Rapport final :
Projet Prince of Persia

Oriane Crouzet
Fatima Ayed

04 mai 2025

Table des matières

Table des figures

1. Introduction

L'UE PSTL s'inscrit dans le cadre du deuxième semestre du M1 STL, à l'Université Sorbonne. Notre projet consiste à étudier le code source en assembleur 6502 pour Apple II et à réimplanter une version moderne du jeu **Prince of Persia** (1989). Notre encadrant est M. Antoine MINÉ.

Le jeu **Prince of Persia** a la particularité d'avoir son code source mis à disposition sur GitHub, avec les commentaires originaux de l'auteur. Le code du jeu est couplé à une documentation technique rédigée également par l'auteur, le rendant particulièrement documenté. Par ailleurs, il existe beaucoup de documentation sur la plateforme du jeu (l'Apple II), le processeur, le langage assembleur 6502 et le micro-assembleur Merlin.

Cette richesse de documentation offre la possibilité de bien comprendre ce code assembleur, et tout particulièrement la documentation technique de l'auteur, qui est à destination des programmeurs qui auront la tâche de porter le jeu sur d'autres machines. C'est pourquoi **Prince of Persia** est fortement intéressant à traduire dans le cadre de l'UE PSTL.

2. Contexte du projet

Le but du projet était d'étudier dans un premier temps le code source de la version originale pour Apple II du jeu Prince of Persia (1989), premier jeu de la série, écrit en assembleur 6502 par Jordan Mechner. Le projet visait ensuite à réimplanter pour un ordinateur contemporain et dans un langage moderne de notre choix une partie de ce jeu. L'accent a été mis sur la compréhension du fonctionnement du code source original et de l'architecture matérielle (MOS 6502, Apple II) qui le faisait fonctionner. Pour cela, il nous a fallu mettre en correspondance le code source original et la version moderne développée dans le projet.

Nous avons donc décidé de choisir le C pour notre traduction moderne du jeu. Étant un langage bas-niveau, ce langage nous a permis de simuler au mieux la structure du code assembleur du jeu original. La programmation impérative et procédurale offerte par le langage C est particulièrement adaptée à ce genre de traduction.

Pour dessiner les images à l'écran, l'utilisation de la bibliothèque graphique SDL 2 semblait être un choix cohérent et pertinent. En effet, c'est une bibliothèque graphique bas niveau, qui permet de facilement dessiner une image à afficher pixel par pixel. Or, les routines du jeu dessinent les images à l'écran pixel par pixel (nous le détaillerons ultérieurement). L'utilisation de SDL 2 permet donc de simuler l'affichage graphique de l'Apple II très fidèlement.

3. Cahier des charges

Le projet est donc décomposé en deux grandes parties : la compréhension et l'analyse du code source original, et le code traduit en C.

Afin de recenser notre parcours lors du projet et afin de permettre aux prochains étudiants qui travailleront potentiellement sur ce projet, nous avons décidé d'apporter au rapport final de la documentation supplémentaire (dans les **Ressources**) afin de rendre l'analyse du code plus rapide et accessible. Ces documents seront essentiels pour toute personne souhaitant se plonger en profondeur dans la traduction fidèle de ce jeu en langage moderne. En effet, il est important de noter que ce projet ne consiste pas simplement à réimplémenter le jeu dans un langage moderne en optimisant tout, le but principal est de rester le plus fidèle possible au code source : en utilisant les mêmes noms de variables/fonctions/fichiers, en respectant le squelette du code source, en essayant de simuler au mieux la mémoire de l'Apple II... Ainsi, le produit final sera donc une traduction fidèle du code assembleur vers le C, et non une amélioration des performances du jeu.

Le projet étant conséquent et difficile à opérer, nous avons décidé pour l'implémentation de commencer par l'extraction des ressources d'images afin d'afficher les niveaux du jeu. En effet, cette partie semble essentielle pour démarrer sur une base solide et concrète. A contrario, le gameplay du jeu ne sera traité qu'en finalité,si nous avons le temps. Nous nous sommes donc concentrées sur l'affichage des niveaux et leur disposition sur l'écran et en mémoire avant de passer à la suite.

4. Tâches réalisées

Etape 1

Notre toute première tâche lors de ce projet a bien évidemment été de lire le code source du projet, de l'analyser et d'en tirer les informations principales pour savoir à quoi correspondent chaque fichier du code. Une autre grande partie du travail était de lire les documentations officielles du jeu, de se renseigner sur la structure de L'Apple II et de se familiariser un peu plus avec l'assembleur 6502.

Etape 2

Comme point de départ, nous nous sommes naturellement dirigées vers l'extraction des images et l'affichage des niveaux. Les images dans le code assembleur sont stockées en hexadécimal dans différentes banques. Pour ce faire, nous avons consulté la documentation du code de Prince of Persia écrit par le créateur lui-même. Nous savons ainsi qu'il y a deux tables d'images qui composent les différents

arrière-plans (palace et dungeon) du jeu ainsi que 7 tables d'images qui représentent tous les personnages et les objets du jeu.
Nous avons accès aux fichiers binaires de chaque table, il est structuré comme ceci (*cf.* ***Figure 1***) :

Le premier octet du fichier correspond au nombre d'images+1 contenu dans le fichier. Ensuite, on trouve l'adresse à laquelle l'image est chargée au niveau de la position 2i et 2i+1 (avec i le numéro de l'image). Les deux adresses qui se trouvent à ses deux positions doivent être lu comme un entier avec la position 2i+1 comme les bit de poids fort.

Ensuite, afin de savoir la position de l'image dans le fichier, il faut soustraire son adresse par l'adresse à laquelle le fichier est chargé dans l'Apple II, car l'adresse de l'image est l'adresse après chargement.

On peut enfin lire les 2 premiers octets au niveau de la position de l'image et le premier correspond au nombre d'octets par ligne (donc la largeur de l'image) et le deuxième octet correspond au nombre de lignes (longueur de l'image). Directement à la suite de ces deux octets, on retrouve l'encodage réel de l'image.

