

# AIRWOLF : AUTOPSIE D'UN JEU MAUDIT

Une plongee dans les entrailles du code d'un classique  
rate de l'Amstrad CPC



Figure 0 : Ecran titre d'Airwolf - Elite Systems, 1985

# AVANT-PROPOS

---

"Qu'y a-t-il au niveau 2 d'Airwolf sur Amstrad CPC ?"

Cette question, posée une nuit d'insomnie, allait déclencher une obsession. Une quête de vérité à travers des milliers de lignes de code assembleur Z80, des registres du Gate Array, et des mystères enfouis dans 64 Ko de mémoire.

Airwolf n'est pas qu'un jeu. C'est un artefact. Un témoin figé de décisions prises sous pression, de compromis techniques, d'ambitions revues à la baisse. C'est l'histoire d'un développement chaotique, lisible entre les lignes de son code source comme on lirait les strates géologiques d'une falaise.

Ce livre est le récit de cette exploration. Une autopsie numérique qui révèle, derrière chaque octet, les intentions, les fatigues et les passions de développeurs anonymes.

---

# PREMIERE PARTIE : LE CONTEXTE

---

## Chapitre 1 : Un jeu, une legende

### La serie televisee

Airwolf, c'est d'abord une serie americaine diffusee entre 1984 et 1987. Un helicoptere de combat furtif, le plus avance au monde, est pilote par Stringfellow Hawke pour le compte d'une agence gouvernementale secrete. Action, suspense, et surtout cet helicoptere noir aux lignes elegantes : Airwolf devient une icone des annees 80.

### L'adaptation videoludique

Comme toute licence populaire de l'epoque, Airwolf est declinee en jeu video. Elite Systems acquiert les droits et developpe des versions pour les micro-ordinateurs dominants : ZX Spectrum, Commodore 64, et Amstrad CPC.

Sur Amstrad, le jeu sort en 1985. Le joueur incarne Stringfellow Hawke aux commandes de son helicoptere dans un reseau de grottes labyrinthique. La mission : sauver des otages, detruire des cibles, et s'echapper avant l'ecoulement du temps imparti.

### La reputation

Rapidement, Airwolf acquiert une reputation particuliere. Non pas celle d'un chef-d'oeuvre, mais celle d'un cauchemar. Les joueurs se heurtent a une difficulte extreme. La plupart n'arrivent pas a depasser quatre ecrans. Ceux qui persistent plus loin rencontrent systematiquement un plantage dans la zone finale.

Une legende urbaine se forme : personne n'a jamais termine Airwolf. Le niveau 2 reste un mythe, une terre promise que nul n'a jamais foulee.

---

## Chapitre 2 : L'enquete commence

### Une nuit d'insomnie

C'est lors d'une periode de sante fragile que l'enquete debute. L'insomnie devenant insupportable, il fallait occuper l'esprit. Repondre a des questions absurdes. Et celle-ci surgit : "A quoi ressemble le niveau 2 d'Airwolf ?"

### Les outils du detective

L'emulateur WinAPE devient le laboratoire. Son debugueur integre permet de visualiser la memoire en temps reel, de poser des breakpoints, de tracer l'execution instruction par instruction. Le code assembleur Z80 se dévoile progressivement.

### La premiere decouverte

Le "crash" de la zone finale n'est pas un bug. C'est une boucle infinie deliberee :

```
INFINITE_LOOP:
    JR INFINITE_LOOP    ; Le jeu boucle eternellement
```

Quelqu'un a volontairement arrete le jeu. Pourquoi ? La reponse se trouve plus profondement dans le code.

