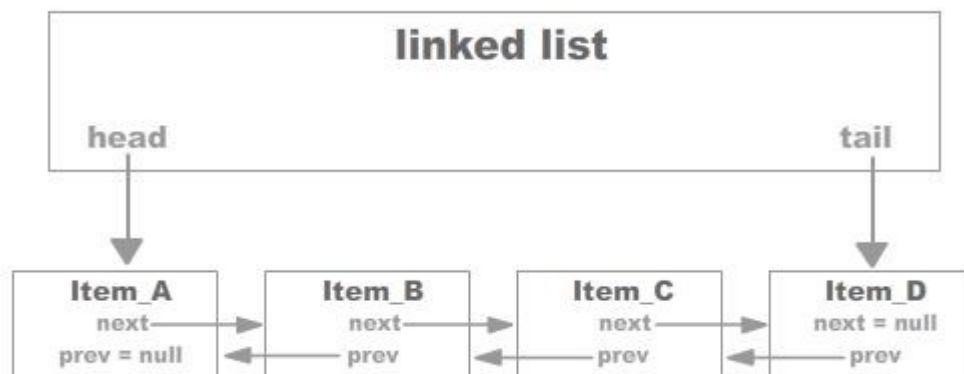
[< retour](#)

Les listes chaînées

Les listes chaînées (linked lists en anglais) sont un type de structure de données au même titre que les Arrays. Elles n'existent pas nativement en AS3 mais leur implémentation est tellement simple qu'il n'est pas bien compliqué de les créer en partant de rien. Pour comprendre ce que sont les listes chaînées le plus simple est de les comparer à une structure familière comme l'Array classique de Flash.

- Un Array permet de stocker une liste d'objets dans un tableau indexé. C'est à dire qu'il est possible d'accéder à n'importe quel objet contenu dans l'Array si l'on connaît son index. Il est bien entendu également possible de 'parcourir' entièrement la liste des objets contenu dans l'Array en partant de l'index 0 et en incrémentant l'index jusqu'à atteindre le dernier élément de la liste.

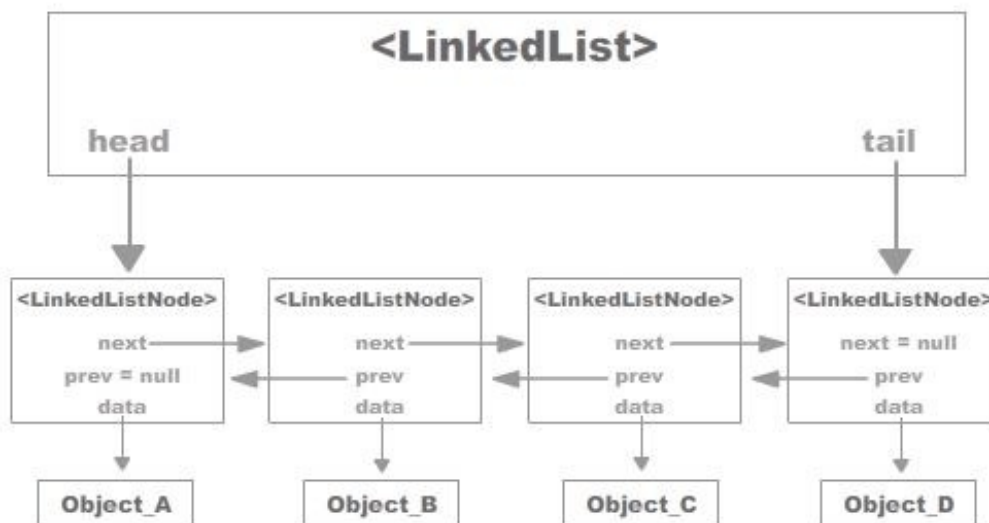
- Une liste chaînée permet également de stocker une série d'objets mais il n'y a pas d'index. La liste ne connaît que le premier élément de la série : la tête (head), et le dernier : la queue (tail). Chaque élément ou noeud (node) possède une référence vers l'élément précédent (sa propriété prev) et vers l'élément suivant (sa propriété next). Ce sont ces références croisées qui assurent la cohésion de la chaîne et qui permettent d'accéder aux différents éléments. Pour une liste chaînée, contrairement à un Array, il n'est pas possible d'accéder directement à l'élément n. Il faut partir de la tête ou de la queue et progresser de noeud en noeud via les propriétés next ou prev jusqu'à atteindre l'élément désiré. Ceci peut paraître fort contraignant mais il ne faut pas oublier que très souvent, quand on stocke des éléments dans un Array, on l'utilise uniquement comme une structure que l'on parse du début à la fin pour mettre à jour chacun des objets qu'il contient. C'est très très souvent le cas dans un jeu où l'on aura à mettre à jour une liste d'ennemis, une liste de projectiles, une liste d'obstacles, etc. Dans ce cas d'utilisation, la liste chaînée est aussi simple à utiliser qu'un Array à quelques détails de syntaxe prêt et surtout aussi performante.



Note : pour être tout à fait rigoureux, la description qui vient d'être faite est celle d'une liste chaînée double (doubly linked list) car elle peut être parcourue dans les 2 sens. En partant de la tête et en allant de noeud en noeud avec leur propriété next, ou bien en partant de la queue et en allant de noeud en noeud avec leur propriété prev. Une liste chaînée simple ne comporte qu'une tête et pas de queue et ses éléments ne possèdent qu'une propriété next. On ne peut donc la parcourir que dans un sens : du début à la fin. Notre implémentation (AlkemiTools.dataStructures.LinkedList) est celle d'une liste chaînée double bien que cela ne soit pas évoqué dans son nom.