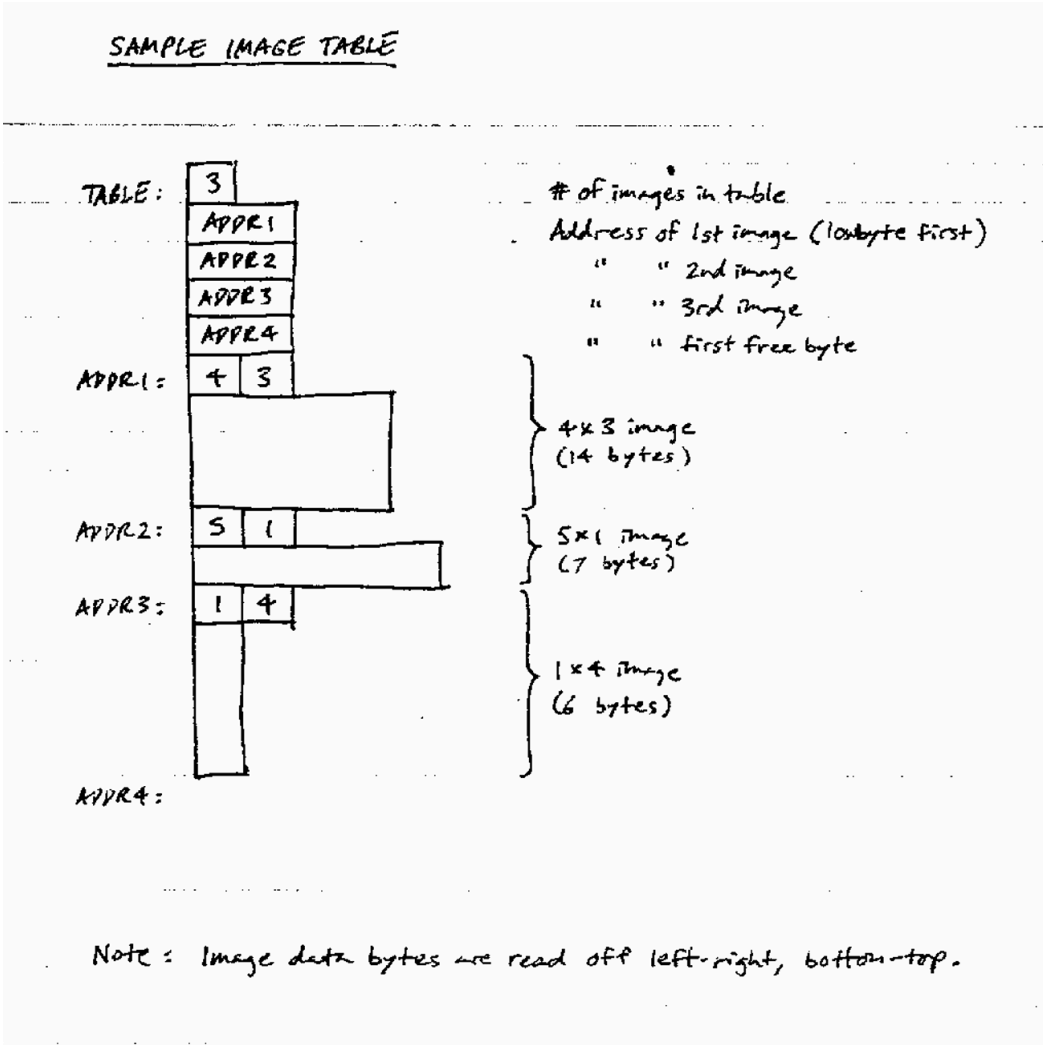

Figure 1 : Exemple pour l'utilisation des tables d'images (source dans ***Ressources****)*

Un pixel correspond à un bit allumé (1). Il faut représenter 7 bit pour chaque octet. En effet, il existe une correspondance avec l'encodage des pixels dans la mémoire de l'écran de l'Apple II : celle-ci utilise le 8ème bit comme une sorte de palette, qui influence la couleur de tous les pixels de la "cellule" de 7 pixels horizontaux encodée par l'octet (*cf.* ***Figure 2***).

Apple II Hi-Res colors and YIQ values[1][2][3]

High bit	Pixel pair	Number	Name	YIQ		
				Y	I	Q
0	00	0	black	0.0	0.0	0.0
0	01	1	purple	0.5	1.0	1.0
0	10	2	green	0.5	−1.0	−1.0
0	11	3	white	1.0	0.0	0.0
1	00	4	black	0.0	0.0	0.0
1	01	5	blue	0.5	1.0	−1.0
1	10	6	orange	0.5	−1.0	1.0
1	11	7	white	1.0	0.0	0.0

Figure 2 : Couleurs proposées par l'Apple II selon les paires de pixels (source dans ***Ressources****)*

Nous l'avons d'abord fait en code ASCII puis nous avons utilisé SDL 2. Avant de réussir à comprendre tout cet encodage, nous avons d'abord exploré plusieurs pistes qui se sont révélées infructueuses et ont entraîné une perte de temps :

- Au début du projet, nous avons d'abord téléchargé et testé énormément de logiciels d'extraction d'images en pensant que cela pourrait être utile. Bien évidemment, le format est trop vieux et aucun extracteur de fichier ne s'est montré utile.

- Nous avons ensuite décidé d'effectuer quelques recherches pour voir si une solution d'extraction des images du jeu n'a pas déjà été proposée par d'autres personnes et nous sommes tombées sur le "*Princed project*" qui réunit une communauté de fans qui ont déjà effectué des réimplantations de différentes versions du jeu. Ils ont notamment développé un outil qui permet d'extraire les images et de les modifier.
  Nous n'avons pas réussi à utiliser cet outil, car il ne fonctionnait pas avec nos fichiers, mais avec des fichiers .PAT.

- Nous avons ensuite pu nous lancer dans l'extraction des images à partir des fichiers binaires grâce à notre encadrant qui nous a expliqué comment est encodé le fichier. La première extraction a donné de bons résultats pour la première bgtable seulement (table des background) pour la raison suivante :

Dans le fichier *GAMEEQ.S* se trouvent toutes les adresses où sont chargées les tables d'images, adresses dont on a besoin pour calculer la position de l'image dans le fichier. Or en utilisant ces adresses, les images extraites n'étaient interprétables que pour *bgtable1* qui est à l'adresse 6000. Il s'avérait que pour trouver la position de n'importe quelle image de tous les fichiers, il fallait utiliser l'adresse 6000 à chaque fois au lieu des adresses définies dans le code source ce qui nous a montré que le code source pouvait contenir des données inexploitables.

- Nous avons enfin réussi à extraire les images, mais elles étaient à l'envers, la documentation indiquait que les octets devaient être lus de gauche à droite et de bas en haut, nous avons donc changé notre code en conséquence.

Vous pouvez retrouver la banque d'images sur notre GitHub, spécifié dans les **Ressources**, dans le dossier "images". En voici un exemple :

→ Prince en position debout (provient de **IMG.CHTAB3**)

Maintenant, que nous disposons de notre banque d'images, nous pouvons nous intéresser au chargement des niveaux, c'est-à-dire déterminer les différents plans qui les composent.

Chaque level est composé de 24 "screens" sur l'écran qui est découpé en 30 blocs (10 blocs pour la largeur et 3 blocs pour la hauteur). Chaque bloc correspond à une des images extraites de la banque d'images, comme les pics, flammes, le sol…

Néanmoins, dessiner un bloc est plus technique. Un bloc possède plusieurs plans : A, B, C et D (*cf.* ***Figure 3***). Afin d'obtenir le bon bloc, il faut commencer par dessiner le plan C ensuite B puis D et enfin A. Afin de pouvoir dessiner les screens, il faut bien comprendre l'encodage des fichiers *Levels*.

Les fichiers de *Levels* sont d'abord encodés par les données *Bluetype* sur 720 octets découpés en 30 octets pour chaque screens (24*30=720). Chaque bit correspond à un bloc du screen.

