

Remake du CRAD sous AS3 : Tutoriel

Introduction

Rappel : Le *Cathode Ray Amusement Device* est un jeu sur oscilloscope conçu en 1947. On peut le considérer comme le premier jeu vidéo de l'histoire .

Le niveau de technologie des années 40-50 imposait qu'il s'agisse d'un jeu très simple. Il nous sera donc relativement facile de recréer ce jeu (en le remaniant un peu) sous Flash, à l'aide de programmation.

Comment procéder

On sait précisément quel système de jeu constituer sous AS3 et avec quelles mécaniques : il faut donc, au préalable, réfléchir à l'ordre dans lequel on va coder les différents éléments du jeu. C'est une sorte d'analyse du futur-programme, sur le plan algorithmique.

Si l'on veut modifier, ajouter ou supprimer des mécaniques, il faut n'avoir aucun doute sur le rôle de chaque portion de code. Il sera donc beaucoup plus facile de s'occuper de ces derniers ajustements après avoir programmé le premier prototype pleinement fonctionnel du jeu.

Compte tenu des mécaniques de notre jeu, l'ordre dans lequel on pourra procéder sera le suivant :

- 1 – Créer les éléments de base : le sprite de la cible.
- 2 – Créer la HUD
- 3 – Gestion du temps
 - => 3b – Système de temps additionnel
- 4 – Événements liés à la cible (clic)
 - => 4b – Systèmes de scoring et de combos
- 5 – Rafraîchissement de la HUD
- 6 – Délimitation d'une partie : Title screen, game over
- 7 – Boutons « Retry », niveaux de difficulté
 - => 7b – Déplacements automatiques de la cible (selon la difficulté)
- 8 – Ajustements éventuels : tailles en pixels, rayon de la cible, couleur du background...

- Tous les termes utilisés dans cette liste seront décrits et expliqués au fur et à mesure.
- Tout le code brut (sans détails explicatifs ni commentaires, donc copiable) sera redonné dans les parties « Récapitulatif ».

Commencer à coder

Préparer le fichier

On ouvre Flash pour créer un nouveau fichier en .fla. La touche F9 permet d'ouvrir la fenêtre « Actions » : c'est dans cette fenêtre qu'on va écrire tout notre code. À retenir : le raccourci Ctrl+F y est disponible pour retrouver un morceau de code.

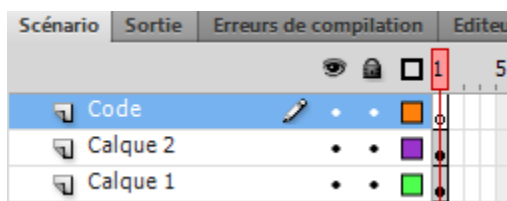
On prépare aussi le fichier en pensant à mettre 60 images par seconde, comme il s'agit d'un jeu. Les dimensions du fichier et la couleur de fond auront une importance plus tard, mais resteront modifiables.

Ce qu'il y a à savoir avec Flash IDE

On sera amené à mettre des bitmaps dans la scène (par exemple, pour le background). Comme vu en cours, mieux vaut réserver un calque vide pour y mettre le code et insérer les bitmaps dans d'autres calques. Notre calque « Code » sera au-dessus de tous les autres.

Autre point important : ici, on n'animera rien via IDE. Sur la timeline de Flash, tout sera donc contenu en une seule frame (pas la peine de créer des images supplémentaires, ni des image-clés, etc). On reverra tout ça dans la partie « Gestion du temps ».

Enfin, sous Flash, près de l'onglet Timeline (Scénario), il y a l'onglet Sortie. C'est la fenêtre dans laquelle apparaîtra tout ce qu'on voudra vérifier avec la fonction `trace()`, qui est un excellent moyen de vérifier que quelque chose fonctionne correctement lorsque l'on programme.



Se faciliter la tâche en aérant son code

Pour structurer son code et le rendre lisible, on utilise les sauts de ligne, les indentations (les grands espaces en début de ligne avec la touche tab), et les commentaires pour décrire ce que l'on fait. Un commentaire, c'est du texte qui ne sera pas lu comme compilateur, et qu'on utilise donc comme pense-bête. Il y a deux façons de faire des commentaires :

```
/*Ceci est un commentaire qui prend  
beaucoup de place donc s'étale sur  
plusieurs lignes.*/  
//Ce commentaire ne prend qu'une ligne.
```

Création du sprite de la cible

Rappel sur les Sprite

Un Sprite, c'est comparable à un calque d'Illustrator. C'est un élément initialement vide qui peut contenir des objets graphiques : on peut donc le remplir en y dessinant des tracés de Bézier, des figures géométriques, des bitmaps...

On va donc, pour commencer, créer un Sprite vide.

Création du Sprite vide

```
var cible:Sprite = new Sprite();
```

Le `var` permet de déclarer la variable (car après tout, un Sprite, c'est une variable). Le `new Sprite()` est le constructeur. On l'utilise plutôt que de rentrer une valeur quelconque ou de mettre la variable quelque part. Ça permet de la créer « intacte ». On pourrait faire ça avec n'importe quelle autre classe de variable : par exemple la ligne de code `var monNombre:Number = new Number();` créerait une variable numérique, mais pas encore assignée à une valeur précise.

Bref, on a créé un Sprite, comme si l'on avait créé un nouveau calque sous Illustrator ou Photoshop. Et on l'a nommé « cible ». Pour l'instant, il est vide. Et il n'est nulle part. On va commencer par l'intégrer à la scène avec la ligne suivante :

```
addChild(cible);
```

Dessiner dans le Sprite

Notre cible est un cercle bleu, légèrement transparent, en plein milieu de l'écran : il ne reste plus qu'à remplir le Sprite en la dessinant. L'`addChild()` aurait pu se faire à n'importe quel moment, même après avoir dessiné le cercle. Autant l'avoir fait avant, comme ça on ne risque pas de l'oublier

Lorsque l'on modifie les dessins d'un Sprite, on doit passer par sa « boîte à outils » : `.graphics`.

```
cible.graphics.beginFill(0x8888FF,0.8);  
cible.graphics.drawCircle(275,200,20);  
cible.graphics.endFill();
```

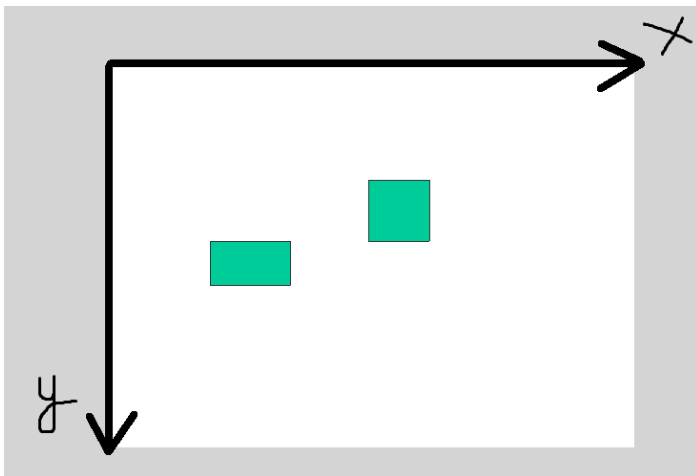
Les méthodes `beginFill()` et `endFill()` sont à considérer comme des balises, elles indiquent que l'intérieur des éléments concernés (ici, le cercle) sera coloriée dans la couleur indiquée par le `beginFill()`.

La couleur est indiquée sous la forme d'un nombre hexadécimal (`uint`), marqué

sous AS3 par un 0x au début du nombre. L'autre valeur (0.8) est facultative : c'est l'alpha de notre cercle, sa transparence. C'est un nombre entre 0 (invisible) et 1 (opaque), valant 1 par défaut. Ici, on a voulu que le cercle soit légèrement transparent.

