# **Développer un jeu vidéo ... rétro ?**

## **Un peu de magie**

Si vous lisez ceci, c'est que, comme moi à une époque, vous avez envie de perçer ce mystère : Comment ont-ils réussi à coder ce jeu vidéo ?

Un bon jeu vidéo c'est comme un bon tour de magie, ça vous plonge dans un univers où ce qui semble impossible devient réel ... Derrière tout ça il y a du travail et de la technique. Que vous vouliez devenir magicien ou comprendre le tour, il va falloir y passer du temps, et utiliser les bons accessoires ...

L'idée de ce manuel est de vous fournir toutes les informations pour comprendre et maîtriser à votre tour l'implémentation des jeux vidéo.

## **La plateforme**

Avant de se lancer il va falloir faire deux choix. Pour rendre ce manuel possible j'ai fait ces choix pour vous :

* la plateforme d'exécution sera le Thomson TO8/TO8D/TO9+
* le langage de programmation sera l'assembleur

Rien ne vous empêche cependant d'utiliser ce manuel pour d'autres plateformes et d'autres langages, les concepts restent les mêmes.

Pourquoi ces choix ? Un manuel sur la programmation de jeu Megadrive ou NES aurait été tout aussi pertinent. Pas de panique, justement le code mis en oeuvre dans ce manuel s'inspire de celui des jeux sur console, il est simplement transposé à une autre machine.

Cette plateforme est intéressante car elle ne bénéficie pas d'accélération matérielle particulière, de ce fait le code sera relativement peu dépendant du matériel, même si le code sera bien entendu spécifique au processeur Motorola 6809. De plus, c'est une machine très répandue en France, vous n'aurez pas de mal à vous la procurer.

La machine bénéficie des caractéristiques suivantes :

* processeur Motorola 6809 @1Mhz
* 256Ko de RAM (512Ko au total avec l'extension de RAM)
* port cartouche
* port extension (midi, carte son)
* port 2 manettes (2 boutons)
* port crayon optique
* sortie audio (6 bits)
* sortie vidéo RVB (50Hz)
* palette de couleurs programmable
* timer irq

## **Le processeur Motorola 6809**

L'apprentissage de l'assembleur est un prérequis indispensable à la lecture de ce manuel. Bonne nouvelle : le 6809 est un processeur simple à aborder.

à rédiger - historique

à rédiger - architecture et fonctionnement

à rédiger - instructions

à rédiger - ressources (livres et sites internet)

## **Les modèles de conception**

en cours de rédaction - La structure d’un programme

à rédiger - La boucle principale

en cours de rédaction - Le double buffering

en cours de rédaction - Le positionnement écran

en cours de rédaction - [La gestion des objets](https://github.com/wide-dot/thomson-to8-game-engine/blob/main/doc/chapters/object.md)

en cours de rédaction - [La gestion des collisions](https://github.com/wide-dot/thomson-to8-game-engine/blob/main/doc/chapters/collision.md)

à rédiger - Les interruptions

à rédiger - Les manettes

à rédiger - L'affichage de sprites

à rédiger - Le scrolling

à rédiger - La streaming

à rédiger - La gestion des cartes son et midi

en cours de rédaction - [Les variables globales](https://github.com/wide-dot/thomson-to8-game-engine/blob/main/doc/chapters/global.md)

## **Les composants**

à rédiger - Le déplacement des sprites

à rédiger - Les effets de palette

à rédiger - Les contrôles utilisateur

## **Les routines pour Thomson TO8**

à rédiger - Le lancement automatique

à rédiger - La lecture de disquette

à rédiger - L'accès aux manettes

à rédiger - L'accès au clavier

à rédiger - La palette de couleur

**Structure d’un programme**

Il y a trois concepts à comprendre :

- Game Mode

- Act

- Object

Ces notions permettent le chargement et l'exécution de ressources depuis la RAM.

**Game Mode**

Le Game Mode est le programme chargé en mémoire non commutable ($6100-$9EFF) Remarque : $9F00-$9FFF est utilisé pour des données globales/temporaires et est initialisé par jsr InitGlobals.

A chaque chargement d'un Game Mode, le loader va lire la disquette et écraser le programme en mémoire non commutable ($6100-$9EFF) par un autre. Si le nouveau programme est plus petit, il restera les données de l'ancien (non utilisées) à la suite du nouveau, on n’efface pas les données de toute la page avant. On associe au Game Mode un ensemble d'Objects.

Dans la version FD et T2 ces objets sont chargés en même temps que le Game Mode, mais sont placés dans les pages de RAM en zone commutable, de manière à pouvoir être exécutés par recouvrement de la zone cartouche. Ainsi lorsque l'on charge un nouveau Game Mode, les objets sont effacés et de nouveaux sont chargés.

On voit déjà un premier problème : J'ai un personnage dont les images font 150ko, je vais pas les recharger entre chaque niveau de jeu ! Pour palier à ça j'ai mis en place une notion de Game Mode "Common" dont on aura l'occasion de reparler plus tard, mais en gros ça permet de charger des Objets en début de RAM (les premières pages) et de les conserver entre chaque Game Mode.

Le futur ... Cette problématique est celle qui m'a fait revoir ma copie et mettre en place un système différent de "Packages" pour la V2. La notion de Game Mode et d'Act disparaît : En V2 on fera la définition de Packages qui sont constitués de code binaire ou de données dont on définit :

- L'emplacement source sur un média

- L'emplacement destination en RAM ou ROM

On pourra également reloger du code par auto modification (donc sans les contraintes des mécanismes actuels de type , pcr) ... bref simplification du concept La notion d'objet persistera en V2 je vous rassure et on verra certainement apparaître d'autres notions pour indexer les ressources.

**Act**

Un Act est un preset défini dans le builder, qui permet de positionner une palette, une couleur de bordure et d'initialiser la mémoire vidéo avec soit une image, soit une couleur unies.

C'est pourquoi vous voyez un jsr LoadAct au démarrage du Game Mode Il applique les presets définis pour l'Act par défaut (actBoot dans leproperties) C'est un des premiers concepts ajouté dans le moteur de jeu et il est mort né. En particulier il n'y a pas de code qui exploite la possibilité de charger les autres presets que celui par défaut.