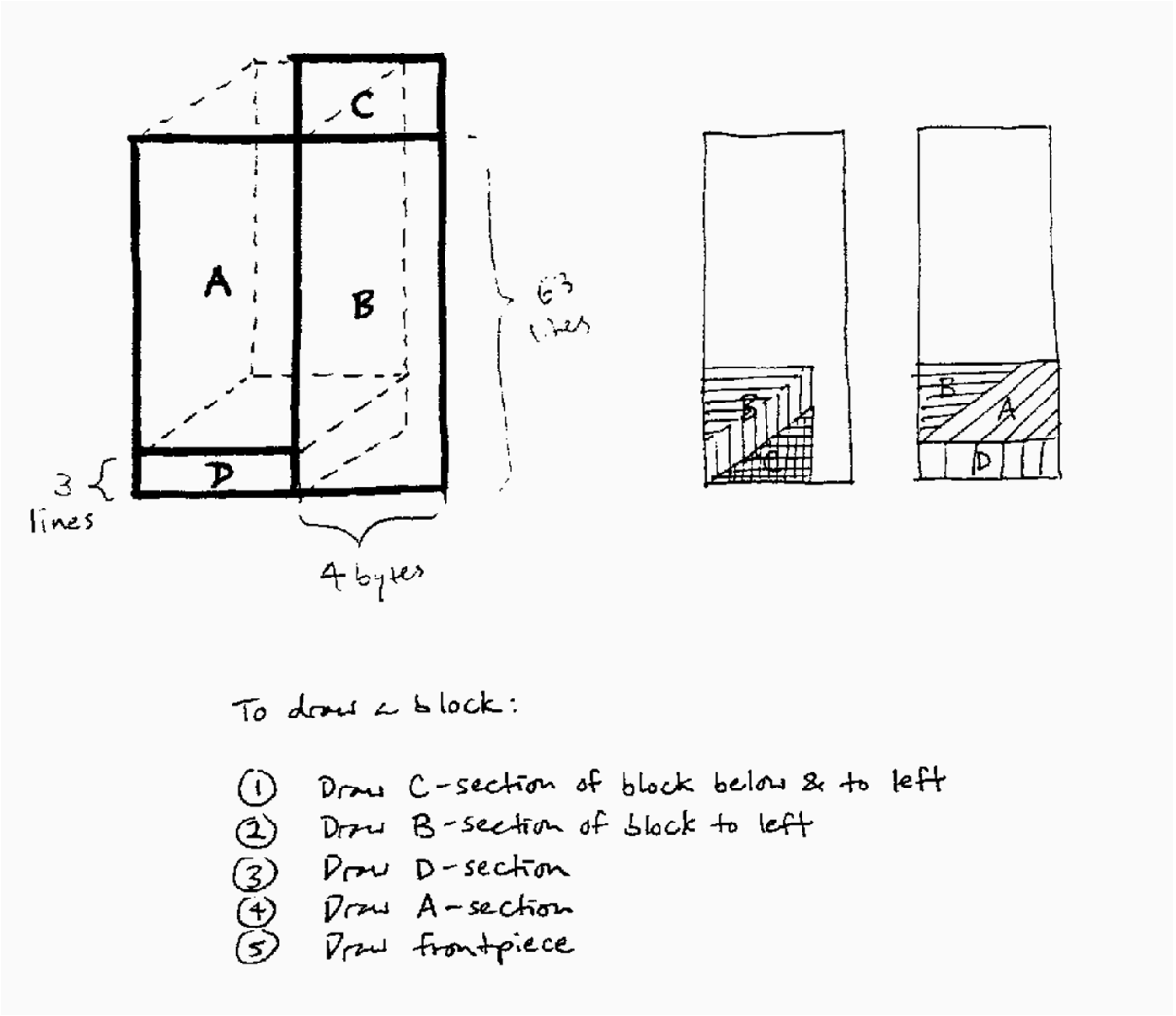

*Figure 3 : Protocole pour dessiner un bloc (source dans **Ressources**)*

Ensuite, sur 720 octets, il y a le *BlueSpec*, chaque octet de cette partie forme une paire avec chaque octet de *BlueType*. La signification de l'octet de *BlueSpec* change en fonction de l'objet désigné par l'octet de *BlueType* qui contient l'identifiant de l'objet. La liste des identifiants des images est donnée dans la documentation. Par exemple, si l'objet peut bouger comme le portail du château, *BlueSpec* indiquera l'état de l'objet (ouvert, fermé…).

Pour avoir le numéro de l'image,nous effectuons aussi un masque en faisant l'opération logique "and 1F" afin d'ignorer le bit de poids fort qui nous indique dans quelle *bgtable* elle se trouve.

Pour les objets statiques comme le sol, il indiquera le design choisi. Enfin, pour les plaques de pression, il indique quels portails elles contrôlent.

Ensuite, on retrouve la "*link list*" sur 256 octets pour *LinkLok* et de même pour *LinkMap*. Ces deux listes donnent plusieurs informations sur les plaques de pressions et les portails (position, correspondance…).

Nous retrouvons après cela *Map,* sur 96 octets, qui indique comment les différents screens sont connectés entre eux. Puisque le personnage peut avancer de l'arrière à l'avant et peut monter et descendre, les screens sont agencés en conséquence. Chaque screen correspond à 4 octets dans *Map* et le premier correspond au screen à sa gauche, ensuite le screen à sa droite puis celui au-dessus et celui en dessous.

Le fichier se termine sur des données d'information sur les positions de début du joueur et des personnages. La documentation est bien plus détaillée sur l'encodage des niveaux que sur celui des images. En effet, l'encodage des niveaux est très spécifique au jeu et non dicté par des contraintes matérielles (à part la taille limitée de la RAM).

Afin d'afficher les niveaux, nous devons maintenant nous référer au code et traduire les fonctions d'affichage en langage C. Pour nous faciliter la tâche, nous avons décidé de traduire ligne par ligne, ce qui nous permet de mieux comprendre la structure du code en assembleur qui n'est pas facile à interpréter.

Etape 3

Pour dessiner les images sur l'écran, 3 fichiers source entrent en jeu :

- *FRAMEADV.S* s'occupe principalement de décider quelles sont les images à charger et à afficher à l'écran. Il **rajoute** des images dans la liste des choses à afficher.
- *HIRES.S* contient les vraies routines d'affichage des images à l'écran.
- *GRAFIX.S* permet de faire l'intermédiaire entre HIRES.S et FRAMEADV.S. Il **gère** les listes d'images à afficher.

Les routines dans *HIRES.S* sont inhérentes à la mémoire, à des adresses et à des sous-routines spécifiques de l'Apple II. C'est donc dans ces méthodes-ci que nous allons utiliser SDL 2. Les routines de dessin de *HIRES.S* qui utiliseront SDL 2 permettront ainsi de garder *FRAMEADV.S* et *GRAFIX.S* fidèles, en termes d'implémentation, à leur équivalent en assembleur. Un travail conséquent de lecture et de traduction de *FRAMEADV.S* et de *GRAFIX.S* a été effectué pour comprendre comment le code gère les images à afficher par niveaux.