---

# DEUXIEME PARTIE : L'ARCHITECTURE

## Chapitre 3 : La cartographie memoire

### Organisation generale

L'Amstrad CPC dispose de 64 Ko de memoire vive. Airwolf les utilise avec une organisation revelatrice :

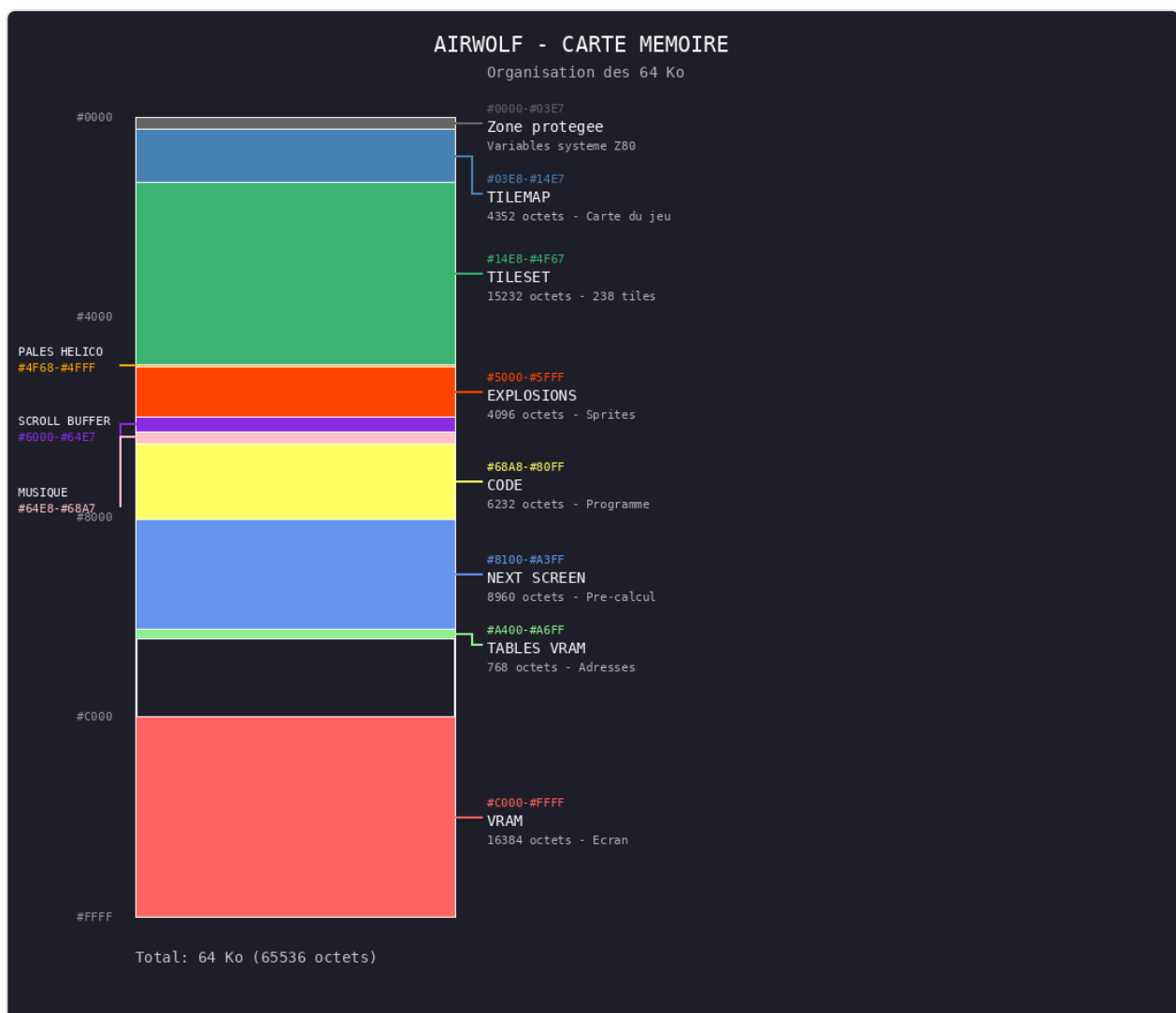


Figure 1 : Organisation complete des 64 Ko de memoire d'Airwolf

Adresse	Taille	Contenu
-----		
#0000-#03E7	1000	Zone protegee (variables systeme)
#03E8-#14E7	4352	TILEMAP - Carte du jeu
#14E8-#4F67	15232	TILESET - Les graphismes des tiles
#4F68-#4FFF	152	Sprites des pales d'helicoptere
#5000-#5FFF	4096	Sprites d'explosion
#6000-#64E7	1256	Buffer de scrolling
#64E8-#68A7	960	Donnees musicales
#68A8-#80FF	6232	CODE executable
#8100-#A3FF	8960	Buffer ecran suivant (pre-calcul)
#A400-#A6FF	768	Tables d'adresses VRAM
#C000-#FFFF	16384	Memoire video

## Le monde en une seule carte

Premiere revelation majeure : il n'y a pas de niveau 2. Toute la tilemap du jeu est chargee d'un bloc au demarrage. Elle occupe 4352 octets a partir de l'adresse #03E8. Le jeu ne fait que naviguer dans cette carte unique.