Si les performances d'accès sont semblables, quel est l'intérêt d'aller s'ennuyer avec des listes chaînées ?! Et bien il y a un domaine dans lequel la liste chaînée pulvérise l'Array en terme de performances surtout quand on manipule de très longue liste d'objets : l'ajout et la suppression d'éléments. Quand on ajoute un élément à un Array et surtout quand on l'ajoute au début ou proche du début, il faut décaler tous les éléments suivants pour les réaffecter à un index n+1. A l'inverse quand on supprime à un élément d'un Array surtout quand il est proche du début, il faut décaler tous les éléments suivants pour les réaffecter à un index n-1. Cette opération est d'autant plus coûteuse que l'Array est grand. Il faut également ajouter que comme beaucoup de classes natives de Flash, les méthodes de la classe Array telles que push(), pop(), shift(), unshift() ou splice() sont horriblement lentes ! Quand on ajoute ou que l'on supprime un élément à une liste chaînée, il n'y a qu'un nombre très limité d'opérations à effectuer : il suffit de réaffecter correctement les propriétés prev et next des éléments ajoutés / retirés et de leurs voisins ainsi que de mettre à jour le pointeur vers la tête et la queue de la liste si nécessaire. Aucune réindexation n'est nécessaire.

Pour que des objets soient ordonnés dans une liste chaînée, il leur faut donc des propriétés prev et next. Pour éviter d'avoir à ajouter ces propriétés à tous les objets listés, on utilise un conteneur, un noeud (l'entité LinkedListNode) qui les possèdera et qui stockera l'objet dans une propriété data. On peut donc représenter la structure de nos listes chaînées comme suit :



Remarque : à chaque fois qu'on ajoute un élément à la liste il faut donc créer le noeud qui va contenir l'objet. Effectuer un new pour instancier un objet LinkedListNode est un peu ennuyeux quand on cherche justement à pousser les performances au maximum ! Nous avons donc mis en place un système de pool interne aux listes chaînées pour recycler les objets LinkedListNode et ne pas avoir à constamment en instancier de nouveaux. A ce stade, nous n'avons toujours pas vu comment fonctionnait réellement un pool mais il suffit pour l'instant de dire que cet aspect des choses est géré automatiquement par la classe LinkedList. Une fois que vous aurez assimilé le concept de pool et son utilisation, une simple lecture des commentaires de la classe LinkedList devrait vous permettre de comprendre le système.

Un exemple d'utilisation concret :

```
// création d'une liste chaînée
// le paramètre est optionnel et permet d'instancier une série de nodes
//pour éviter de le faire à un moment plus critique.
var myList : LinkedList = new LinkedList ( 100 ) ;

// Ajout d'un élément à la fin de la liste
var newObj : MyType = new MyType () ;
myList.append ( newObj ) ;

// Ajout d'un élément au début de la liste
var newObj : MyType = new MyType () ;
myList.prepend ( newObj ) ;

// Parsing de tous les éléments d'une liste chaînée en partant du début
var aNode : LinkedListNode = myList.head,
    nextNode : LinkedListNode,
    myObj: MyType ;

while ( aNode )
{
    // pensez à stocker une référence vers le prochain node avant toute chose
    // si votre parsing implique une suppression d'un node, il sera trop tard ensuite
    nextNode = aNode.next ;

    // Malheureusement la liste et ses nodes ne connaissent pas le type
    //de l'objet stocké comme pour un Array,
    // vous devrez effectuer un cast de l'objet vers le type correct
    //à chaque fois que vous le récupérez de la liste !
    myObj = MyType ( aNode.data ) ;

    // update de l'objet
    myObj.update () ;

    // une propriété interne de l'objet définie s'il doit être éliminé ou non,
    //si oui on supprime le node qui le contient et non l'objet lui même
    if ( myObj.isOver )
        myList.removeNode( aNode ) ;

    aNode = nextNode ;
}
```

La plus grosse contrainte de cette implémentation est la nécessité pour retirer un objet de la liste de connaître son node conteneur. En effet l'objet lui-même ne contient pas forcément de référence vers celui-ci. Vous pourriez décider de le faire et d'ajouter manuellement un pointeur vers le node conteneur à chaque fois qu'un objet est ajouté à la liste, en effet `append()` et `prepend()` retourne le node nouvellement utilisé pour stocker l'objet ajouté

```
myObj.node = myList.append ( myObj ) ;
```

En conclusion j'ajouterai que notre implémentation de liste chaînée ou plutôt devrais-je dire de liste chaînée double est assez sommaire :

- une méthode `append()` qui insert un nouvel élément à la fin de la liste et qui retourne le node utilisé pour le stocker
- une méthode `prepend()` qui insert un nouvel élément au début de la liste et qui retourne le node utilisé pour le stocker
- une méthode `removeHead()` qui retire le premier élément de la liste et qui retourne l'objet contenu dans le node extrait
- une méthode `removeTail()` qui retire le dernier élément de la liste et qui retourne l'objet contenu dans le node extrait
- une méthode `removeNode (node : LinkedListNode)` qui retire un élément de la liste et qui retourne l'objet contenu dans le node extrait
- une méthode `clear()` qui vide la liste

Une implémentation plus complète devrait probablement inclure :

- une méthode `insertAfter (nodeToInsert : LinkedListNode, targetNode : LinkedListNode)` pour insérer un nouvel élément après un autre
- une méthode `insertBefore (nodeToInsert : LinkedListNode, targetNode : LinkedListNode)` pour insérer un nouvel élément avant un autre
- une ou plusieurs méthodes de tri
- etc.

[< retour](#)