Comme évoquée précédemment, la traduction du code s'effectue ligne à ligne, pour rester fidèle à la structure initiale. Le fichier *FRAMEADV.S* est traduit dans un fichier *frameadv.c* dans notre code. Les fonctions de notre code gardent les mêmes noms que les routines en assembleur, il en va de même pour les noms de variables, constantes…

Dans le fichier *eq.h* et *bgdata.h*, nous avons défini toutes les constantes nécessaires au bon fonctionnement du jeu, telles que les adresses de mémoire, les tailles des différentes listes (images de fond, objets, messages, etc.), et les informations sur les écrans. Ces fichiers utilisent également des variables utilisées dans nombres fichiers sources. Nous notons que *eq.h* et *bgdata.h* sont la traduction des fichiers *EQ.S* et *BGDATA.S* du code original.

Exemple de traduction (mise en parallèle) :

Ligne	EQ.S	Ligne	eq.h
289	*---------- * * Image lists * *---------- maxback = 200 ;x4 maxfore = 100 ;x4 maxwipe = 20 ;x5 maxpeel = 46 ;x4 maxmid = 46 ;x11 maxobj = 20 ;x12 maxmsg = 32 ;x5 dum imlists genCLS ds 1 bgX ds maxback bgY ds maxback bgIMG ds maxback	60	/*---------- * * Image lists * *----------*/ // Définition des tailles maximales des listes d'images #define MAX_BACK 200 // Background images #define MAX_FORE 100 // Foreground images #define MAX_WIPE 20 // Wipe effects #define MAX_PEEL 46 // Peel images #define MAX_MID 46 // Mid-layer images #define MAX_OBJ 20 // Objects #define MAX_MSG 32 // Messages // Génération du flag d'effacement d'écran extern uint8_t genCLS; // Background (plan arrière) extern uint8_t bgX[MAX_BACK]; extern uint8_t bgY[MAX_BACK]; extern uint16_t bgIMG[MAX_BACK];
309	bgOP ds maxback	81	extern uint8_t bgOP[MAX_BACK];

Les variables utilisées dans le code original, comme *bgX*, *bgY*, *bgIMG*, *bgOP* pour les images de fond, ou *objX*, *objY* pour les objets, ont été déclarées de la même façon afin de simuler l'environnement Apple II, mais sur une machine moderne.

Le code original en assembleur utilisait des **jump tables** pour gérer le passage entre différentes fonctions. Ces tables de sauts ont été ignorées dans la version C en appelant directement les fonctions correspondant aux routines nécessaires implémentées dans les fichiers .c.

Dans un premier temps, à mi-semestre,le fichier *FRAMEADV.S* a été entièrement traduit, tandis que *GRAFIX.S* et *HIRES.S* sont encore en cours de traduction. Une fois la traduction terminée, nous pourrons enfin afficher les niveaux dans leur entièreté.
Pour 2267 lignes de code dans *FRAMEADV.S*, on obtient environ 1640 lignes de code dans *frameadv.c* une fois la traduction effectuée. La traduction de l'assembleur 6502 vers le C permet de réduire environ de moitié la taille du code.

Exemple de traduction (mise en parallèle) :

Ligne	FRAMEADV.S	Ligne	frameadv.c
1450	*------------------------------ * Draw spikes A *------------------------------ drawspikea ldx state bpl :1 ;hibit clear --> frame # ldx #spikeExt ;hibit set --> spikes extended :1 lda spikea,x beq]rts sta IMAGE lda blockxco sta XCO lda Ay sec sbc #1 sta YCO lda #ora sta OPACITY	914 928	/*------------------------------ * Draw spikes A *------------------------------*/ void drawspikea() { uint8_t x = (state & 0x80) ? SPIKEEXT : state; IMAGE = spikea[x]; if (IMAGE == 0) return; // Pas d'image, on quitte XCO = blockxco; YCO = Ay - 1; // Décalage de -1 pixel en Y OPACITY = ORA; add(); }
1468	jmp add		

Etape 4

Après avoir rendu notre rapport de mi-semestre, nous avons opté pour un changement de stratégie. En effet, il est rapidement devenu impossible de traduire les fonctions sans considérer leur contexte d'appel. La principale difficulté de ce projet a été d'identifier un point d'entrée permettant de traduire le code, plus précisément la routine responsable de l'appel à toutes les fonctions nécessaires à l'affichage d'un niveau.

La recherche de cette routine s'est avérée cruciale, car tout le code en assembleur 6502 repose sur l'utilisation intensive des registres (nous y reviendrons plus tard). Plutôt que de simuler ces registres, nous souhaitions adopter une approche de plus haut niveau en exploitant les possibilités offertes par le langage C.

Lorsqu'une fonction est appelée, son implémentation peut dépendre d'une valeur préalablement stockée dans l'accumulateur par la fonction appelante. Il était donc essentiel de reconstituer la hiérarchie complète des appels de routines pour garantir un résultat cohérent.

Nous avons finalement décidé de ne traduire que les routines strictement nécessaires. Grâce à l'aide de notre encadrant, nous nous sommes concentrés sur la traduction de la routine *DOSURE*, responsable de l'affichage des écrans de niveau. Cette routine se trouve

dans le fichier *TOPCTRL.S*, qui contient toutes les routines d'initialisation et de lancement du jeu.

Nous allons maintenant nous pencher plus en détail sur la hiérarchie d'appel des fonctions et les présenter.

5. Présentation des routines et de leur hiérarchie d'appel

Il est important de noter que le langage assembleur 6502 est insensible à la casse. Cette particularité a entraîné de nombreuses confusions lors de notre analyse, car nous rencontrions fréquemment des routines et des variables portant le même nom, mais avec des différences (majuscules/minuscules). Cette variation orthographique, bien que sans effet sur l'exécution du code, a considérablement compliqué notre compréhension de la base logicielle.

TOPCTRL.S

La routine *DOSURE* effectue les opérations suivantes :

- Elle récupère d'abord le numéro d'écran (*SCRNUM*) en cours
- Elle initialise à 0 plusieurs variables critiques (principalement des tableaux) via *ZEROLSTS*
- Elle appelle ensuite la routine *SURE* qui configure les paramètres d'affichage de l'image (nous analyserons son fonctionnement en détail ultérieurement)

Deux autres routines d'initialisation complètent ce processus :

- *ZEROPEELS* : réinitialisation des variables liées aux effets graphiques
- *ZERORED* : remise à zéro des paramètres de rendu

```
*------------------------------
DoSure
 lda VisScrn
 sta SCRNUM

 jsr zerolsts ;zero image lists

 jsr sure ;Assemble image lists

 jsr zeropeels ;Zero peel buffers
 jsr zerored ;and redraw buffers
;(for next DoFast call)

 jmp drawall ;Dump contents of image lists to screen

*------------------------------
```

*Figure 4 : fonction DOSURE (source dans **Ressources**)*

FRAMEADV.S

Le rendu des écrans s'effectue selon une logique de superposition en trois couches distinctes. La première couche correspond au **background** (arrière-plan), qui sert de fond de base. Vient ensuite le **middleground** (plan intermédiaire) contenant les éléments décoratifs, puis le **foreground** (premier plan) pour les éléments situés devant le personnage. Notre analyse se concentrera spécifiquement sur l'implémentation du background.