`drawCircle()` est la méthode qui trace un cercle dans notre Sprite. On donne ses coordonnées (l'abscisse x, puis l'ordonnée y) puis le rayon du cercle en pixels (ici, 20 pixels). On a choisi de le tracer en 275,200 car le format par défaut de la fenêtre est 550*400 : le petit cercle est alors positionné en plein centre.

Attention par contre, la valeur de l'ordonnée sous Flash monte quand on va vers le bas, pas comme en mathématiques. (C'est aussi pour cette raison que les valeurs d'angles ne sont pas dans le sens trigonométrique. Mais ici, ça n'a aucune importance.)



Récapitulatif : Création du sprite de la cible

```
var cible:Sprite = new Sprite();
addChild(cible);
cible.graphics.beginFill(0x8888FF,0.8);
cible.graphics.drawCircle(275,200,20);
cible.graphics.endFill();
```

Squelette de la HUD

HUD, squelette : De quoi s'agit-il ?

Rappel : HUD signifie *Head-Up Display*, c'est l'ensemble des informations qui sont communiquées au joueur par l'interface du jeu. Par exemple, une barre de vie, un temps limite, des munitions...

Ici, on va afficher un score, le combo en cours, le meilleur combo réalisé dans la partie, et le temps restant.

En revanche, on ne s'occupera que plus tard d'afficher les valeurs numériques de façon dynamique (c'est-à-dire rafraîchies en temps réel), et l'on règlera plus tard un problème majeur : le joueur n'aura sûrement pas le temps d'aller regarder la HUD de près, dans le feu de l'action. C'est la partie qu'on a appelée tout à l'heure « Rafraîchissement de la HUD », et elle nécessite de s'être déjà occupé de la gestion du temps.

Bref, on va créer pour l'instant le squelette de la HUD : les boîtes de texte dans lesquelles tout s'affichera, et les champs de texte qui n'ont pas besoin d'être rafraîchis, c'est-à-dire les mots « Score », « Combo », « HiCombo » et « Time ».

Fonctionnement logique des boîtes de texte

Une boîte de texte, c'est un objet de classe `TextField`, doté de propriétés qui permettent de le styliser : on peut y régler sa taille, sa couleur, son contenu... Mais pas styliser son contenu (mettre en page le texte, le mettre en gras, changer sa police, etc).

Pour tous les paramètres qui touchent directement au texte mais pas à la boîte de texte, on utilise un objet de mise en page, de classe `TextFormat`, dont les propriétés vont modifier le texte.

Enfin, une fois que les deux sont créés, on les lie ensemble pour indiquer à Flash que notre `TextFormat` concerne le contenu du `TextField` voulu.

Création du `TextFormat`

On va commencer par le format de texte, car il sera le même pour les quatre boîtes. Il suffit donc de créer un seul `TextFormat`, qu'on liera aux quatre `TextField` quand on les aura créés.

Dans ce `TextFormat`, on va donc préciser qu'on veut un texte en Calibri, de taille 16, gras, et centré. (Une exception : la couleur du texte se règle dans les propriétés du `TextField`.)

```
var format1:TextFormat = new TextFormat;  
format1.font = "Calibri"; // Attention aux guillemets  
format1.size = 16;  
format1.bold = true;  
format1.align = TextFormatAlign.CENTER;
```

Création d'une fonction

Il est temps de créer les TextField. Mais d'abord, réfléchissons à quelque chose : on a quatre boîtes de texte, et elles vont à peu près toutes avoir le même aspect. Donc, quatre TextField. Ces TextField vont donc avoir de nombreuses propriétés en commun : on peut s'éviter d'avoir à recopier quatre fois les mêmes lignes de code, en les mettant dans une fonction.

Attention, la couleur du texte ne se change pas dans le TextFormat, mais ici (avec la propriété `.textColor`).

```
function miseEnForme(boite) {  
    boite.border = true;  
    boite.borderColor = 0x8888FF;  
    boite.background = true;  
    boite.backgroundColor = 0x000088;  
    boite.textColor = 0x8888FF;  
    boite.width = 55;  
    boite.height = 24;  
    boite.selectable = false;  
}
```

La fonction vient d'être créée. Elle n'est pas pour autant exécutée : à chaque fois qu'on voudra exécuter ces six lignes, il faudra « appeler la fonction », c'est-à-dire faire appliquer la fonction au TextField voulu.

Le « boîte » utilisé dans la création de la fonction correspond à l'argument de la fonction : c'est l'objet sur lequel on applique la fonction. N'importe quel nom peut convenir, il suffit juste d'être cohérent jusqu'au moment où l'on referme l'accolade. Et d'éviter de prendre le nom d'une variable déjà existante (quitte à utiliser des mots qui n'ont rien à avoir, comme des noms de légume).

Note : les propriétés `.border` et `.background` sont des Boolean. Une variable booléenne est une variable dont la valeur ne peut être que « true » ou « false » : ce sont comme des interrupteurs. Ici, on les place sur true pour que nos boîtes aient un contour et un fond de couleurs réglables.

Création des TextField

Encore une dernière remarque, pourquoi une largeur de 55 et une hauteur de 38 ? Ça dépend de la forme que l'on veut donner à notre HUD, ensuite il faut comparer le nombre de boîtes au nombre de pixels dans lesquels on veut faire rentrer tout ça, or le fichier fait 400px de largeur... Bref, c'est calculé comme ça et ajustable selon nos besoins.

Les coordonnées de nos boîtes suivront le même principe pour les placer. Bon, d'abord, créons les boîtes :

```
var boiteScore:TextField = new TextField();  
var boiteCombo:TextField = new TextField();  
var boiteHiCombo:TextField = new TextField();  
var boiteTemps:TextField = new TextField();
```

Ensuite, on n'oublie pas le addChild() pour chacune des boîtes. On n'a pas pu le mettre dans la fonction parce que c'est une *méthode* (en gros, un outil qui a le même fonctionnement qu'une fonction).

```
addChild(boiteScore);  
addChild(boiteCombo);  
addChild(boiteHiCombo);  
addChild(boiteTemps);
```

Après, on applique notre fonction préalablement créée à nos boîtes. Ça va prendre quatre lignes alors que ça en aurait pris 28 si on avait tout refait pour chacune... Ici, on n'a plus qu'à *appeler* notre fonction quatre fois.

```
miseEnForme(boiteScore);  
miseEnForme(boiteCombo);  
miseEnForme(boiteHiCombo);  
miseEnForme(boiteTemps);
```

Ajustement des boîtes de texte

Il reste les propriétés de chaque boîte qui sont différentes pour chacune et qu'il était donc impossible de mettre dans la fonction. C'est-à-dire : les coordonnées x et y, et le texte contenu dans chaque boîte.

```
boiteScore.text = "Score";
boiteScore.x = 0;
boiteScore.y = 350;
boiteCombo.text = "Combo";
boiteCombo.x = 110;
boiteCombo.y = 350;
boiteHiCombo.text = "HiCombo";
boiteHiCombo.x = 220; //Valeur que l'on va vite modifier...
boiteHiCombo.y = 350;
boiteTemps.text = "Time";
boiteTemps.x = 330; //...De même pour celle-ci.
boiteTemps.y = 350;
```

Important : les coordonnées cartésiennes (en x et y) concernent le *point d'ancrage* de la boîte, qui correspond à son coin haut-gauche.

Après test, le mot « HiCombo » est trop gros pour sa boîte de texte. On va donc élargir cette boîte de 15 pixels et décaler le reste en modifiant les coordonnées entrées ci-dessus.

```
boiteHiCombo.width = 70;
```

En toute logique, cette valeur « écrase » celle indiquée par la fonction `miseEnForme`, parce que la ligne de code a été tapée **après** avoir appelé la fonction.