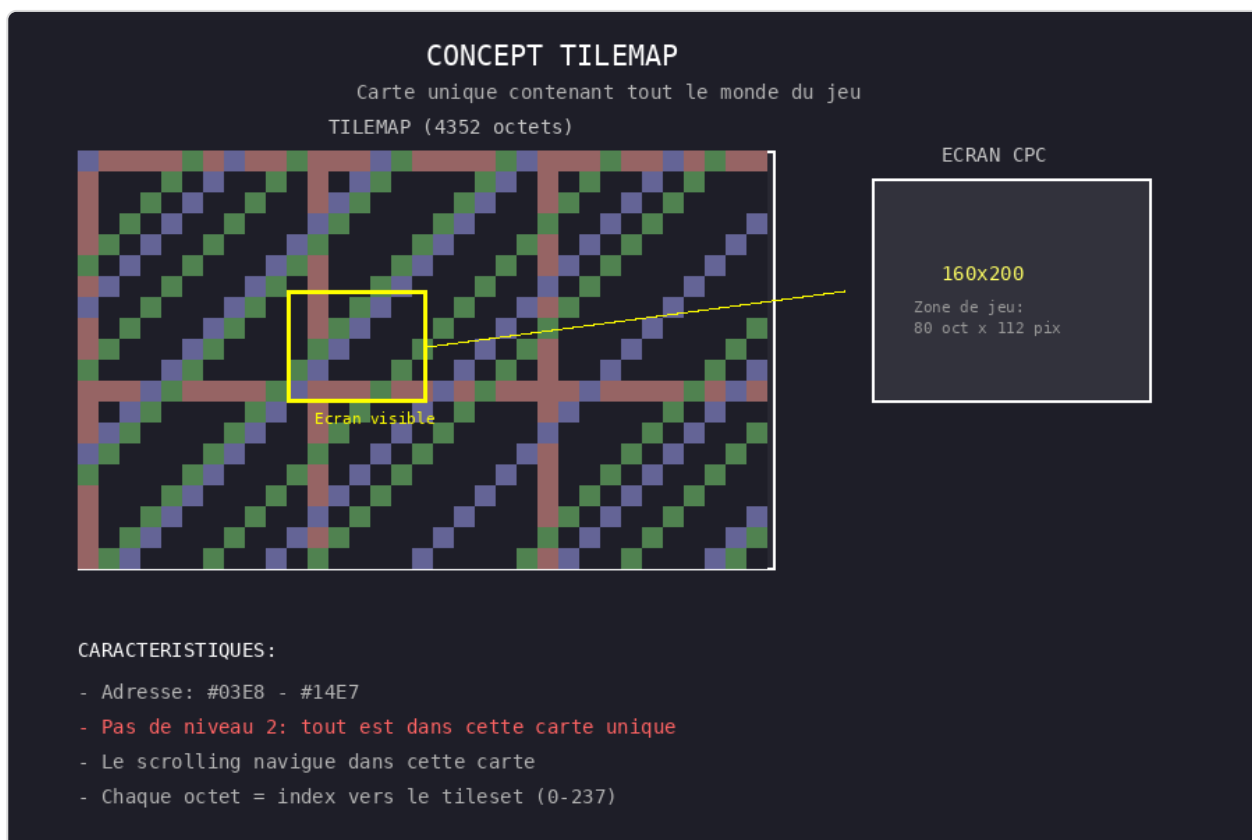


Figure 2 : Le concept de tilemap - une carte unique contenant tout le monde

La tilemap definit l'agencement des tiles (blocs graphiques de 16x16 pixels) qui composent le decor. Chaque octet est un index vers le tileset, la bibliotheque de 238 tiles differents stockee a partir de #14E8.