La construction du **background** repose sur quatre tableaux clés d'une taille de 200 octets chacun. Les tableaux *bgX* et *bgY* stockent respectivement les coordonnées horizontales et verticales des éléments à afficher. Le tableau *bgIMG* contient les identifiants des sprites correspondants, tandis que *bgOP* gère leurs paramètres d'opacité. Pour chaque position i dans ces tableaux, le moteur de jeu va afficher le sprite *bgIMG[i]* aux coordonnées *(bgX[i], bgY[i])* en appliquant l'effet de transparence spécifié par *bgOP[i]*.

La routine *SURE* présente une complexité particulière due à son rôle étendu. Non seulement elle gère l'affichage du screen courant, mais elle assure également la cohérence visuelle avec les écrans adjacents. Pour cela, elle récupère les trois blocs les plus à droite du screen voisin via *GETPREV*, ainsi que la ligne inférieure du screen situé en dessous via *GETBELOW*. Dans notre implémentation actuelle, nous avons simplifié ce mécanisme en considérant que *GETPREV* retourne systématiquement des blocs vides, ce qui nous permet de nous concentrer sur le comportement principal du screen courant.

Le processus de génération débute par la routine *CALCBLUE*, qui localise l'adresse du blueprint correspondant au numéro d'écran *SCRNUM*. La construction du background s'effectue ensuite à travers deux boucles imbriquées : une boucle externe parcourant les trois blocs verticaux, et une boucle interne traitant les dix blocs horizontaux. La routine *GETOBJID* joue un rôle central dans ce processus en déterminant l'identifiant final de l'objet à afficher. Elle analyse les paramètres *bluespec* et *bluetype* du blueprint, puis les compare avec les différentes variantes graphiques disponibles. Par exemple, un portail peut exister sous plusieurs états (fermé, ouvert, entrouvert), et *GETOBJID* sélectionne la version appropriée en fonction du contexte.

Comme mentionné précédemment, le dessin des différentes sections de chaque bloc suit une séquence précise qui correspond à l'ordre d'insertion des images dans les tables du background. Ce processus s'effectue selon la hiérarchie suivante : nous commençons par ajouter l'image nécessaire pour la section C, puis la section B, et ainsi de suite pour chaque bloc du niveau. La routine *RedBlockSure* est responsable de cette organisation et fait appel à plusieurs sous-routines spécifiques dont nous examinerons quelques exemples représentatifs.

Une fois l'image et ses coordonnées ajoutées dans les tables correspondantes (*bgX*, *bgY*, *bgIMG*), le système enregistre deux informations :

- *OBJID* est sauvegardé dans *PRECED*
- *STATE* est stocké dans *SPRECED*

Cette conservation d'état permet d'assurer une cohérence visuelle entre les blocs adjacents. Par exemple, si un bloc représente des pics, cette information influencera le rendu du bloc suivant en fonction de ses propres paramètres tout en maintenant une transition naturelle. Ce mécanisme crée ainsi une continuité logique entre les éléments du niveau.

Pour référence, le code pertinent de la routine *SURE* concernant ces opérations est disponible en **Annexe** de ce document. Cette section inclut notamment les appels à *RedBlockSure* et la gestion des tables d'affichage, offrant une vision concrète de l'implémentation décrite.

```
RedBlockSure
 jsr drawc ;C-section of piece below & to left
 jsr drawmc

 jsr drawb ;B-section of piece to left
 jsr drawmb

 jsr drawd ;D-section
 jsr drawmd

 jsr drawa ;A-section
 jsr drawma

 jmp drawfrnt ;A-section frontpiece
;(Note: This is necessary in case we do a
;layersave before we get to f.g. plane)
```

*Figure 5 : fonction RedBlockSure (source dans **Ressources**)*

La routine *RedBlockSure* hiérarchise le remplissage des tableaux nécessaires au rendu du background. Son exécution suit une séquence précise :

- Traitement de la section C
- Puis de la section B
- Suivi de la section D
- Enfin de la section A

Cet ordonnancement explique pourquoi la section A, traitée en dernier via *drawfront*, apparaît au premier plan. La routine *drawfront* utilise quatre tableaux spécifiques (similaires à ceux du background) pour gérer son rendu.

Chaque section dispose de deux routines spécialisées :

- Une pour les objets statiques (ex : *drawc*)
- Une pour les objets dynamiques (ex : *drawmc* - "movable C")

Leur mécanisme repose sur :

- La comparaison de *l'objID* avec des valeurs prédéfinies (comme *pillartop* dans *checkc*)
- La gestion d'un flag de carry (positionné à 1 si la section C est visible)
- L'appel conditionnel aux fonctions de rendu effectif

La routine *dodrawc* illustre parfaitement ce système :

1. *checkc* vérifie la visibilité de la section

2. Si visible, *dodrawc* remplit les variables temporaires :
    - *YCO* (coordonnée Y)
    - *XCO* (coordonnée X)
    - *IMAGE* (identifiant du sprite)
    - *OPACITY* (niveau de transparence)

3. La fonction *add* (*addbackground*) transfère ces valeurs :
    - Dans la première case libre de *bgX* (pour *XCO*)
    - Puis dans les cases correspondantes des autres tableaux

Le système implémente des mécanismes avancés :

- Application de masques (comme *domaskB* pour cacher la section B voisine)
- Routines spécialisées (*drawspikeb* pour un rendu particulier des pics)
- Gestion fine des superpositions (*cf.* ***Figure 3***)

L'ensemble des routines mentionnées (*checkc*, *dodrawc*, *drawmc*, *domaskB*, etc.) sont visualisables dans l'**Annexe**.

GRAFIX.S et HIRES.S

Après l'exécution complète de la routine *SURE* et le remplissage des tableaux dédiés au background, le flux du programme retourne à *DOSURE*. C'est à cette étape qu'intervient la routine *DRAWALL*, dont la fonction principale consiste à superposer et à afficher les trois couches graphiques constituant l'affichage final : le background en premier lieu, suivi du midground, puis du foreground qui vient en surcouche.

La routine *DRAWBACK*, localisée dans le fichier *GRAFIX.S*, prend en charge spécifiquement le rendu de l'arrière-plan. Son fonctionnement s'articule autour de trois étapes principales. Dans un premier temps, elle identifie l'image à afficher via son

numéro de référence, ce qui permet à setbgimg de sélectionner la table de données graphiques correspondante. Ensuite, elle extrait les paramètres nécessaires depuis les différents tableaux du background (coordonnées X et Y, référence d'image et paramètre d'opacité) à l'indice courant de traitement. Enfin, elle déclenche le processus d'affichage proprement dit en faisant appel à *fastlay*, le tout au sein d'une boucle qui itère jusqu'à épuisement des éléments graphiques à traiter.

La routine *fastlay*, implémentée dans *HIRES.S*, constitue le cœur du système de rendu bas niveau. Spécialement optimisée pour des performances maximales, elle effectue les opérations directes sur le buffer graphique en s'appuyant sur les paramètres préparés par les routines précédentes. Son analyse détaillée, qui fera l'objet d'une section ultérieure spécifique, permettra d'examiner ses mécanismes d'accélération graphique et son interaction avec le matériel.