Donc, on modifie les valeurs rentrées ci-dessus pour ne rien bouleverser : la coordonnée x de la boîte du temps doit valoir 345 et non 330.

Enfin, on lie le format de texte aux boîtes. Cette ligne non plus ne pouvait pas être mise dans la fonction, car pour la liaison, on va se servir d'une méthode.

```
boiteScore.setTextFormat(format1);
boiteCombo.setTextFormat(format1);
boiteHiCombo.setTextFormat(format1);
boiteTemps.setTextFormat(format1);
```

Affichage des valeurs numériques

On a laissé de la place entre les boîtes. C'est parce que l'on va en créer d'autres pour afficher, en plus gros, les valeurs du score, du combo, etc. Celles-ci ne seront pas dynamiques pour le moment (cf. les explications sur le squelette de la HUD), mais il faut bien qu'elles soient créées et existantes dès le début.

Les toutes premières lignes de notre code (juste après les import) doivent être dédiées à la déclaration des variables utilisées tout au long du programme. C'est donc au début de notre programme qu'on va (enfin) créer les valeurs correspondant au score, au combo, au plus haut combo de la partie, et au temps restant (en secondes).

```
var score:int = 0;  
var combo:int = 0;  
var hiCombo:int = 0;  
var time:int = 15;
```

Bref, revenons là où on en était dans notre code (la dernière ligne). On a besoin de créer un nouveau format de texte que l'on appliquera à ces chiffres (parce qu'on veut les afficher en plus gros), et quatre nouvelles boîtes de texte. Et leur hauteur sera plus grande, donc soit on crée une nouvelle fonction soit on pense à changer la hauteur après avoir appliqué la fonction de tout à l'heure.

Pour faire court : il s'agit de refaire ce qui a été fait précédemment, avec un nouveau format de texte. Le texte contenu dans ces nouvelles boîtes correspondra aux variables numériques créées tout à l'heure, **converties en chaînes de caractères**. Pour cela, on se sert de la méthode `.toString()`. Exemple :

```
boiteCombo2.text = combo.toString(); //Donc pas de guillemets  
sur ce coup.
```

Récapitulatif : Squelette de la HUD

En début de code (après les import) :

```
var score:int = 0;
var combo:int = 0;
var hiCombo:int = 0;
var time:int = 15;
```

À la suite du reste :

```
var format1:TextFormat = new TextFormat;
format1.font = "Calibri";
format1.size = 16;
format1.bold = true;
format1.align = TextFormatAlign.CENTER;

function miseEnForme(boite){
    boite.border = true;
    boite.borderColor = 0x8888FF;
    boite.background = true;
    boite.backgroundColor = 0x000088;
    boite.textColor = 0x8888FF;
    boite.width = 55;
    boite.height = 24;
    boite.selectable = false;
}

var boiteScore:TextField = new TextField();
var boiteCombo:TextField = new TextField();
var boiteHiCombo:TextField = new TextField();
var boiteTemps:TextField = new TextField();

addChild(boiteScore);
addChild(boiteCombo);
addChild(boiteHiCombo);
addChild(boiteTemps);

miseEnForme(boiteScore);
miseEnForme(boiteCombo);
miseEnForme(boiteHiCombo);
miseEnForme(boiteTemps);
```

```
boiteScore.text = "Score";
boiteScore.x = 0;
boiteScore.y = 350;
boiteCombo.text = "Combo";
boiteCombo.x = 110;
boiteCombo.y = 350;
boiteHiCombo.text = "HiCombo";
boiteHiCombo.x = 220;
boiteHiCombo.y = 350;
boiteTemps.text = "Time";
boiteTemps.x = 345;
boiteTemps.y = 350;

boiteHiCombo.width = 70;

boiteScore.setTextFormat(format1);
boiteCombo.setTextFormat(format1);
boiteHiCombo.setTextFormat(format1);
boiteTemps.setTextFormat(format1);

var format2:TextFormat = new TextFormat;
format2.font = "Calibri";
format2.size = 24;
format2.bold = true;
format2.align = TextFormatAlign.CENTER;

var boiteScore2:TextField = new TextField();
var boiteCombo2:TextField = new TextField();
var boiteHiCombo2:TextField = new TextField();
var boiteTemps2:TextField = new TextField();

addChild(boiteScore2);
addChild(boiteCombo2);
addChild(boiteHiCombo2);
addChild(boiteTemps2);

miseEnForme(boiteScore2);
miseEnForme(boiteCombo2);
miseEnForme(boiteHiCombo2);
miseEnForme(boiteTemps2);

boiteScore2.height = 38;
boiteCombo2.height = 38;
boiteHiCombo2.height = 38;
boiteTemps2.height = 38;
```

```
boiteScore2.text = score.toString();
boiteScore2.x = 55;
boiteScore2.y = 360;
boiteCombo2.text = combo.toString();
boiteCombo2.x = 165;
boiteCombo2.y = 360;
boiteHiCombo2.text = hiCombo.toString();
boiteHiCombo2.x = 290;
boiteHiCombo2.y = 360;
boiteTemps2.text = time.toString();
boiteTemps2.x = 400;
boiteTemps2.y = 360;

boiteScore2.setTextFormat(format2);
boiteCombo2.setTextFormat(format2);
boiteHiCombo2.setTextFormat(format2);
boiteTemps2.setTextFormat(format2);
```

Gestion du temps

Rappel sur les événements

Les événements sont la base de l'interaction en ActionScript 3 : il est possible de ne faire exécuter une fonction qu'à l'instant où l'utilisateur clique sa souris, la passe au-dessus d'un objet, appuie sur une touche de son clavier... À chacune de ces actions correspond un type différent d'événement. On distingue ainsi 3 types d'événements :

- MouseEvent (souris)
- KeyboardEvent (clavier)
- Event (tous les événements qui ne dépendent ni de la souris ni du clavier)

Ces trois types se divisent ensuite en sous-catégories telles que CLICK, DOUBLE_CLICK, KEY_UP (relâcher une touche du clavier), etc. Dans la partie qui suit, on va s'intéresser uniquement à une sous-catégorie du type Event : la catégorie ENTER_FRAME.

L'événement ENTER_FRAME

ENTER_FRAME est un événement très utile : alors que la plupart signifient « lorsque l'utilisateur clique ici », « lorsque l'utilisateur appuie ici », « l'utilisateur fait ça », celui-ci signifie « à chaque frame ».

Sous cet événement, on va associer une fonction à l'événement, qui sera exécutée 60 fois par seconde. C'est en utilisant judicieusement cet outil qu'on va donc pouvoir gérer tout ce qui est **continu** dans le programme, à commencer par l'écoulement du temps limite.

```
stage.addEventListener(Event.ENTER_FRAME,ecoulement);  
function ecoulement(e:Event){  
}
```

En gros, voilà à quoi ressemble la structure de l'événement. Ça fonctionne comme ça pour tous les types d'événements, et on étudiera mieux cette structure plus tard en réalisant l'événement souris lié à la cible. La méthode addEventListener() assigne un événement d'un certain type (et sous-type) à l'objet de notre choix, et assigne une fonction à cet événement. Après quoi, on crée la fonction qui est censée s'exécuter sur activation de l'événement : son argument doit être « e:Event ».

Ici, on a lié l'événement à l'objet « stage », qui désigne la scène entière. En toute logique il n'y a que pour les MouseEvent que l'on devra utiliser un autre objet (l'objet sur lequel l'utilisateur devra cliquer).

Évolution du temps limite

Il ne reste donc plus qu'à remplir les accolades de la fonction avec le code qui va faire avancer le temps limite. On a besoin que la variable que l'on a appelée *time* soit décrémentée (= baissée de 1) à chaque seconde.