Ce concept d'Act doit être décommissionné pour les raisons suivantes :

- Il introduit dans le builder de la génération de code (et oui le LoadAct est généré dedans !!!)

- le concept est foireux, il y a peu de routines appelées dedans et on peut vouloir intialiser de tant de manières différentes …

- ça fait manipuler des properties alors que ce serait tellement pratique de le faire directement dans le code ... C'est un mauvais choix de design ... a ignorer (modifié)

### **Double Buffering**

### 

### **Le positionnement écran**

La zone où sera affichée l’ensemble des éléments graphiques (menus, tableaux de bords, scores, sprites, etc.) occupe virtuellement un espace plus grand que l’écran réel.

Cela permet de donner “vie” à certains éléments en dehors de l’affichage réel et de suivre leur propre logique de vie, même hors des yeux du joueur.

Par exemple un garde qui patrouille devant une porte de château, il peut temporairement disparaître de l’affichage, mais il existe quand même et son comportement peut être important

Afin d’associer un concept à un certain vocabulaire :

* l’écran réel : **Screen**
* la surface virtuelle : **Playfield**

La zone “écran virtuel” (Playfield) varie en fonction des résolutions réelles (Screen) choisies :

| **Mode graphique** | **Résolution réelle** | **Taille de la surface écran virtuel** |
| --- | --- | --- |
| BM16 | 160x200 | 256x256 |
| << à compléter >> | << à compléter >> | << à compléter >> |

Ce qui est affiché à l’écran est strictement **centré** par rapport à zone virtuelle. On peut considérer que l’écran réel est le “*viewport*” de la surface écran virtuelle.



Chaque objet possède des coordonnées virtuelles X et Y dans le repère du ***playfield*** qui seront transformées automatiquement en coordonnées réelles, s’il est visible, dans le répère du ***screen***. Ce mécanisme est géré automatiquement par l’engine.

Les coordonnées partent de 0,0 dans le coin supérieur gauche du ***playfield***, pour atteindre la limite de 255,255 en bas à droite.

**Nota** : les coordonnées des objets placés dans le ***playfield*** sont exprimées au moyen de nombre à virgule fixe : 8 bits pour la partie entière et 8 bits pour la partie fractionnaire. Cela permet de faire des animations fluides, sans saccade. Ces notions seront abordées plus loin.

# **La gestion des objets**

## Définition d’un objet (obj.properties)

### Assembly code

The sprite is firstly associated to its own assembly code, using the following syntax :

code=./objects/enemies/01/bink/obj.asm

(The root is the project itself)

### Frames

Each frame of a sprite is to be specified with the following syntax (one line for each frame) :

sprite.**Img\_bink\_0**=**./objects/enemies/01/bink/images/bink\_00.png**;**NB0,NB1**

| Object | Definition | Description |
| --- | --- | --- |
| sprite.**Img\_bink\_0** | The sprite name | The object starts with sprite. followed by a chosen name (for example, the sprite name and frame number)  In the given example, “bink” represents the sprite name, and 0 its frame. |
| **./objects/enemies/01/bink/images/bink\_00.png** | The path to the PNG file | The path is always relative to the project root folder. |
| **NB0,NB1** | Compilation options | You can define how the engine is going to compile the sprite. You can select multiple options, using a comma in between each.  First option :   * N specifies a “normal” sprite, oriented as you designed it * X specified a horizontal flip of the sprite * Y specifies a vertical flip of the sprite * XY specifies both a horizontal and vertical flip of the sprite   Second option :   * B (background) specifies that the backgroun of the sprite will need to be saved and restored * D (draw) means that the background does not need to be saved (and restored)   Third option :   * 0 : Create the sprite at position 0 * 1 : Create the sprite at position 1 (shift by 1 pixel to the right). This is to allow a single pixel movement   Example :  I want my sprite frame to be compiled with the possibility of moving by 1 pixel (hence with 1 shift), with saving of the background and its equivalent with horizontal flipping (mirror) as my sprite can move right or left with a flipped design.  My sprite options would therefore be :  NB0,NB1,XB0,XB1 |

### Animations

The sequence of animation between a same sprite frames can be specified with the following syntax :

animation.**Ani\_bink**=**2**;**Img\_bink\_3;Img\_bink\_2;Img\_bink\_3;Img\_bink\_1;\_resetAnim**

| Object | Definition | Description |
| --- | --- | --- |
| sprite.**Ani\_bink** | Assembly jump point | This defines where, in the assembly code (obj.asm) the sprite will be executed. |
| **2** | Frame rate | This defines when the sprite is rotated to its next frame :   * 1 means every game frame * 2 means every other game frames * 3 means every 3 game frames * … etc … |
| **Img\_bink\_3;Img\_bink\_2;Img\_bink\_3;Img\_bink\_1;\_resetAnim** | Animation sequence | This specifies the frames order the sprite will be animated.  Each frame name refers to the frames created (See “Frames” section above) and are separated by a semicolon ;  The animation is closed by \_resetAnim  \_nextRoutine |

## Un embryon d'objet

Dans la boucle principale d'un programme, on peut exécuter librement un ensemble de routines. Un problème se pose cependant lorsque l'on veut exécuter plusieurs fois la même routine, mais associée à un ensemble de données différent.

Pour répondre à ce besoin, la première chose qui nous vient à l'esprit est de passer un pointeur vers un ensemble de données en paramètre de cette routine.

### Avec le registre U

Nous allons utiliser le registre U.

; ...

ldu #data1

jsr routine

ldu #data2

jsr routine

; ...

routine

ldd x\_pos,u ; horizontal position

; processing ...

ldd y\_pos,u ; vertical position

; processing ...

rts

; offsets to data structure

x\_pos equ 0

y\_pos equ 2

data1

fill 0,4 ; an array of 4 bytes

data2

fill 0,4 ; an array of 4 bytes

### Avec le registre DP

Une autre stratégie est d'utiliser le registre DP (Direct Page). Le traitement sera alors plus rapide, l'inconvénient étant que la structure de donnée a une taille fixe de 256 octets et doit être positionnée en mémoire à une adresse multiple de 256.