6. Choix d'implémentation

Nous allons discuter de certains choix d'implémentation afin d'obtenir des routines équivalentes. Pour faciliter la compréhension de nos traductions vers le langage C, nous avons conservé les mêmes noms de routines et de variables.

Choix généraux

Contrairement à l'assembleur qui utilise les registres X, Y et l'accumulateur A pour manipuler les données, notre implémentation en C simplifie cette approche. Plutôt que de simuler ce mécanisme bas niveau, nous avons opté pour une solution plus directe : les variables globales sont codées en dur via des directives #define.

Par exemple, la variable *gate=4* présente dans *BGDATA.S* devient simplement : **#define gate 4.**

Certaines variables essentielles, bien que non initialisées en assembleur, sont réservées en mémoire dès le départ. Dans notre version C, nous les déclarons comme variables globales externes.

Prenons le cas de *XCO* dans *EQ.S* :

- En assembleur : ***XCO* ds 1** (allocation d'un octet)
- En C : nous déclarons **extern int *XCO***; dans *EQ.h*
- Cette variable sera ensuite initialisée lors de sa première utilisation

Cette approche nous permet de conserver la même logique que le code original, d'éviter des simulations inutiles des registres, de garantir l'accessibilité des variables dans tous les fichiers concernés et de maintenir une bonne lisibilité du code.

Le passage à des #define pour les constantes et à des variables externes pour les données modifiables offre un compromis idéal entre fidélité à l'original et pratiques modernes de programmation en C.

Les boucles et tableaux

```
;-------------------------------
DRAWBACK lda bgX ;# of images in list
 beq ]rts

 ldx #1
:loop stx index

 lda bgIMG,x
 sta IMAGE ;coded image #
 jsr setbgimg ;extract TABLE, BANK, IMAGE

 lda bgX,x
 sta XCO
 lda bgY,X
 sta YCO
 lda bgOP,x
 sta OPACITY
 jsr fastlay

 ldx index
 inx
 cpx bgX
 bcc :loop
 beq :loop
]rts rts
```

*Figure 6 : fonction DRAWBACK (source dans **Ressources**)*

La fonction *DrawBack* offre une excellente illustration du mécanisme des boucles. Son fonctionnement commence par une vérification préalable : si le tableau *bgX* est vide, la fonction se termine immédiatement en retournant via l'instruction *rts*. Ce tableau, dimensionné à *maxback* octets (soit 200), constitue le stockage principal des données graphiques.

Le processus suit une séquence bien définie :

- Initialisation avec la valeur 1 dans le registre X (*ldx*), qui sert d'index
- Chargement dans l'accumulateur de la valeur *bgIMG* correspondant à cet index
- Stockage de cette valeur dans la variable *IMAGE*
- Répétition de ce schéma pour les autres données nécessaires

L'appel à *fastlay* s'effectue via *jsr*, créant une suspension temporaire de *DrawBack* jusqu'au retour (*rts*) de la sous-routine. La gestion de la boucle repose sur trois instructions clés :

- Chargement de l'index dans X
- Incrémentation (*inx*)
- Comparaison avec *bgX*

Le branchement conditionnel (*bcc/beq* vers *loop*) maintient l'exécution tant que les conditions sont remplies. Notons que *bgX* seul désigne la capacité totale (*maxback*), tandis que *bgX*,x accède à la valeur spécifique à l'index courant.

Codes auto-modifiants

```
setback lda #addback
 sta ]add+1
 lda #>addback
 sta ]add+2
 rts
```

*Figure 7 : fonction SETBACK (source dans **Ressources**)*

La fonction *setback* (*cf.* ***Figure 7***) va charger l'adresse de la routine *addback* dans la variable *add* en deux fois (octet de poids fort puis faible). *Add* va pointer sur *addback* qui sera appelé à chaque appel de *add*. Cela permet plus de flexibilité, car nous pouvons décider d'utiliser d'autres fonctions comme *addmid*.

Ce mécanisme est facilement implémentable en C avec un pointeur de fonction qui pourra pointer vers la fonction voulue.

Une autre forme de code auto-modifiant consiste à stocker une variable dans un label. Ainsi, lorsqu'on saute sur ce label, le code s'exécutera avec la valeur précédemment stockée, comme dans l'exemple de code ci-dessous :

```
lda OPCODE,x
sta  :smod
```

La fonction charge dynamiquement la xième valeur de *OPCODE* dans le label *smod*. Sur la **Figure 16** (*cf.* ***Annexe***), la fonction effectue une opération *ORA* entre la ième valeur de *BASE* et la valeur de *smod*, puis sauvegarde le résultat dans *BASE*. Cette astuce évite d'utiliser une variable globale.

Dans notre version, nous remplaçons ce mécanisme par une variable locale, plus simple à gérer en C tout en conservant la même logique. Cette opération s'exécute en boucle comme dans le code original de fastlay, avec une implémentation plus adaptée à nos besoins.

Système de Carry

Le bit de carry joue un rôle essentiel dans les opérations arithmétiques (additions, soustractions) et logiques (décalages, comparaisons) en assembleur 6502. Son activation (**sec = set carry**) et sa désactivation (**clc = clear carry**) permettent de gérer les retenues lors d'opérations dépassant 255 (par exemple, un résultat supérieur à 8 bits active le carry à 1). Ce bit conditionne aussi les sauts, comme *bcs* (branch if carry set), permettant de bifurquer vers d'autres parties du code.

Dans notre traduction en C, nous avons simplifié ce mécanisme en utilisant un booléen pour représenter le carry. Cependant, nous ne l'employons que dans des cas spécifiques nécessitant une activation explicite, sans lien avec des calculs arithmétiques (contrairement à l'usage originel en assembleur).

Pour référence, l'implémentation originale en 6502 est disponible dans les **Ressources**.

```
void checkc(){
    if(objid==0||objid==pillartop||
       objid==panelwof||
       objid>=archtop1){//section C visible
        carry=true;
        return;
    }
    else{ carry=false;
    return;
}

}


void drawc(int colno){
    checkc();
    if(carry== false) return;
    dodrawc(colno);
    domaskb();

}
```

```
checkc
 lda objid ;Does this space contain solid floorpiece?
 beq :vis
 cmp #pillartop
 beq :vis
 cmp #panelwof
 beq :vis
 cmp #archtop1
 bcs :vis
 bcc ]rts ;C-section is hidden
:vis sec ;C-section is visible
]rts rts

*------------------------------
*
```

*Figure 8 : comparaison de l'utilisation de carry entre notre implémentation (gauche) et l'implémentation de base (droite) (source dans **Ressources**)*

Fonctionnement de FASTLAY et processus de dessin sur l'écran Apple II

Fastlay est la routine optimisée pour le dessin rapide d'images à l'écran. Contrairement à sa contrepartie plus lente, mais plus robuste (*lay*), *fastlay* ne gère pas les offsets, le clipping ou le mirroring, ce qui lui permet d'atteindre des performances supérieures au détriment de certaines vérifications de sécurité.

Sur Apple II, l'affichage repose sur deux banques mémoire distinctes : la **Main RAM** et la **Aux RAM**.