Pour cela, il existe des objets permettant de mesurer le temps sous Flash, mais ils sont compliqués d'utilisation. (D'ailleurs, si on les avait utilisés, on n'aurait pas eu besoin d'enterframes.)

On va donc bricoler un compteur de frames et faire baisser le temps limite toutes les 60 frames (comme le jeu tourne à 60 fps). Ce compteur, c'est une simple variable numérique que l'on va incrémenter (= augmenter de 1) à **chaque frame**. D'où la nécessité de l'événement ENTER_FRAME. On crée ce compteur en début de code (là où l'on a créé score, combo, hiCombo et time) :

```
var frameCounter:int = 0;
```

Et dans les accolades de notre fonction, on va dire à Flash d'incrémenter cette variable, puis de décrémenter la variable *time* si *frameCounter* est un multiple de 60 (et seulement s'il reste au moins une seconde de temps limite, parce qu'on va éviter de finir avec du temps négatif).

Subtilité : Au lieu de décrémenter le compteur à 60, 120, 180... on le décrémente à 59, 119, 179, etc. Parce que 0 est considéré comme un multiple de 60, ce qui « volerait » une seconde au joueur en début de partie.

```
frameCounter++;  
if((time>0) && (frameCounter%60==59)) {  
    time--;  
}  
if(time==0) {  
    trace("Game Over");  
}
```

Le trace() est temporaire, on le remplacera lorsque l'on aura une portion de code correspondant au game over.

Explication sur les structures logiques et mathématiques :

- > signifie « est supérieur à »
- == signifie « est égal à »
- && signifie « et » : c'est un connecteur qui fusionne deux propositions logiques.
- % est l'opérateur mathématique « modulo ». (frameCounter%60==59) signifie « le reste de la division de frameCounter par 60 vaut 59 », donc *frameCounter* peut valoir 59, 119, 179...
- ++, c'est l'incréméntation (+1). À l'inverse, --, c'est la décréméntation (-1).

Pour le moment, on va s'arrêter ici pour l'ENTER_FRAME. Mais au fur et à mesure de la création de ce jeu, on aura d'autres portions de code à y mettre, et on enrichira notre fonction. Surtout, ne pas créer de nouvelle fonction ou de nouvel ENTER_FRAME, et bien garder celui-ci : c'est un outil qui consomme beaucoup de ressources et le jeu risque d'être ralenti si on en fait plus d'un. Donc, lorsque l'on aura quelque chose qui doit se produire soit **tout le temps** soit **n'importe quand**, on retournera dans cette portion de code pour agrandir le contenu de la même fonction.

C'est notamment ce que l'on fera par la suite pour actualiser l'affichage de ces paramètres dans la HUD dès l'instant où ils sont modifiés : en effet, même si la valeur du temps baisse correctement, pour l'instant la HUD nous indique toujours « 15 secondes restantes. » À ce stade, si l'on veut vérifier notre travail, on doit passer par la fonction trace(), en attendant de rendre la HUD dynamique.

Système de temps additionnel

Le concept de jeu prévoyait une mécanique de temps additionnel en fonction du hiCombo (secondes supplémentaires = moitié de la valeur du hiCombo). Profitons de l'occasion pour l'implanter maintenant : la prolongation doit être donnée au moment où il n'y a plus de temps. Le « Game Over » n'est donc plus la seule conséquence possible de la proposition « time = 0 » : il faut modifier la structure logique dans l'ENTER_FRAME.

Par ailleurs, pour s'assurer que la prolongation n'est donnée qu'une seule fois, on va utiliser un « **interrupteur** » : une variable de classe Boolean. (La prolongation a-t-elle été déjà donnée ? => Oui/Non (True/False) => soit on la donne, soit il y a game over.)

On crée la booléenne en début de code (au même endroit que d'habitude pour la création de variables) :

```
var prolongation:Boolean = false;
```

Maintenant, dans l'ENTER_FRAME, on va un peu changer les conditions du deuxième *if* pour ajouter des éventualités. Il existe d'autres schémas logiques aboutissant au même résultat, celui-ci est un exemple :

```
if((time==0)&&(prolongation==false)){  
    time += hiCombo/2;  
    prolongation = true; //On bascule l'interrupteur. La  
    prochaine fois qu'on aura time==0, ce sera game over.  
}else if(time==0){  
    trace("Game Over");  
}
```

Une fois de plus le `trace()` est temporaire. On attend d'avoir un écran de game over à afficher, et cela impliquera d'autres phénomènes que l'on codera un peu plus tard.

Récapitulatif : Gestion du temps

En début de code (après les import) :

```
var frameCounter:int = 0;
var prolongation:Boolean = false;
```

À la suite du reste :

```
stage.addEventListener(Event.ENTER_FRAME,ecoulement);
function ecoulement(e:Event){
    frameCounter++;
    if((time>0)&&(frameCounter%60==59)){
        time--;
    }
    if((time==0)&&(prolongation==false)){
        time += hiCombo/2;
        prolongation = true;
    }else if(time==0){
        trace("Game Over");
    }
}
```

Événements liés à la cible (clic)

L'événement souris : le MouseEvent

On s'occupe maintenant de l'événement lié à la cible : lorsque l'on clique dessus, elle change aléatoirement d'emplacement. L'événement correspond à un clic, il s'agit donc d'un MouseEvent de catégorie CLICK. On le crée tout de suite :

```
cible.addEventListener(MouseEvent.CLICK,mouvement);  
function mouvement(e:MouseEvent){  
}
```

Comme précédemment, il reste donc à remplir les accolades de la fonction.

Déplacement aléatoire

On doit changer les coordonnées de la cible pour une valeur aléatoire. La seule façon qu'on connaît d'indiquer à Flash une valeur aléatoire : Math.random()*XX correspond à une valeur aléatoire entre 0 et XX.

Problème : pour le Sprite, le point de coordonnées 0,0 est son centre, c'est-à-dire le centre de la cible.

On veut donc une valeur d'abscisse aléatoire entre -180 et +180, et une ordonnée entre -255 et 130 (calcul sur le nombre de pixels en tenant compte du rayon de la cible, qui sera sujet à ajustements plus tard). Pour obtenir une telle fourchette, on va partir d'un aléatoire en Math.random() et y soustraire une valeur donnée.

```
cible.x = (Math.random()*360)-180;  
cible.y = (Math.random()*385)-255;
```

Système de scoring et de combos

Le clic sur la cible a aussi d'autres effets qu'un simple déplacement : il doit aussi augmenter la valeur du combo, du hiCombo s'il y a lieu, et monter le score en conséquence. On va donc compléter notre fonction :

```
combo++;  
if(hiCombo<combo){  
    hiCombo = combo;  
}  
score += combo; //Formule qui correspond à la mécanique de  
scoring définie dans le GOD.
```

Et en cas de missclick (échec), comment briser le combo ? Il aurait fallu mettre un autre MouseEvent sur le stage (donc pas sur la cible) **avant** le MouseEvent de la cible. De cette façon, il s'appliquera à toute zone de l'écran qui n'est pas la cible : autrement dit, cliquer n'importe où entraînera la perte du combo, excepté sur la cible.

L'ordre est important : en toute logique, l'événement qui est placé après écrase le premier. On s'assure ainsi que cliquer sur la cible n'activera pas l'événement du stage.

Si ça ne fonctionne pas, on mettra l'événement du missclick sur l'image de background.

```
stage.addEventListener(MouseEvent.CLICK,missclick);  
function missclick(e:MouseEvent){  
    combo = 0;  
}
```

Récapitulatif : événements liés à la cible

```
stage.addEventListener(MouseEvent.CLICK,missclick);  
function missclick(e:MouseEvent){  
    combo = 0;  
}  
  
cible.addEventListener(MouseEvent.CLICK,mouvement);  
function mouvement(e:MouseEvent){  
    cible.x = (Math.random()*360)-180;  
    cible.y = (Math.random()*385)-255;  
    combo++;  
    if(hiCombo<combo){  
        hiCombo = combo;  
    }  
    score += combo;  
}
```

Rafraîchissement régulier de la HUD

Principe

L'événement souris qu'on vient de poser sur la cible est visible et vérifiable : si on teste le fichier (Ctrl+Enter), on a bien une cible qui bouge aléatoirement lorsque l'on clique dessus.