; ...

lda #data1

tfr a,dp

jsr routine

lda #data2

tfr a,dp

jsr routine

; ...

routine

ldd <x\_pos ; horizontal position

; processing ...

ldd <y\_pos ; vertical position

; processing ...

rts

; offsets to data structure

x\_pos equ 0

y\_pos equ 2

align 256

data1

fill 0,4 ; an array of 4 bytes

align 256

data2

fill 0,4 ; an array of 4 bytes

Même si ce principe reste rudimentaire on peut considérer que c'est un début de gestion d'objet, en effet nous avons ici un comportement associé à un ensemble d'attributs propres, dont les données sont persistantes.

Cela n'est cependant pas suffisant si l'on souhaite créer ces objets à plus grande échelle et de manière dynamique.

## Instancier un objet

L'instanciation d'objet permet de pouvoir conserver plusieurs états d'un même objet. On va donc attribuer à chaque instance un espace mémoire pour y stocker ses données. Dans le moteur de jeu, ce mécanisme a été mis en place pour stocker les données d'état d'un sprite et stocker des données utilisateur.

Si vous utilisez ce mécanisme d'instanciation, l'objet sera automatiquement exécuté par RunObjects.

Si vous ne voulez pas qu'il soit exécuté par RunObjects, vous pouvez définir un espace mémoire dédié et lancer manuellement l'objet avec la macro suivante :

\_RunObject ObjID\_Player1,#player1

Vous aurez alors défini dans le Game Mode un espace de stockage :

player1 fill 0,<taille mémoire>

Pour instancier un objet, on utilise jsr LoadObject\_u ou la variante \_x

La routine retourne en u ou x l'adresse mémoire allouée.

Il est important juste après de positionner l'id dans cette zone. Si vous ne positionnez pas l'id et faites l'appel à un autre LoadObject, il va allouer de nouveau l'espace précédent car c'est l'id qui détermine si le slot est libre (id=0).

jsr LoadObject\_x ; Demande l’allocation d’un objet, placé en x

beq pasdobject ; Un objet n’a pu être alloué

lda #ObjID\_scantfire ; Positionnement de ...

sta id,x ; ... id de l’objet

Le fait d'allouer la mémoire enregistre l'objet dans une liste d'objets à exécuter lors du RunObjects et seulement lors de l'appel à cette routine (les routines d'affichage utilisent une autre liste pour traiter les objets)

Tant que vous ne référencez pas votre objet comme sprite (voir plus bas), les données de l'objet sont libres d'utilisation (vous pouvez ré-utiliser tous les emplacements habituellement utilisés par x\_pos, y\_pos ...). Pour cela il suffit de définir vos propres offsets (comme déclarés pour les sprites dans constants.asm).

L'objet sera donc exécuté comme les autres mais ne sera pas traité par les routines graphiques.

## 

## Détruire un objet

## Définir des attributs

## Définir des comportements

## Exécuter un comportement

## Utiliser la Direct Page

Un moyen d'accéder plus rapidement aux données est l'utilisation de la Direct Page. Ce principe peut être mis en œuvre pour les données de l'objet du joueur principal. Dans ce cas de figure on ne fait pas appel à la routine d'allocation d'objet. La méthode RunObjects n'exécute donc pas le code. Première étape, déclarer l'objet :

lda #ObjID\_Player1

sta id+dp

On peut positionner des valeurs pour les variables de la même manière :

ldd #$60

std x\_pos+dp

ldd #$028F

std y\_pos+dp

Vous constatez donc qu’ici, on a pas besoin de U pour accéder aux données objets. On utilise l'offset de la variable + la constante dp

Seconde étape, exécuter le code objet Dans la boucle principale, ajouter ce code :

\_RunObject ObjID\_Player1,#dp

Dans le code objet l'utilisation de ,u fonctionnera toujours (on charge u avec la zone mémoire pointée par dp) , ça permet de garder la compatibilité avec les routines d'animation ... mais dans votre code utilisez x\_pos+dp pour aller plus vite

## Constantes d’un objet

| Nom | Position | Octet | Définition |
| --- | --- | --- | --- |
| id | 0 | 1 | reference to object model id (ObjID\_) (0: free slot) |
| subtype | 1 | 1 | reference to object subtype (Sub\_) DEPENDENCY subtype must follow id |
| render\_flags | 2 | 1 | Voir ci apres |
| run\_object\_prev | 3 | 2 | previous object to update when deleting current object |
| run\_object\_next | 5 | 2 | next object to run by RunObjects |
| priority | 7 | 1 | display priority (0: nothing to display, 1:front, ..., 8:back) |
| anim | 8 | 2 | reference to current animation (Ani\_) |
| prev\_anim | 10 | 2 | reference to previous animation (Ani\_) |
| anim\_frame | 12 | 1 | index of current frame in animation |
| anim\_frame\_duration | 13 | 1 | number of frames for each image in animation, range: 00-7F (0-127), 0 means display only during one frame |
| anim\_flags | 14 | 1 | byte offset to reference an anim\_flags LUT (adv) / store a link flag (non adv) |
| status\_flags | 15 | 1 | orientation of sprite, is applied to animation xmirror flag during AnimateSprite - Voir ci dessous |
| image\_set | 16 | 2 | reference to current image (Img\_) (0000 if no image) |
| x\_pos | 18 | 2 | x playfield coordinate |
| x\_sub | 20 | 1 | x subpixel (1/256 of a pixel), must follow x\_pos in data structure |
| y\_pos | 21 | 2 | y playfield coordinate |
| y\_sub | 23 | 1 | y subpixel (1/256 of a pixel), must follow y\_pos in data structure |
| xy\_pixel | 24 | 2 | x and y screen coordinate |
| x\_pixel | 24 | 1 | x screen coordinate |
| y\_pixel | 25 | 1 | y screen coordinate |
| x\_vel | 26 | 2 | horizontal velocity |
| y\_vel | 28 | 2 | vertical velocity |
| x\_acl | 30 | 2 | horizontal gravity |
| y\_acl | 32 | 2 | vertical gravity |
| routine | 34 | 1 | index of current object routine |
| routine\_secondary | 35 | 1 | index of current secondary routine |
| routine\_tertiary | 36 | 1 | index of current tertiary routine |
| routine\_quaternary | 37 | 1 | index of current quaternary routine |