Le basculement entre ces deux espaces s'effectue via des **soft-switchs** (adresses mémoire spéciales comme $C002). Cette architecture permet des opérations pixel par pixel en conservant une image dans Aux RAM tout en effectuant des calculs logiques avec les images de fond stockées en Main RAM, améliorant ainsi le rendu graphique.

Fastlay utilise les soft-switchs pour appliquer des **masques** entre les images déjà dessinées et les nouvelles images à afficher.

Le type de masque est déterminé par le paramètre *OPACITY* :

- Valeur 2 ou 5 : opération "sta" (stockage direct)
- Valeur 0 ou 4 : opération logique "and"
- Valeur 1 : opération "or"
- Valeur 3 : opération "xor"

Notons que l'opération "*sta*" fait appel à *fastlaySTA*, une version encore plus rapide de *fastlay* que nous n'avons pas implémentée, car elle exclut totalement les opérations de transparence.

Comme mentionné précédemment, les sections des blocs de niveau peuvent se superposer. Ce mécanisme nécessite une gestion précise de la transparence pour assurer la visibilité correcte des différentes couches. Fastlay détermine quels pixels doivent rester transparents afin de révéler les parties appropriées des blocs du-dessous.

La routine réalise les étapes suivantes :

1. Récupération des données d'image :
    - Appel à *setimage* pour obtenir l'adresse de départ des données graphiques (stockée dans *IMAGE*)
    - Lecture de l'opcode et branchement vers le label correspondant

2. Vérifications préalables :
    - Contrôle des dimensions de l'image par rapport aux limites de l'écran
    - Gestion minimale du clipping (ignorer les lignes avec index négatif tout en affichant le reste)

3. Boucles de rendu :

- Utilisation de la variable *BASE* pour stocker la ligne courante en cours de traitement
- Double boucle d'affichage :
    - Boucle externe : parcourt les lignes (hauteur)
    - Boucle interne : parcourt les colonnes (largeur)
- Affichage progressif depuis la fin vers le début de l'image

Le système utilise des techniques avancées pour déterminer la position des pixels :

- Code auto-modifiant : modification dynamique des instructions
- Gestion des pages :
    - La variable *PAGE* stocke l'adresse de la page de destination
    - Stockée au niveau du label **smPage+1** pour permettre des modifications
    - Alternance entre pages permettant :
        - Affichage d'un écran
        - Préparation du suivant
        - Transition fluide lors des déplacements du personnage

Cette architecture permet des transitions instantanées entre les différents écrans d'un niveau, tout en maintenant une excellente performance graphique sur le matériel Apple II.

L'Apple II utilise un système d'adressage vidéo particulier qui nécessite une conversion des coordonnées en adresses mémoire spécifiques. Pour optimiser ce processus, les routines graphiques s'appuient sur deux tables de précalcul nommées *YLO* et *YHI*. Ces tables contiennent les adresses mémoire correspondant à chaque ligne Y de l'écran, permettant de contourner efficacement la disposition non linéaire de la mémoire vidéo qui est organisée en blocs entrelacés.

Dans la routine *Fastlay*, le calcul de l'adresse d'un pixel donné s'effectue en plusieurs étapes. Pour chaque ligne Y de l'image à afficher, le système commence par consulter *YLO[Y]* pour obtenir l'octet bas de l'adresse, puis *YHI[Y]* pour l'octet haut. À cette adresse de base, on ajoute ensuite la coordonnée *X* (stockée dans *XCO*) ainsi que l'offset de page (*PAGE*) pour obtenir la position mémoire exacte où écrire le pixel (*cf.* ***Figure 14***). Cette méthode permet un affichage rapide malgré la complexité du système vidéo de l'Apple II.

La fonction *setbgimg* a été conçue avec une logique conditionnelle. Elle agit comme un routeur, appelant *setbgimg_bis* avec *BGTAB1* si le bit de poids fort est à 0, ou avec *BGTAB2* dans le cas contraire. Cette approche permet de sélectionner dynamiquement la table de données graphiques appropriée. La fonction secondaire *setbgimg_bis* se charge alors de retourner l'adresse de début des données de la table sélectionnée.

Concernant les dimensions d'affichage, notre implémentation tient compte des spécificités techniques de l'Apple II. L'écran compte 192 lignes de résolution verticale. Pour la largeur, nous avons retenu le calcul 39 octets × 7, ce qui donne 273 points et correspond précisément à la largeur utile de l'affichage. Ce choix permet une couverture optimale de l'espace écran tout en respectant les contraintes matérielles.

7. Avancement dans le projet et tâches à réaliser l'année prochaine

Bilan et perspectives du projet

En début de projet, nous envisagions d'implémenter l'ensemble du gameplay, incluant le déplacement des personnages et l'animation des éléments mobiles comme les flammes. Cependant, les contraintes temporelles nous ont limités à la traduction des routines graphiques de base. La principale difficulté a résidé dans la compréhension de la structure complexe du code assembleur, particulièrement la hiérarchie des appels de fonctions et l'identification des routines clés comme celle responsable de l'affichage des niveaux.

Feuille de route pour l'année prochaine

La priorité sera l'implémentation de *DRAWMID* afin d'obtenir des niveaux complets avec personnages et objets. Les étudiants pourront ensuite aborder :

1. Animation des objets mobiles :

- Explorer la routine *MAINLOOP* dans *TOPCTRL.S*
- Traduire en priorité *NEXTFRAME* et *FRAMADV*
- Tirer parti des routines déjà disponibles comme fastlay

2. Contrôle du personnage :

- Implémenter *GENCTRL* (*CTRL.S*) pour gérer les inputs
- Simuler le joystick Apple II

3. Comportements autonomes :

- Traduire les routines d'*AUTO.S* pour les ennemis
- Se concentrer sur un niveau spécifique (collisions, vies, etc.)
- Implémenter *ANIMATTRANS* pour les transitions d'animation

Stratégies recommandées

- Commencer par *ATTRACTLOOP* (écran d'accueil)
- Implémenter les routines d'affichage texte (messages, numéros de niveau)
- Privilégier Linux pour :
    - Une meilleure compatibilité avec SDL2
    - Des recherches plus efficaces via grep (en tenant compte de la casse)

Contexte narratif du jeu

D'après la documentation, le niveau 1 correspond à l'évasion de cellule, tandis que le niveau 2 met en scène l'affrontement avec le gardien. Cette progression pourra guider l'implémentation des mécaniques de jeu spécifiques à chaque niveau.