En revanche, le reste est plus délicat : notre score, combo et hiCombo changent de valeur. Mais on n'a aucun moyen de le savoir (à part avec la fonction `trace()`) parce que notre HUD n'actualise pas les valeurs.

La solution : il faut réécrire le texte de la boîte de la HUD que l'on veut actualiser, dès que son contenu change. Et penser à réinstaller le format de texte sur la boîte, car il est perdu lorsque l'on change le texte. À chaque fois qu'une de notre quatre variables de la HUD est changée, il faut donc retaper les deux lignes suivantes : (exemple avec la variable `combo`)

```
boiteCombo2.text = combo.toString();  
boiteCombo2.setTextFormat(format2);
```

On va donc parcourir notre code, repérer tout endroit où l'une de ces variables est modifiée (donc surtout dans `ENTER_FRAME` et le `MouseEvent`) et y écrire ces deux lignes. Au total, on a ça pour :

- Lorsque le temps est décrémenté
- Lorsque la prolongation de temps est accordée
- Lorsque le score monte
- Lorsque le combo monte
- Lorsque le combo tombe à 0
- Lorsque le hiCombo monte

Par la suite, quand on donnera au joueur la possibilité de recommencer une partie, on devra penser à faire la même chose pour remettre les variables à leurs valeurs initiales (15 pour le temps, 0 pour le reste).

Récapitulatif : rafraîchissement régulier de la HUD

Après la ligne « score += combo; »

```
boiteScore2.text = score.toString();  
boiteScore2.setTextFormat(format2);
```

Après les lignes « combo++; » et « combo = 0; »

```
boiteCombo2.text = combo.toString();  
boiteCombo2.setTextFormat(format2);
```

Après la ligne « hiCombo = combo; »

```
boiteHiCombo2.text = hiCombo.toString();  
boiteHiCombo2.setTextFormat(format2);
```

Après les lignes « time--; » et « time += hiCombo/2 » :

```
boiteTemps2.text = time.toString();  
boiteTemps2.setTextFormat(format2);
```

Délimitation d'une partie

Organisation

Pour l'instant, ce jeu souffre de deux énormes problèmes :

- La partie démarre dès que le programme est lancé.
- Il n'y a pas réellement de game over.

On va donc mettre en place ce qui peut retenir la partie de débiter avant que le joueur ne l'ait souhaité, et qui va coincer la partie après le game over. En fait, il suffit d'une simple booléenne : soit le jeu est lancé et le temps s'écoule, soit le jeu « attend » et l'écoulement du temps limite sera figé.

Pour la réactivité de la cible aux clics, c'est encore plus simple car il suffit de « détruire » l'événement souris avec `removeEventListener()` lorsque l'on n'en a pas besoin.

Empêcher une partie de démarrer toute seule

Bref, on va commencer par fabriquer la booléenne qui va contrôler le temps, là où on a déclaré toutes les variables en début de code :

```
var jeuEnCours:Boolean = false;
```

Maintenant on va empêcher le temps de s'écouler tant que cette variable a pour valeur false. Pour rappel, l'écoulement du temps est entièrement conditionné par la ligne de code suivante :

```
frameCounter++;
```

Il suffit donc d'encadrer cette ligne d'un if portant sur la booléenne `jeuEnCours`.

```
if(jeuEnCours==true) {  
    frameCounter++;  
}
```

Et voilà, le temps ne va pas s'écouler tant que le joueur ne démarre pas la partie. On va considérer que le démarrage de la partie correspond au moment où le joueur clique pour la première fois sur la cible. C'est donc dans la fonction du `MouseEvent` de la cible que l'on va rajouter la ligne basculant notre booléenne sur true :

```
jeuEnCours = true;
```

Le Game Over

Voilà pour les débuts de parties. Il reste maintenant à s'occuper du Game Over (et, plus tard, du bouton qui permet de lancer une nouvelle partie).

On va commencer par définir clairement un Game Over, pour savoir ce qu'on va devoir coder :

- L'écoulement du temps s'arrête (et le compteur de frames retourne à 0)
- Cliquer sur la cible n'est plus possible
- La cible revient au centre de l'écran (histoire de faire propre)
- L'écran indiquant qu'on a fait un Game Over apparaît.

Donc voilà ce qu'on va devoir mettre à la place du vulgaire `trace("Game Over")` de tout à l'heure :

```
jeuEnCours = false;  
frameCounter = 0;  
cible.x = 0;  
cible.y = 0;  
cible.removeEventListener(MouseEvent.CLICK, mouvement)  
GameOver1.visible = true;
```

GameOver1, c'est l'écran de GameOver que l'on va faire apparaître. Pour l'instant, il n'est pas encore fabriqué, donc lancer le programme tel quel causera une erreur. Bref, il est temps de s'occuper des écrans de démarrage et de game over.

Titlescreen, Game Over screen, background du jeu

Il s'agit des trois éléments que l'on va créer en dehors de Flash : on va prendre un bitmap pour le background, que l'on va importer dans Flash en tant que symbole.

Et pour les écrans de démarrage et de game over, on fait ce qu'on veut avec Illustrator ou Photoshop et on l'importe dans la bibliothèque en **.png** (pour gérer la transparence).

Sur des calques différents (et différents de celui sur lequel on code), dans l'IDE Flash, on pose des occurrences de nos symboles « écran ». Évidemment le background va rester en permanence visible derrière la cible, le titlescreen et le game over screen : on le met dans le calque le plus bas.

Pour créer en tant que symbole :

```
Fichier => Importer... => Importer dans la bibliothèque => On prend notre fichier en .png.
```

Pour convertir en Clip ce symbole importé :

Ctrl+L ouvre la bibliothèque. Ensuite, clic droit sur notre titlescreen/background/etc : convertir en symbole.

Pour le mettre dans la scène :

On drag'n'drop le Clip dans la scène depuis la bibliothèque : on a maintenant une **occurrence** de ce Clip dans notre scène. Dans la colonne de droite (en affichage classique de l'IDE Flash), on peut renommer cette occurrence. Son nom est très important car c'est ce nom qu'on utilisera dans le code.

J'ai appelé mes occurrences respectivement Background1, GameOver1 et StartScreen1, bien qu'il n'y ait jamais besoin de mentionner le background dans le code. Le 1 en fin du nom de l'occurrence est juste là pour la clarté.

Les occurrences de Clip dans la scène sont considérées comme des objets de classe MovieClip. Comme le Sprite, ils ont des propriétés que l'on va pouvoir modifier, mais ne nécessitent pas que l'on fasse appel à une boîte à outils comme .graphics.

La propriété qui va nous intéresser ici, c'est *.visible*, une booléenne utilisée pour cacher et faire réapparaître les deux écrans.

Donc au tout début de notre code (là où on a créé toutes nos variables), on rajoute ces deux-là :

```
StartScreen1.visible = true;  
GameOver1.visible = false;
```

(La première ligne est inutile car c'est déjà sur true par défaut. Mais on le remet pour rester cohérent, faciliter compréhension du code, etc.)