## Description des flags

### Render Flags

#### Background Erase Pack

| Nom | Valeur | Bit | Définition |
| --- | --- | --- | --- |
| render\_xmirror\_mask | $01 | 0 | DEPENDENCY should be bit 0 - tell display engine to mirror sprite on horizontal axis |
| render\_ymirror\_mask | $02 | 1 | DEPENDENCY should be bit 1 - tell display engine to mirror sprite on vertical axis |
| render\_overlay\_mask | $04 | 2 | DEPENDENCY should be bit 2 - compiled sprite with no background save |
| render\_playfieldcoord\_mask | $08 | 3 | tell display engine to use playfield (1) or screen (0) coordinates |
| render\_xloop\_mask | $10 | 4 | (in screen coordinate) tell display engine to hide sprite when x is out of screen (0) or to display (1) |
| render\_todelete\_mask | $20 | 5 | tell display engine to delete sprite and clear OST for this object |
| render\_subobjects\_mask | $40 | 6 | tell display engine to render subobjects for this object |
| render\_hide\_mask | $80 | 7 | tell display engine to hide sprite (keep priority and mapping\_frame) |

#### Overlay Pack

| Nom | Valeur | Bit | Définition |
| --- | --- | --- | --- |
| render\_xmirror\_mask | $01 | 0 | DEPENDENCY should be bit 0 - tell display engine to mirror sprite on horizontal axis |
| render\_ymirror\_mask | $02 | 1 | DEPENDENCY should be bit 1 - tell display engine to mirror sprite on vertical axis |
| *Non utilisé* | *NA* | 2 | *Non utilisé* |
| render\_playfieldcoord\_mask | $08 | 3 | tell display engine to use playfield (1) or screen (0) coordinates |
| render\_xloop\_mask | $10 | 4 | (in screen coordinate) tell display engine to hide sprite when x is out of screen (0) or to display (1) |
| render\_todelete\_mask | $20 | 5 | tell display engine to delete sprite and clear OST for this object |
| render\_subobjects\_mask | $40 | 6 | tell display engine to render subobjects for this object |
| render\_hide\_mask | $80 | 7 | tell display engine to hide sprite (keep priority and mapping\_frame) |

### Status Flags

| Nom | Valeur | Bit | Définition |
| --- | --- | --- | --- |
| status\_xflip\_mask | $01 | 0 | X Flip |
| status\_yflip\_mask | $02 | 1 | Y Flip |
| *Non utilisés* | *NA* | 2-7 | *Non utilisés* |

**A intégrer au chapitre :**

**Object**

Un Objet permet de charger du code et des données en RAM dans des pages commutables. Il est constitué d'un code asm qui doit tenir dans une page de 16Ko. Toutes les étiquettes définies dans ce code sont locales, il n'y a à l'heure actuelle pas de possibilité de les rendre visibles aux autres code sources (ça sera différent en V2 car on utilisera le linker de LWASM). Il n'y a pas d'obligation d'associer autre chose à un objet qu'un code source c'est le seul prérequis. Le builder va attribuer un id à l'objet en utilisant l'ordre de déclaration dans le properties du Game Mode. Cet id c'est un index sur deux tables qui sont ajoutées au code asm du Game Mode (en fin du programme). Vous pouvez les voir dans le .lst produit dans le répertoire GeneratedCode.

***main.lst***

…

ObjID\_Player1 equ 1

ObjID\_Weapon1 equ 2

ObjID\_Weapon2 equ 3

ObjID\_Weapon3 equ 4

…

Obj\_Index\_Page

fcb $00 ; id 0 is not used

fcb $6B

fcb $64

fcb $64

…

Obj\_Index\_Address

fcb $00,$00

fcb $31,$BD

fcb $39,$AB

fcb $33,$9D

…

Vous êtes libres d'utiliser ces tables pour monter une page de RAM en zone cartouche et d'exécuter manuellement ou venir lire des données dans votre code objet.

Il existe des macro (macros.asm) qui permettent de faciliter ces opération, je ne vous les cite pas toutes, mais un exemple : \_MountObject

MACRO

; param 1 : ObjID\_

; manual mount of an object from the resident page 1

lda Obj\_Index\_Page+\1

\_SetCartPageA

ldx Obj\_Index\_Address+2\*\1

ENDM

Jusqu'ici je n'ai pas parlé de slot de mémoire, de sprite ... c'est fait exprès, c'est pour vous montré l'usage le plus simple d'un objet :

- Définir une routine ou des data en dehors de l'espace non commutable du Game Mode

Il existe des macros pour exécuter un code objet depuis le main du Game Mode, et même depuis un autre objet dans une autre page.

L'engine se charge de gérer l'aller retour et les commutations de page !

Vous pouvez ainsi étendre l'espace parfois trop restreint de votre Game Mode en externalisant des routines ou données dans des objets. C'est le cas dans Sonic 2 pour le driver son qui s'exécute sous la forme d'un objet. Instanciation d'objet L'instanciation d'objet permet de pouvoir conserver plusieurs états d'un même objet.

On va donc attribuer à chaque instance un espace mémoire pour y stocker ses données. Dans le moteur de jeu, ce mécanisme a été mis en place pour stocker les données d'état d'un sprite et stocker des données utilisateur.