8. Ressources

*Documentation officielle du code, écrite par le développeur du jeu (**Figures 1, 3**)*
https://www.jordanmechner.com/downloads/popsource.pdf

*Notre github (**Figure 8**)*
https://github.com/OrianeCrouzet/PSTL.git

*Code source (**Figures 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16**)*
jmechner/Prince-of-Persia-Apple-II: A running-jumping-swordfighting game I made on the Apple II from 1985-89

*Documentation pour l'écran de l'Apple II (**Figure 2**)*
Apple II graphics - Wikipedia

9. Annexe

```
*------------------------------
*  Draw spikes B
*------------------------------
drawspikeb
 ldx spreced
 bpl :1 ;hibit clear --> frame #
 ldx #spikeExt ;hibit set --> spikes extended
:1 lda spikeb,x
 beq ]rts
 sta IMAGE
 lda blockxco
 sta XCO
 lda Ay
 sec
 sbc #1
 sta YCO
 lda #ora
 sta OPACITY
 jmp add
```

*Figure 9 : fonction DrawSpikeB (source dans **Ressources**)*

```
*------------------------------
DRAWALL
 jsr DOGEN ;Do general stuff like cls

 lda blackflag ;TEMP
 bne :1 ;

 jsr SNGPEEL ;"Peel off" characters
;(using the peel list we
;set up 2 frames ago)

:1 jsr ZEROPEEL ;Zero just-used peel list

 jsr DRAWWIPE ;Draw wipes

 jsr DRAWBACK ;Draw background plane images

 jsr DRAWMID ;Draw middle plane images
;(& save underlayers to now-clear peel list)

 jsr DRAWFORE ;Draw foreground plane images

 jmp DRAWMSG ;Draw messages
```

*Figure 10 : fonction DRAWALL (source dans **Ressources**)*

```
SURE
 lda #1
 sta genCLS ;clear screen

 jsr setback ;draw on bg plane

 jsr getprev ;get 3 rightmost blocks of screen to left

 lda SCRNUM
 jsr calcblue ;get blueprint base addr

*  Draw 3 rows of 10 blocks (L-R, T-B)

 ldy #2
:row sty rowno ;0 = top row, 2 = bottom row

 lda BlockBot+1,y
 sta Dy ;get Y-coord for bottom of D-section
 sec
 sbc #3
 sta Ay ;& A-section

 lda Mult10,y
 sta yindex ;block # (0-29)

 lda PREV,y
 sta PRECED
 lda sprev,y
 sta spreced ;get objid & state of preceding block

 jsr getbelow ;get 10 topmost blocks of screen below

 lda #0
 sta colno ;0 = leftmost column, 9 = rightmost
:loop asl
 asl
 sta XCO
 sta blockxco ;get X-coord for A-section

 ldy yindex
 jsr getobjid
 sta objid ;get object id# of current block

 jsr RedBlockSure ;Redraw entire block

 lda objid
 sta PRECED
 lda state
 sta spreced ;Move on to next block

 inc yindex
 inc colno

 lda colno
 cmp #10
 bcc :loop ;...until we've done 10 blocks

:nextln ldy rowno
 beq :done
 dey
 jmp :row ;...and 3 rows
```

*Figure 11 : fonction SURE (source dans **Ressources**)*

```
*------------------------------
*
*  Mask B-section of piece to left
*
*------------------------------
domaskb
 ldx PRECED
 lda maskb,x
 beq ]rts
 sta IMAGE

 lda Dy
 sta YCO
 lda #and
 sta OPACITY
 jmp add
```

*Figure 12 : fonction DoMaskB (source dans **Ressources**)*

```
*------------------------------
*
* D R A W  M O V A B L E  ' B '
*
*------------------------------
drawmb
 lda PRECED

 cmp #gate ;check for special cases
 bne :1
 jmp drawgateb ;draw B-section of moving gate

:1 cmp #spikes
 bne :2
 jmp drawspikeb

:2 cmp #loose
 bne :3
 jmp drawlooseb

:3 cmp #torch
 bne :4
 jmp drawtorchb
:4
:5 cmp #exit
 bne :6
 jmp drawexitb

:6
]rts rts
```

*Figure 13 : fonction DrawmB (source dans **Ressources**)*

```
YLO hex 0000000000000000808080808080808080
 hex 0000000000000000808080808080808080
 hex 0000000000000000808080808080808080
 hex 0000000000000000808080808080808080

 hex 2828282828282828A8A8A8A8A8A8A8A8
 hex 2828282828282828A8A8A8A8A8A8A8A8
 hex 2828282828282828A8A8A8A8A8A8A8A8
 hex 2828282828282828A8A8A8A8A8A8A8A8

 hex 5050505050505050D0D0D0D0D0D0D0D0
 hex 5050505050505050D0D0D0D0D0D0D0D0
 hex 5050505050505050D0D0D0D0D0D0D0D0
 hex 5050505050505050D0D0D0D0D0D0D0D0

YHI hex 2024282C3034383C2024282C3034383C
 hex 2125292D3135393D2125292D3135393D
 hex 22262A2E32363A3E22262A2E32363A3E
 hex 23272B2F33373B3F23272B2F33373B3F

 hex 2024282C3034383C2024282C3034383C
 hex 2125292D3135393D2125292D3135393D
 hex 22262A2E32363A3E22262A2E32363A3E
 hex 23272B2F33373B3F23272B2F33373B3F

 hex 2024282C3034383C2024282C3034383C
 hex 2125292D3135393D2125292D3135393D
 hex 22262A2E32363A3E22262A2E32363A3E
 hex 23272B2F33373B3F23272B2F33373B3F
```

*Figure 14 : looktable pour calculer les positions des pixels sur l'écran (source dans **Ressources**)*

```
drawc
 jsr checkc
 bcc ]rts
 jsr dodrawc ;OR C-section of piece below & to left
 jmp domaskb ;Mask B-section of piece to left

*------------------------------
*
*  Return cs if C-section is visible, cc if hidden
*
*------------------------------
checkc
 lda objid ;Does this space contain solid floorpiece?
 beq :vis
 cmp #pillartop
 beq :vis
 cmp #panelwof
 beq :vis
 cmp #archtop1
 bcs :vis
 bcc ]rts ;C-section is hidden
:vis sec ;C-section is visible
]rts rts

*------------------------------
*
*  Draw C-section of piece below & to left
*
*------------------------------
dodrawc
 ldx colno
 lda BELOW,x ;objid of piece below & to left
 tax
 cpx #block
 beq :block
 lda piecec,x
 beq ]rts ;piece has no c-section
 cmp #panelc0
 beq :panel ;special panel handling
:cont sta IMAGE
 lda blockxco
 sta XCO
 lda Dy
 sta YCO
 lda #ora
 sta OPACITY
 jmp add

* Special panel handling

:panel ldx colno
 lda SBELOW,x
 tay
 cpy #numpans ;# of different panels
 bcs ]rts
 lda panelc,y
 bne :cont
 rts

:block ldx colno
 lda SBELOW,x
 tay
 cpy #numblox
 bcc :1
 ldy #0
:1 lda blockc,y
 bne :cont
]rts rts
```

*Figure 15 : fonction DRAWC (source dans **Ressources**)*

```
:outloop
 lda YLO,x
 clc
:smXCO adc #0
 sta BASE

 lda YHI,x
:smPAGE adc #$20
 sta BASE+1

:smSTART ldy #3

:inloop
]ramrd3 sta $c003 ;RAMRD aux

 lda (IMAGE),y

 sta $c002 ;RAMRD main

:smod ora (BASE),y
 sta (BASE),y

 dey
 bpl :inloop

:smWIDTH lda #4
 adc IMAGE ;assume cc
 sta IMAGE
 bcc :2
 inc IMAGE+1
:2
 dex
:smTOP cpx #$ff
 bne :outloop

 rts
```

*Figure 16 : boucle de la fonction FastLay (source dans **Ressources**)*