Ainsi, au début, lorsque l'on lance le jeu, on verra l'écran de démarrage mais pas l'écran de game over (logique). Il faut donc :

- Faire disparaître StartScreen1 lorsque la partie démarre
- Faire apparaître GameOver1 lorsque la partie est finie

On s'est déjà occupé du GameOver1.visible = true plus haut, c'est même ce qui nous a amenés à fabriquer les deux écrans. Il ne manque donc que la disparition de StartScreen1 au début de la partie, c'est-à-dire lorsque le joueur clique sur la cible pour commencer. On va donc mettre ça dans la fonction du MouseEvent de la cible.

```
StartScreen1.visible = false;
```

Récapitulatif : délimitation d'une partie

Créer le background, et les deux écrans de titlescreen/game over

- Fichier => Importer... => Importer dans la bibliothèque => On prend notre fichier en .png.
- Ctrl+L ouvre la bibliothèque. Ensuite, clic droit sur notre titlescreen/background/etc : convertir en symbole.
- On drag'n'drop le Clip dans la scène depuis la bibliothèque. Dans la colonne de droite (en affichage classique de l'IDE Flash), on renomme cette occurrence.
- Ici, on utilisera les noms de StartScreen1 et GameOver1.
- Rappel : s'il y avait un problème avec l'événement du missclick (pour briser les combos en cas d'erreur du joueur), on pose cet événement sur l'image de background.

En début de code (après les import) :

```
StartScreen1.visible = true;  
GameOver1.visible = false;  
var jeuEnCours:Boolean = false;
```

À rajouter dans le MouseEvent de la cible : (La fonction qu'on a appelée « mouvement »)

```
StartScreen1.visible = false;  
jeuEnCours = true;
```

Dans l'ENTER_FRAME à la place de « frameCounter++; » :

```
if(jeuEnCours==true){  
    frameCounter++;  
}
```

Dans l'ENTER_FRAME à la place de « trace("Game Over"); » :

```
jeuEnCours = false;  
frameCounter = 0;  
cible.x = 0;  
cible.y = 0;  
cible.removeEventListener(MouseEvent.CLICK, mouvement)  
GameOver1.visible = true;
```

Boutons « Retry » et niveaux de difficulté

Recommencer la partie

Dernier problème : le Game Over coince littéralement la partie à cause de la disparition du MouseEvent. C'est pour cela qu'on a laissé de la place sur la HUD : il ne reste plus qu'à créer un bouton permettant de recommencer une partie.

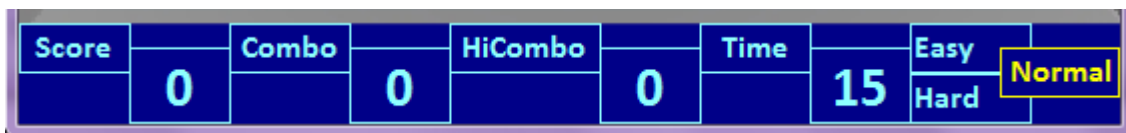
(Et s'il ne reste plus assez de place parce qu'on a mal calculé notre coup, on réajustera ça après.)

On va en profiter pour installer le choix du niveau de difficulté : pour gagner de la place sur la HUD, on va plutôt faire trois petits boutons Retry, chacun correspondant à un niveau de difficulté.

Enfin, le choix des difficultés étant proposé, on va pouvoir s'occuper d'une des mécaniques du jeu les plus importantes : le déplacement automatique de la cible, accéléré en fonction de l'avancement de la partie. Ce déplacement va être paramétré différemment selon le niveau de difficulté, c'est pourquoi on ne le code que maintenant.

Création des trois boutons

On va essayer de se servir du format de texte « format1 » pour les trois boutons, ça nous épargnera la moitié du travail et ce sera plus harmonieux avec le reste. En revanche, l'alignement du texte ne sera pas le même donc on va devoir recréer un format de texte. Plus précisément, on va les agencer comme ça :



(La couleur jaune pour la difficulté choisie, on ne va pas s'en préoccuper pour l'instant, ça fera partie des suppléments envisageables tout à la fin pour améliorer un peu le jeu.)

Bref, on part sur le modèle de TextField qu'on a déjà fait. Ensuite, on changera les emplacements des boîtes et l'alignement du texte (aligné à gauche pour Easy et Hard, aligné à droite pour Normal).

```
var boiteEasy:TextField = new TextField();  
var boiteHard:TextField = new TextField();  
var boiteNormal:TextField = new TextField();
```

Il est capital de créer la boîte Normal en dernier, et lui apporter ses modifications en dernier, pour qu'elle chevauche et écrase les deux autres.

On va donc créer un format de texte similaire en tous points à format1, sauf pour l'alignement de texte. On doit même en faire deux : un aligné à gauche pour Easy et Hard, un aligné à droite pour Normal.

```
var format3:TextFormat = new TextFormat();
format3.font = "Calibri";
format3.size = 16;
format3.bold = true;
format3.align = TextFormatAlign.LEFT;

var format4:TextFormat = new TextFormat();
format4.font = "Calibri";
format4.size = 16;
format4.bold = true;
format4.align = TextFormatAlign.RIGHT;
```

Ensuite, on les met en forme puis on y crée le texte comme d'habitude. On va pouvoir réutiliser la fonction de mise en forme créée au début pour le reste de la HUD :

```
miseEnForme(boiteEasy);
boiteEasy.x = 465;
boiteEasy.y = 350;
boiteEasy.text = "Easy";
boiteEasy.setTextFormat(format3);

miseEnForme(boiteHard);
boiteHard.x = 465;
boiteHard.y = 374;
boiteHard.text = "Hard";
boiteHard.setTextFormat(format3);

miseEnForme(boiteNormal);
boiteEasy.x = 510;
boiteEasy.y = 362;
boiteEasy.text = "Normal";
boiteEasy.setTextFormat(format4);
```

Il ne reste plus qu'à y ajouter le MouseEvent qui en fera de vrais boutons.

Remise à zéro totale

C'est le principe des boutons Retry : remettre le programme à l'état de « sur le point de commencer une partie ». Ce qui signifie que les quatre variables de la HUD sont remises à leurs valeurs initiales, ainsi que les quelques booléennes qui auraient pu changer pendant la partie.

(Et c'est au moment de la rédaction des MouseEvent qu'on est bien contents d'avoir listé toutes les variables en début de code, ça nous évite d'oublier d'en réinitialiser une et de tout relire.)

On va aussi mettre, dans le MouseEvent de chacun des boutons, le changement de difficulté : pour cela, on va se servir de deux booléennes (que l'on va devoir créer en début de code).

```
var easyMode:Boolean = false;  
var hardMode:Boolean = false;
```

Le fonctionnement conjugué des deux booléennes est assez compréhensible :

- Si les deux sont false, on est en mode Normal. (par défaut)
- Si l'une des deux est true, on est dans le mode indiqué par cette booléenne (Easy ou Hard).
- On va s'arranger pour que le cas où les deux sont true n'arrive jamais.

Bref, on crée le MouseEvent de chacun des boutons :

- Ça va prendre beaucoup de place parce qu'on ne peut pas imbriquer des fonctions. Donc, impossible de stocker dans des fonctions les lignes de codes redondantes.
- Il ne faut pas non plus oublier de rafraîchir l'affichage sur la HUD une fois les variables remises à leurs valeurs initiales.
- Et n'oublions pas non plus de remettre l'EventListener sur la cible : on l'avait retiré lors du game over. La ligne du « addEventListener » suffit, la fonction « mouvement » ayant déjà été déclarée et définie plus tôt.