Si vous utilisez ce mécanisme d'instanciation, l'objet sera automatiquement exécuté par RunObjects. Si vous ne voulez pas qu'il soit exécuté par RunObjects, vous pouvez définir un espace mémoire dédié et lancer manuellement l'objet avec la macro suivante :

\_RunObject ObjID\_Player1,#player1

Vous aurez alors défini dans le Game Mode un espace de stockage :

player1 fill 0,<taille mémoire>

Pour instancier un objet, on utilise jsr LoadObject\_u ou la variante \_x La routine retourne en u ou x l'adresse mémoire allouée. Il est important juste aprèsde positionner l'id dans cette zone. Si vous ne positionnez pas l'id et faites l'appel à un autre LoadObject, il va allouer de nouveau l'espace précédent car c'est l'id qui détermine si le slot est libre (id=0). Le fait d'allouer la mémoire enregistre l'objet dans une liste d'objets à exécuter lors du RunObjects et seulement lors de l'appel à cette routine (les routines d'affichage utilisent une autre liste pour traiter les objets)

Tant que vous ne référencez pas votre objet comme sprite (voir plus bas), les données de l'objet sont libres d'utilisation (vous pouvez réutiliser tous les emplacements habituellement utilisés par x\_pos, y\_pos ...). Pour cela il suffit de définir vos propes offsets (comme déclarés pour les sprites dans constants.asm). L'objet sera donc exécuté comme les autres mais ne sera pas traité par les routines graphiques.

**Association d'images et d'animations**

# **Collision**

Il existe plusieurs objectifs à la mise en place de tests de collision et donc plusieurs stratégies.

Dans le cas des shoot 'em up, toute collision est "fatale", le test doit simplement permettre de savoir s'il y a eu collision, pas de savoir a quelle position. Les jeux de plateformes permettre au contraire des interactions avec le sol ou des objets solides.

## **shoot 'em up**

Dans un shoot 'em up il est important de permettre au joueur d'avoir un retour visuel sur une collision qui s'est produite. Ainsi il est préférable de voir les objets entrer en collision avant de les faire exposer. Etant donné qu'il n'y a pas d'interaction entre les positions des objets (pas de résistance au sol ou aux murs, on ne pousse pas les objets non plus), il est possible de réaliser les tests de collision en dehors des routines objets.

Position des tests de collision dans la boucle principale du programme :

MainLoop

jsr WaitVBL

jsr ReadJoypads

jsr DoCollision ; le test est réalisé après le rendu et avant la prochaine exécution des routines objets

jsr RunObjects

; ...

jsr DrawSprites

bra MainLoop

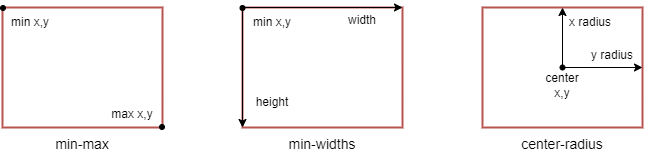
Les routines d'affichage font une conversion des positions dans le référentiel du terrain de jeu (16bits) en positions écran (8bits). Nous allons donc pouvoir utiliser ces calculs déjà réalisés pour effectuer les tests de collision sur 8bits.

## **Axis-Aligned Bounding Box**

Une zone de collision peut être définie de plusieurs manières. Dans ce chapitre nous allons utilisé les AABB (Axis-Aligned Bounding Box), il s'agit d'un rectangle dont les côtés sont alignés sur les axes x et y.



Une AABB peut être représentée de trois manières différentes :



min-max

* x\_min, x\_max, y\_min, y\_max

min-widths

* x, y, width, height

center-radius

* center x, center y, radius x, radius y

Les routines de collision de ce chapitre utilisent la représentation center-radius.

## **Déclaration des zones de collision**

Chaque objet est responsable de déclarer ou modifier lui-même ses zones de collision lorsque cela est nécessaire. L'objet est également chargé du déréférencement des zones lorsque celles-ci ne sont plus utiles.

Pour déclarer une zone de collision l'objet va appeler la routine suivante :

jsr AddAABB\_x

En retour, cette routine va donner dans le registre X, l'adresse d'un slot disponible pour les données de la zone de collision. L'objet va donc utiliser le pointeur X pour paramétrer les valeurs de la zone suivant cette organisation des données (toutes sur 8 bits):

* potentiel
* rayon x
* rayon y
* centre x
* centre y

Exemple :

lda #10

sta AABB.potentiel,x

ldd 10,5

std AABB.radius,x

ldd xy\_pixels,u

std AABB.center,x

Dans cet exemple nous positionnons un potentiel de 10 points à la zone de collision, puis définissons un rectangle de 20 pixels de largeur par 10 pixels de hauteur en paramétrant un radius de 20/2 = 10 et un radius y de 10/2 = 5. Le positionnement de la zone est centrée sur celle de l'objet, par conséquent nous recopions la valeur de positionnement écran de l'objet (xy\_pixel) pour définir la position de la zone. si la zone doit être positionnée de manière décalée par rapport à l'objet, une simple addition peut être ajoutée :

\_ldd xy\_pixels,u

adda #5 ; décale le centre de la zone de collision de 5px vers

std AABB.center,x ; la droite par rapport au centre de l'objet

Enfin, l'objet sauvegardera le pointeur de la zone. Seul l'objet conserve la liste de ses zones associées, cette information n'est pas stockée par le moteur de jeu.

AABB1 equ ext\_variables

; ...

stx AABB1,u

La routine AddAABB contient le code suivant :

## **Suppression des zones de collision**

Pour supprimer une zone de collision l'objet va appeller la routine suivante :

jsr DeleteAABB

Cette routine contient le code suivant :

## **Test de collision**

r = a.rx + b.rx;

if ((unsigned int)(a.cx - b.cx + r) > r + r) return 0;

r = a.ry + b.ry;

if ((unsigned int)(a.cy - b.cy + r) > r + r) return 0;

return 1;

AABB STRUCT

p rmb 1

rx rmb 1

ry rmb 1

cx rmb 1

cy rmb 1

ENDSTRUCT

StructStart

AABB\_a AABB

AABB\_b AABB

org StructStart

fill 0,sizeof{AABB}

fill 0,sizeof{AABB}

DoCollision

ldu #AABB\_a

ldx #AABB\_b

lda AABB.rx,u

adda AABB.rx,x

asla

sta @rx

asra

adda AABB.cx,u

suba AABB.cx,x

cmpa #0

@rx equ \*-1

bhi @rts

lda AABB.ry,u

adda AABB.ry,x

asla

sta @ry

asra

adda AABB.cy,u

suba AABB.cy,x

cmpa #0

@ry equ \*-1

bhi @rts

; compute collision damage

! ldb #0

lda AABB.p,u

suba AABB.p,x

beq @draw

bmi @loose

@win sta AABB.p,u

stb AABB.p,x

@rts rts

@loose lda AABB.p,x

suba AABB.p,u

sta AABB.p,x

stb AABB.p,u

rts

@draw stb AABB.p,u

stb AABB.p,x

rts

# **Les variables globales**

Ce que l’on appelle variable globale est un emplacement mémoire qui peut être accédé en lecture et en écriture depuis n’importe quelle partie d’un programme.