L'exemple suivant est le MouseEvent correspondant au bouton Easy. Pour les deux autres, il faudra donc penser à appeler la fonction de l'événement différemment, et modifier les booléennes easyMode et hardMode en fonction de ce que l'on veut avoir.

```
boiteEasy.addEventListener(MouseEvent.CLICK, retryEasy);  
function retryEasy(e:MouseEvent) {  
    score = 0;  
    boiteScore2.text = score.toString();  
    boiteScore2.setTextFormat(format2);  
    combo = 0;  
    boiteCombo2.text = combo.toString();  
    boiteCombo2.setTextFormat(format2);  
    hiCombo = 0;  
    boiteHiCombo2.text = hiCombo.toString();  
    boiteHiCombo2.setTextFormat(format2);  
    time = 15;  
    boiteTemps2.text = time.toString();  
    boiteTemps2.setTextFormat(format2);  
    easyMode = true;  
    hardMode = false;  
    prolongation = false; //Facile à oublier, celui-là.  
    GameOver1.visible = false;  
    StartScreen1.visible = true;  
    cible.addEventListener(MouseEvent.CLICK, mouvement);  
}
```

Récapitulatif : Boutons « Retry » et niveaux de difficulté

En début de code (après les import) :

```
var easyMode:Boolean = false;
var hardMode:Boolean = false;
```

À la suite du reste :

```
var boiteEasy:TextField = new TextField();
var boiteHard:TextField = new TextField();
var boiteNormal:TextField = new TextField();

var format3:TextFormat = new TextFormat();
format3.font = "Calibri";
format3.size = 16;
format3.bold = true;
format3.align = TextFormatAlign.LEFT;

var format4:TextFormat = new TextFormat();
format4.font = "Calibri";
format4.size = 16;
format4.bold = true;
format4.align = TextFormatAlign.RIGHT;

miseEnForme(boiteEasy);
boiteEasy.x = 465;
boiteEasy.y = 350;
boiteEasy.text = "Easy";
boiteEasy.setTextFormat(format3);

miseEnForme(boiteHard);
boiteHard.x = 465;
boiteHard.y = 374;
boiteHard.text = "Hard";
boiteHard.setTextFormat(format3);

miseEnForme(boiteNormal);
boiteEasy.x = 510;
boiteEasy.y = 362;
boiteEasy.text = "Normal";
boiteEasy.setTextFormat(format4);
```

```
boiteEasy.addEventListener(MouseEvent.CLICK, retryEasy);
function retryEasy(e:MouseEvent) {
    score = 0;
    boiteScore2.text = score.toString();
    boiteScore2.setTextFormat(format2);
    combo = 0;
    boiteCombo2.text = combo.toString();
    boiteCombo2.setTextFormat(format2);
    hiCombo = 0;
    boiteHiCombo2.text = hiCombo.toString();
    boiteHiCombo2.setTextFormat(format2);
    time = 15;
    boiteTemps2.text = time.toString();
    boiteTemps2.setTextFormat(format2);
    easyMode = true;
    hardMode = false;
    prolongation = false;
    GameOver1.visible = false;
    StartScreen1.visible = true;
    cible.addEventListener(MouseEvent.CLICK, mouvement);
}

boiteHard.addEventListener(MouseEvent.CLICK, retryHard);
function retryHard(e:MouseEvent) {
    score = 0;
    boiteScore2.text = score.toString();
    boiteScore2.setTextFormat(format2);
    combo = 0;
    boiteCombo2.text = combo.toString();
    boiteCombo2.setTextFormat(format2);
    hiCombo = 0;
    boiteHiCombo2.text = hiCombo.toString();
    boiteHiCombo2.setTextFormat(format2);
    time = 15;
    boiteTemps2.text = time.toString();
    boiteTemps2.setTextFormat(format2);
    easyMode = false;
    hardMode = true;
    prolongation = false;
    GameOver1.visible = false;
    StartScreen1.visible = true;
    cible.addEventListener(MouseEvent.CLICK, mouvement);
}
```

```
boiteNormal.addEventListener(MouseEvent.CLICK, retryNormal);
function retryNormal(e:MouseEvent) {
    score = 0;
    boiteScore2.text = score.toString();
    boiteScore2.setTextFormat(format2);
    combo = 0;
    boiteCombo2.text = combo.toString();
    boiteCombo2.setTextFormat(format2);
    hiCombo = 0;
    boiteHiCombo2.text = hiCombo.toString();
    boiteHiCombo2.setTextFormat(format2);
    time = 15;
    boiteTemps2.text = time.toString();
    boiteTemps2.setTextFormat(format2);
    easyMode = false;
    hardMode = false;
    prolongation = false;
    GameOver1.visible = false;
    StartScreen1.visible = true;
    cible.addEventListener(MouseEvent.CLICK, mouvement);
}
```

Déplacement automatique de la cible

Planifier le déplacement automatique

Notre objectif est le suivant : on veut ajouter une mécanique qui complexifie le jeu et le rend de plus en plus difficile avec le temps qui passe. On a opté pour un déplacement aléatoire de la cible (comme lorsque l'on clique dessus), mais qui se fasse de façon automatique. Ce déplacement doit se faire de plus en plus fréquent au cours de l'avancement de la partie, et son rythme doit aussi changer selon la difficulté.

Cette cadence ne sera donc pas aléatoire. Le plus simple est de « programmer » les instants où la cible va bouger, en les identifiant à la frame près. Pour ça, on a le frameCounter qu'on avait créé pour l'écoulement du temps : le numéro de frame va nous permettre d'identifier chaque instant de la partie.

On n'a donc plus qu'à lister tous les instants (toutes les frames) où l'on veut voir la cible bouger. Dans notre ENTER_FRAME, on fera ensuite bouger la cible dès que le numéro de frame coïncide.

Fonctionnement des Array

On pourrait tout simplement mettre « si le numéro de la frame vaut ça, ou alors ça, ou alors ça, ou alors ça, ou alors ça... alors la cible bouge ». C'est beaucoup trop long, c'est idiot, et on risque beaucoup plus de faire une erreur d'inattention. Alors comment procéder ?

On va créer une liste dans laquelle on va mettre tous les numéros de frame auxquels on veut voir bouger la cible. Ensuite, on dit à Flash de lire toute la liste et de bouger la cible si la frame actuelle correspond à l'un des numéros de la liste.

Cette liste, c'est un Array. On en crée trois (un par difficulté) au début de notre code avec les autres variables :

```
var arrayEasy:Array =  
[192,384,512,640,768,864,960,1056,1152,1224,1296,1368,1440,1512];  
//C'est une seule ligne, mais elle est longue vu le nombre  
d'éléments de la liste. Il y a retour à la ligne parce que je  
le tape sous Word, mais tout tiendra en une ligne sous Flash.  
var arrayHard:Array =  
[128,224,320,392,464,536,590,644,698,752,806,860,900,940,980,1020,1060,1100,1140,1164,1184,1212,1236,1260,1284,1308,1332,1350,1368,1386,1404,1422,1440,1458,1476,1494];  
//Ca aurait été trois fois plus long sans faire d'Array..
```

```
var arrayNormal:Array =  
[128,256,352,448,544,616,688,760,832,886,940,994,1048,1102,114  
2,1182,1222,1262,1302,1342,1366,1390,1414,1438,1462,1486,1510]  
;
```

Bref. Ça a l'air immonde parce qu'on y a mis des dizaines d'éléments donc ça prend beaucoup de place. Mais sa structure est très simple :

- Les crochets délimitent l'Array
- Les virgules séparent ses différents éléments.
- Flash a compris qu'on y mettait des Number, on a pas besoin de créer une tonne de variables.

On aurait aussi bien pu créer des Array vides (avec `new Array()`) et les remplir après, mais on aurait à peine gagné en lisibilité dans le code. Sous Flash, ce sera assez compréhensible comme ça.

Parcourir l'Array avec une boucle en « for »

Rappel : « for » permet de créer une boucle, c'est-à-dire une portion de code qui sera relue et ré-exécutée par Flash autant de fois qu'on le souhaite. On utilise très souvent les Array et les boucles ensemble : cela permet de **parcourir** l'Array.

Autrement dit, Flash regarde le premier élément de l'Array, fait ce qu'on lui demande avec, puis passe au second élément, fait la même chose avec, etc.

Notre schéma d'action sera le suivant : à chaque frame (donc dans l'ENTER_FRAME), on va faire une boucle pour que Flash examine chaque numéro de frame contenu dans l'Array qui nous intéresse (easy, normal ou hard). Si Flash y trouve le numéro correspondant au numéro de la frame actuelle, il bouge la cible.

On va commencer par le plus simple : déterminer quelle boucle lancer selon la difficulté.

```
if(easyMode==true){  
    //Notre boucle Easy ici  
}else if(hardMode==true){  
    //Notre boucle Hard ici  
}else{  
    //Notre boucle Normal ici  
}
```

Ensuite, les boucles. (On va en faire une, les deux autres suivent exactement le même principe.)

Création des boucles

Lorsque l'on crée une boucle avec `for`, il y a trois paramètres à indiquer en une ligne (séparés par des points virgules) :

- Créer une variable (int) qui va servir à compter le nombre d'itérations de la boucle
- Indiquer la condition pour laquelle la boucle va se poursuivre
- Indiquer quel changement opérer sur la variable à chaque itération

En gros, on crée la variable, on indique jusqu'à quelle valeur de la variable la boucle va se refaire, et on précise qu'on va incrémenter la variable à chaque itération.

C'est donc dans le deuxième paramètre (la condition) que l'on va indiquer combien de fois on veut voir la boucle se répéter. Ce nombre d'itérations, dans notre cas, va être le nombre d'éléments contenus dans l'Array : sa **longueur**. Ce nombre est une simple propriété de l'Array, la propriété `.length` : par exemple, `arrayEasy.length`.

```
for(var i:int= 0; i<arrayEasy.length; i++){  
    if(arrayEasy[i]==frameCounter){  
        cible.x = (Math.random()*360)-180;  
        cible.y = (Math.random()*385)-255;  
    }  
}
```

```
for(var j:int= 0; j<arrayHard.length; j++){  
    if(arrayHard[j]==frameCounter){  
        cible.x = (Math.random()*360)-180;  
        cible.y = (Math.random()*385)-255;  
    }  
}
```

```
for(var k:int= 0; k<arrayNormal.length; k++){  
    if(arrayNormal[k]==frameCounter){  
        cible.x = (Math.random()*360)-180;  
        cible.y = (Math.random()*385)-255;  
    }  
}
```

On a nos trois boucles : on les met dans les `if` précédents, et c'est terminé pour le réglage de la difficulté.

`arrayEasy[i]` correspond à l'élément de l'Array qui porte l'index *i* : c'est donc, disons, la "i-ème" valeur que l'on y a rentrée. Or, comme indiqué dans la ligne de création de la boucle, *i* vaudra d'abord 0, puis 1 à la prochaine itération de la boucle, puis 2, puis 3... Et s'arrêtera à (par exemple) 13 si l'on a mis 14 éléments dans l'Array (ça commence à 0 et non à 1, attention).

Récapitulatif : Déplacement automatique de la cible

En début de code (après les import) :

```
var arrayEasy:Array =
[192,384,512,640,768,864,960,1056,1152,1224,1296,1368,1440,1512];
var arrayHard:Array =
[128,224,320,392,464,536,590,644,698,752,806,860,900,940,980,1020,1060,1100,1140,1164,1184,1212,1236,1260,1284,1308,1332,1350,1368,1386,1404,1422,1440,1458,1476,1494];
var arrayNormal:Array =
[128,256,352,448,544,616,688,760,832,886,940,994,1048,1102,1142,1182,1222,1262,1302,1342,1366,1390,1414,1438,1462,1486,1510];
;
```

Dans l'ENTER_FRAME :

```
if(easyMode==true){
    for(var i:int= 0; i<arrayEasy.length; i++){
        if(arrayEasy[i]==frameCounter){
            cible.x = (Math.random()*360)-180;
            cible.y = (Math.random()*385)-255;
        }
    }
}else if(hardMode==true){
    for(var j:int= 0; j<arrayHard.length; j++){
        if(arrayHard[j]==frameCounter){
            cible.x = (Math.random()*360)-180;
            cible.y = (Math.random()*385)-255;
        }
    }
}else{
    for(var k:int= 0; k<arrayNormal.length; k++){
        if(arrayNormal[k]==frameCounter){
            cible.x = (Math.random()*360)-180;
            cible.y = (Math.random()*385)-255;
        }
    }
}
```

Derniers ajustements

Il n'y aura pas de récapitulatif ici. Parce qu'on peut faire tout ce qu'on veut pour modifier le jeu : ses graphismes, son univers, ses mécaniques... Bref, ce qui va suivre sert à titre d'exemple.

Parfaire la HUD

Au point où on en est, on a déjà quelque chose de fonctionnel et jouable. Mais de gros défauts esthétiques subsistent dans la HUD (bien que la version finale ne soit pas non plus un chef d'oeuvre) :

- Des espaces blancs là où il n'y a ni background (écran d'oscilloscope) ni TextField
- Le nombre de pixels peut ne pas correspondre.

Pour les pixels : on fait des tests pour ajuster, on change les valeurs rentrées plus tôt.

Pour les espaces blancs : c'est la couleur par défaut du fond du fichier, parce qu'on a rien mis à cet endroit-là. On n'a qu'à changer cette couleur par défaut et mettre le #000088 du fond des TextField, et ça se fait dans l'interface graphique de Flash : **Ctrl+J** (paramètres du document) permet de modifier les dimensions et la couleur de fond.

Autre problème possible : le joueur n'aura pas le temps de regarder ses paramètres dans la HUD. Résolution : lorsqu'il faut véhiculer un message important (partie bientôt finie, prolongations lancées, combo brisé, etc), on change la couleur du TextField correspondant pour une couleur vive. C'est ça qui importera, et non la valeur que le joueur pourra faire l'effort de lire s'il le veut/peut.

Creuser les écarts de difficulté

Pour l'instant, le niveau de difficulté ne modifie que la fréquence des déplacements automatiques. On peut aussi modifier la taille de la cible (son rayon) : pour cela, il suffit de retracer un autre cercle dans notre Sprite à chaque clic sur le bouton Retry (donc de sélection de difficulté). En gros, on rajoute ça dans les MouseEvents des trois boutons.

Effets sonores

(Pas encore testé)

- On crée un objet vide, de type Sound, qui contiendra notre son.
- On crée un objet de type URLRequest : c'est notre fichier son (on donne son chemin exact : une url où il est stocké online, ou un chemin de type D:/Musique/Sons/Etc)
- On met l'URLRequest dans le conteneur Sound créé juste avant.
- On crée un canal (SoundChannel) vide. Pour faire simple : pour jouer notre son, on doit le faire depuis un canal. Pour superposer des sons/musiques, on doit créer autant de canaux que nécessaire (exemple pour une mélodie simple : percussions + violon + piano = trois canaux). Dans notre cas, ce sera donc un canal par mp3, c'est-à-dire un canal effets sonores et un canal musique.
- On joue le Sound (qui contient l'URLRequest) dans le SoundChannel.

Donc, cinq lignes seulement. Les quatre premières sont à mettre n'importe quand (donc au début, là où on a déclaré toutes nos variables), la dernière (le *play()*) est à mettre au moment où on veut le faire commencer à jouer (logique).

Enfin, en toute logique, la méthode stop() doit aussi exister si besoin.

```
var conteneurSon:Sound = new Sound();  
var monSon:URLRequest = new URLRequest(D:/Jeux/CRAD/bgm.mp3);  
var canall:SoundChannel = new SoundChannel();  
conteneurSon.load(monSon);  
  
//Et, plus tard, là où on veut le faire jouer...  
  
canall = conteneurSon.play();
```