Afin de répondre à des problématiques de performance, on peut ajouter à cette définition qu’il est souhaitable que ces accès n'aient pas recours à des mécanismes de commutation mémoire.

Certaines variables globales doivent, de plus, pouvoir survivre au changement de programmes en mémoire ou au redémarrage logiciel de la machine. On pourra alors y stocker des informations relatives à la configuration du jeu ou à l’état d’une partie.

Un autre type de variable globale dite temporaire permet de mutualiser l’espace mémoire des variables de routine dans le cas où les valeurs de ces variables n’ont pas vocation à persister jusqu'à la prochaine exécution de la routine.

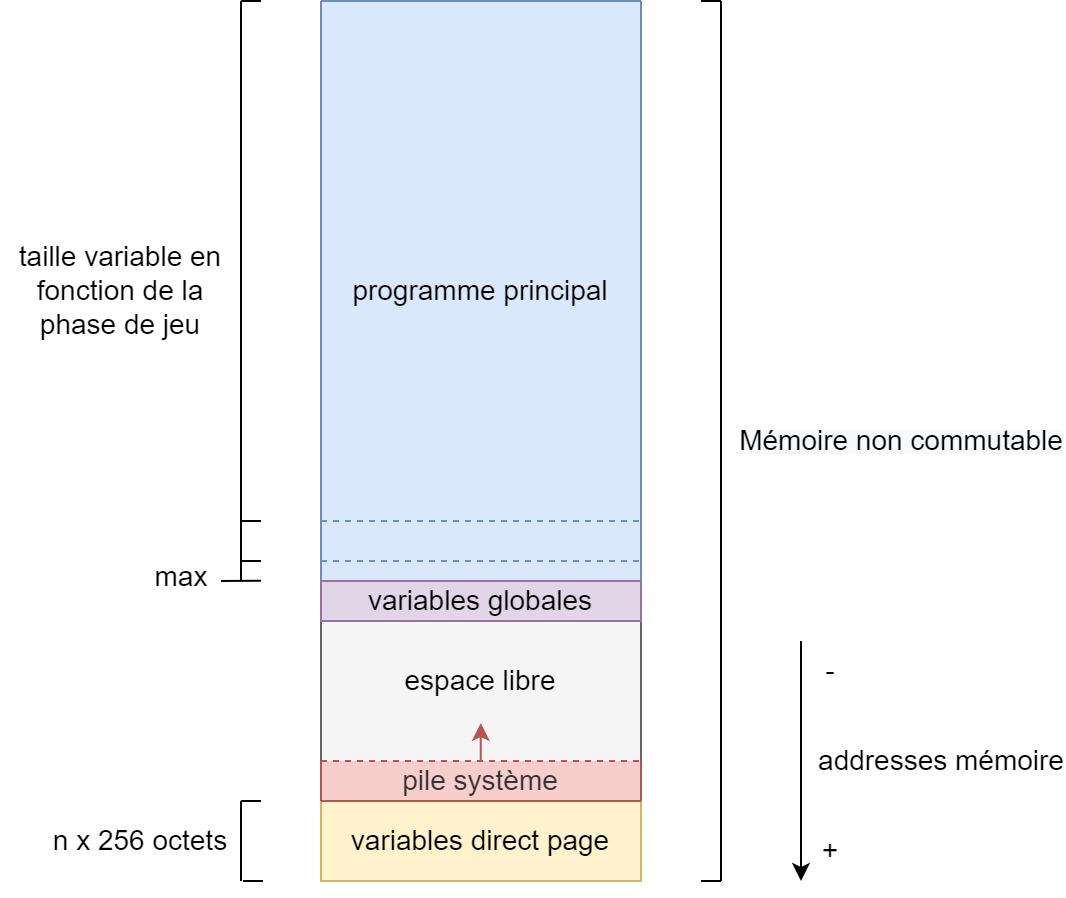
## **Solution**

L’emplacement idéal pour les variables globales va donc dépendre de l'organisation mémoire de la machine. Il s'agit d'identifier une zone mémoire qui reste accessible en permanence, on parle dans ce cas d’une zone mémoire non commutable.

Les systèmes à base de processeur 6809 hébergent habituellement en zone mémoire non commutable :

* le code principal du programme, qui peut évoluer lors de son exécution
* la pile système, qui empile des données ou des adresses lors des appels de routines
* des données accédées par le mode d'adressage direct (utilisation de la *direct page*)

Compte tenu de ces contraintes, voici une organisation mémoire possible :



Il s’agit de positionner les variables globales a la fin du plus gros programme principal qui puisse être chargé en mémoire non commutable, tout en laissant assez de place à la pile dont l’écriture des données est remontante (adresse d’écriture décroissante).

La zone de données accédée au travers du mode d'adressage direct est positionnée en fin de mémoire car sa taille est fixe (256 octets). Il est cependant possible d’utiliser plusieurs blocs de 256 octet, le registre dp étant paramétrable.

(ici présenter les avantages de l’usage de la dp)

## **Mise en oeuvre**

Dans le cadre du moteur de jeu pour le TO8, la mémoire non commutable peut être utilisée ainsi :

(ici insérer un diagramme représentant la mémoire non commutable + exemple d’adresses)

Les variables globales persistantes sont stockées à partir de $9DFF inclus en remontant. Les variables globales temporaires et persistantes pour le moteur de jeu sont stockées en zone dp $9F00-$9FFF

Dans cet exemple, l'espace pour la pile système est de 256 octets, il peut être réduit ou augmenté en fonction des besoins du programmeur.

Dans le cadre du moteur de jeu, la zone "direct page" ne contient pas uniquement des variables temporaires.

Le processeur 6809 offre un système de "direct page" qui permet un accès rapide à la mémoire dans une plage de 256 octets. Ce système est très utile dans le cas de l'utilisation de variables globales (temporaires ou non), il permet de pallier au manque de registres processeur et donne un accès plus rapide aux variables couramment utilisées.

| Name | Octet | Définition |
| --- | --- | --- |
| dp\_engine | 2 | engine routines tmp var space |
| dp\_extreg | 2 | extra register space (user and engine common |
| dp |  | user space (149 bytes max) |
| glb\_system\_stack |  | compilated sprite |
| glb\_register\_s |  | reverved space to store S from ROM routines |
| glb\_screen\_location\_1 |  | start address for rendering of current sprite Part1 |
| glb\_screen\_location\_2 |  | start address for rendering of current sprite Part2 |
| glb\_camera\_height |  |  |
| glb\_camera\_width |  |  |
| glb\_camera\_x\_pos\_coarse |  |  |
| glb\_camera\_x\_pos |  | camera x position in palyfield |
| glb\_camera\_x\_min\_pos |  |  |
| glb\_camera\_y\_pos |  | camera y position in palyfield coordinates |
| glb\_camera\_y\_min\_pos |  |  |
| glb\_camera\_x\_max\_pos |  |  |
| glb\_camera\_y\_max\_pos |  |  |
| glb\_camera\_x\_offset |  |  |
| glb\_camera\_y\_offset |  |  |
| glb\_force\_sprite\_refresh |  |  |
| glb\_camera\_move |  |  |
| glb\_alphaTiles |  |  |
| glb\_timer\_second |  |  |
| glb\_timer\_minute |  |  |
| glb\_timer |  |  |
| glb\_timer\_frame |  |  |
| glb\_Page |  |  |

Tu peux utiliser la fin de la page residente mais avant $9E00. En $9DFE-9DFF par exemple. Il y a un « trou » entre la fin de chacun de tes game mode et :

* la zone dp $9F00-$9FFF
* la pile qui remonte depuis $9EFF inclus Si tu laisse 256 octets à la pile ce qui est "très large", tu peux donc te placer en dessous Par contre tu dois init ta variable car elle n’est pas set a zero. A toi de vérifier dans les .lst de tes GameMode (répertoire generated code) jusqu’où va ton utilisation de cette page

## **Cas d'erreur**

Etant donné qu'il n'y a pas de contrôle de la taille des données écrites lors du chargement du programme principal, il est recommandé de positionner une alerte dans le code qui indiquera si un recouvrement mémoire intervient sur les variables globales.

Dans l'exemple ci dessous on considère que le code principal doit s'arrêter en $9DFD, pour ne pas écraser les variables globales positionnées en $9DFE et $9DFF :

\_end

ifge \_end-$9DFE

error "Main overflow (>=$9DFE)"

endc

Ce code sera interprété durant la phase d'assemblage ou de lien par LWASM et indiquera une erreur si le dépassement se produit.

**La Direct Page pour les données du player one**

Donc je reposte ici l'histoire de l'accès global à n'importe quel objet créé manuellement :

Dans le Player 1 asm :

jsr LoadObject\_u

lda #ObjID\_Player1

sta id,u

stu player1

;...

player1 fdb 0

Dans le code objet :

ldx player1

ldd x\_pos,x

Maintenant la version équivalente, avec le player1 dans DP :

Dans le main.asm

lda #ObjID\_Player1

sta player1+id

;...

\_RunObject ObjID\_Player1,#player1

jsr RunObjects

on aura préalablement mis ce petit alias sympas dans le ram\_data.asm :

player1 equ dp

mais on peut garder dp si on veut ...

dans le code les appels avec x\_pos,u vont fonctionner dans le code du player, mais pour bénéficier de l'accélération du direct page on va faire :

ldd player1+x\_pos

ces instructions sont utilisables de partout dans le code y compris les autres objets

# **Les outils**

## Leanscroll

Parse an image or a tilemap and build a lean tilset and map. When a scroll is made, some pixels on screen do not change, this tool remove any repetitive pixels in a tilemap, taking in account some scroll parameters. It produces an output tileset with the dedicated map.

This tool also provides a way to build an image from a tilemap and vice versa without any transformation.

### Features

* reads a map as:
  + a png image
  + a csv map and a tileset
  + a bin map and a tileset
* applies a lean scroll process (optional) and produce a common image that hold deleted pixels (used for screen initialization before scroll start)
* saves the rendered map, with or without the lean process (optional)
* produces a binary map and a tileset with tiles of an choosen size (optional)
* produces a shifted version of the map and tileset (optional)
* outputs tileset properties to log : nb of tiles, tileset width, tileset height

### Installation

Il convient tout d’abord de cloner la repo <https://github.com/wide-dot/6809-game-builder>. Cette repo est pour l’instant le game engine v2 mais seul les outils sont pour l’instant utilisables.

Dans le répertoire 6809-game-builder, compiler le projet par :

mvn clean package

Tout doit apparaître en “SUCCESS” a la fin de la compilation.

Prenez ensuite le fichier gamebuilder-package-0.0.1-jar-with-dependencies.jar et copiez le dans un répertoire qui contiendra les outils. Une fois copié, double-cliquez sur le fichier .jar, ce qui créera un répertoire 6809-game-builder-tools. Si rien ne se passe, vous pouvez extraire les outils par la commande suivante :

java -jar gamebuilder-package-0.0.1-jar-with-dependencies.jar

Les outils se trouve dans le répertoire 6809-game-builder-tools/bin

### Usage

leanscroll

; input as an image

-image=<file> ; png file (indexed color: 0 transp. 1-16 colors)

; or input as a tileset

-tileset=<file> ; png file (indexed color: 0 transp. 1-16 colors)

-tilesetwidth=<int> ; number of tiles in a tilset row

-tilewidth=<int> ; tile width in pixel

-tileheight=<int> ; tile height in pixel

; tilemap as csv

-csv=<file> ; csv file (semicolon separator, tile index 0-n)

; or tilemap as binary

-tilemap=<file> ; binary file (tile index)

-mapwidth=<int> ; number of tiles in a map row

-mapbitdepth=<int> ; nb of bits for a tile index

[-bigendian] ; (optional) big endian byte order (default little endian)

; output image

[-outimage=<file>] ; (optional) full map image in png

; output tilesets and tilemaps

[-outtileset=<file> ; (optional) output tileset in png

-outtilemap=<file>] ; (optional) output map in binary

[-outtileset1=<file> ; (optional) output tileset in png, X(1) shifted

-outtilemap1=<file>] ; (optional) output map in binary for X(1) shifted tiles

[-outtilewidth=<int>] ; (optional) ouput tile width (default input tile width)

[-outtileheight=<int>] ; (optional) ouput tile height (default input tile height)

[-outmaxsize=<int>] ; (optional) split map files over n bytes (default no split)

; lean processing

[-scrollstep=<int,int,int,int,int,int,int,int> ; (optional) scroll step in pixel for : up, down, left, right, upleft, upright, downleft, downright directions

-nbsteps=<int,int,int,int,int,int,int,int>] ; (optional) number of maximum scroll steps in one frame (used for variable scroll speed) for each directions

[-multidir] ; multidirectional scroll (needs scrollstep for up, down , left and right directions)

[-interlace=<int>] ; (optional) erase even (0) or odd (1) lines

[-leanCsize=<int,int,int,int>] ; common image crop parameters : x, y, width, height

[-lean=<file>] ; (optional) full map image lean in png

[-leanC=<file>] ; (optional) full map image lean common in png

[-leanS=<file>] ; (optional) full map image lean shifted in png

[-leanCS=<file>] ; (optional) full map image lean common shifted in png

### En pratique

(Dans les exemples suivant, il convient de mettre a jour tous les chemins et noms des fichiers et des repertoires)

Pour generer un leanscroll a partir d’un fichier CSV et d’une image PNG contenant les tiles, utilisez par exemple la commande suivante :

../../../6809-game-builder-tools/bin/leanscroll -csv=./objects/levels/01/map/map.csv -outimage=./objects/levels/01/map/out.png -tileset=./objects/levels/01/map/tileset.png -tilewidth=28 -tileheight=14 -tilesetwidth=1 -outtileset=./objects/levels/01/map/0/0.png -outtilemap=./objects/levels/01/map/0/0.bin -outtileset1=./objects/levels/01/map/1/1.png -outtilemap1=./objects/levels/01/map/1/1.bin -scrollstep=0,0,2,0,0,0,0,0 -nbsteps=0,0,1,0,0,0,0,0 -lean=./objects/levels/01/map/0/fullLean.png -leanC=./objects/levels/01/map/init.png -leanS=./objects/levels/01/map/1/fullLean.png -leanCsize=0,0,140,168 -outtilewidth=14 -outtileheight=14

Pour générer un leanscroll a partir d’un fichier PNG contenant tout le map, utilisez par exemple la commande suivante (in.png est le fichier de la map dans cet exemple) :

../../../6809-game-builder-tools/bin/leanscroll -image=./objects/levels/01/map/in.png -outtileset=./objects/levels/01/map/0/0.png -outtilemap=./objects/levels/01/map/0/0.bin -outtileset1=./objects/levels/01/map/1/1.png -outtilemap1=./objects/levels/01/map/1/1.bin -scrollstep=0,0,2,0,0,0,0,0 -nbsteps=0,0,1,0,0,0,0,0 -lean=./objects/levels/01/map/0/fullLean.png -leanC=./objects/levels/01/map/init.png -leanS=./objects/levels/01/map/1/fullLean.png -leanCsize=0,0,140,168 -outtilewidth=14 -outtileheight=14

A la fin de la compilation, des données importantes sont affichées, et qu’il convient d’utiliser et mettre a jour certains fichiers :

18:10:12.792 | INFO | MainCommand > Lean scroll

18:10:12.799 | INFO | Png > Read /<mypath>/thomson-to8-game-engine/game-projects/r-type/./objects/levels/01/map/in.png file ...

18:10:12.939 | INFO | ImageMap > properties for: 0.png 162,1,162

18:10:12.982 | INFO | ImageMap > properties for: 1.png 210,1,210

Les valeurs 161,1,162 et 210,1,210 doivent etre reportées dans le fichier obj.properties dans le repertoire objet/levels/xx/

# Tiles

tileset.Tls\_lvl01=./objects/levels/01/map/0/0.png;**162,1,162**,TOP\_LEFT,./objects/levels/01/map/0/0.0.bin

tileset.Tls\_lvl01\_s=./objects/levels/01/map/1/1.png;**210,1,210**,TOP\_LEFT,./objects/levels/01/map/1/1.0.bin

De plus, le nombre de tiles de votre niveau doit être reporté dans le fichier obj.asm du même répertoire.

stb scroll\_tile\_width

\_ldd 14,**128** ; !!!!!!!!!!!!!!!!!

sta scroll\_tile\_height

Dans l’exemple ci dessus, notre projet a 64 tiles de 28x14, mais elles sont sauvées en tiles de 14x14, donc il y a effectivement 128 tiles de largeur.

Pour finir, il convient aussi de mettre a jour la taille de la map dans le fichier obj.asm qui se trouve dans le game-mode/xx/

map\_width equ **1792**

Avant de recompiler votre projet, effacer le répertoire generated-code/LevelInit

### Erreurs potentielles

Lors de la compilation du projet, des erreurs, suite à l'utilisation de leanscroll, peuvent apparaitre.

Cas 1 :

LevelInit tileset: Tls\_lvl01

La hauteur d'image :1890 n'est pas divisible par le nombre de lignes de tiles :162

Vous avez certainement oublié de reporter les valeurs le fichier obj.properties du repertoire objet/levels/xx/

Cas 2 :

LevelInit tileset: Tls\_lvl01

Le fichier png contient des indices de couleur > 16. Rappel: 0 transparent 1-16 couleurs

Cette erreur peut se produire si vous avez plus que 16 couleurs indexées (outre la couleur de transparence) dans votre fichier map PNG ou dans votre fichier tileset PNG. Il convient donc de réduire la palette. Sous Pro Motion, cela se fait par :