## Pas de chargement dynamique

Contrairement aux jeux plus ambitieux qui chargent les niveaux depuis la cassette ou la disquette, Airwolf contient tout en memoire. Cela explique la taille relativement modeste du monde et l'absence de niveaux supplementaires.

# Chapitre 4 : Le mode graphique

## Mode 0 : le choix de la couleur

Airwolf utilise le Mode 0 de l'Amstrad CPC :

- Resolution : 160x200 pixels
- 16 couleurs simultanees parmi 27 disponibles
- 1 octet = 2 pixels (4 bits par pixel)

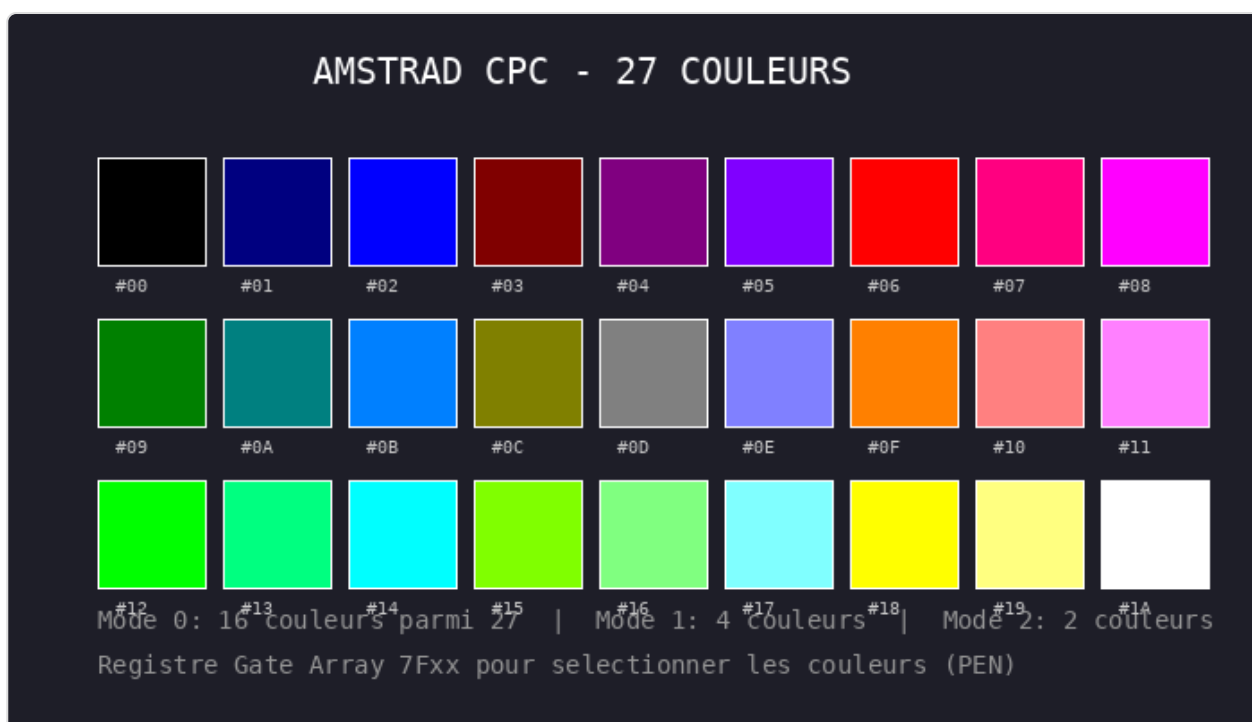


Figure 3 : Les 27 couleurs hardware de l'Amstrad CPC

L'encodage Mode 0 est particulier. Les bits d'un octet ne sont pas contigus :

```
Octet : |b7|b6|b5|b4|b3|b2|b1|b0|
        | | | | | | | |
Pixel 0: b7----b5----b3----b1
Pixel 1: b6----b4----b2----b0
```

## Les tiles : briques du monde

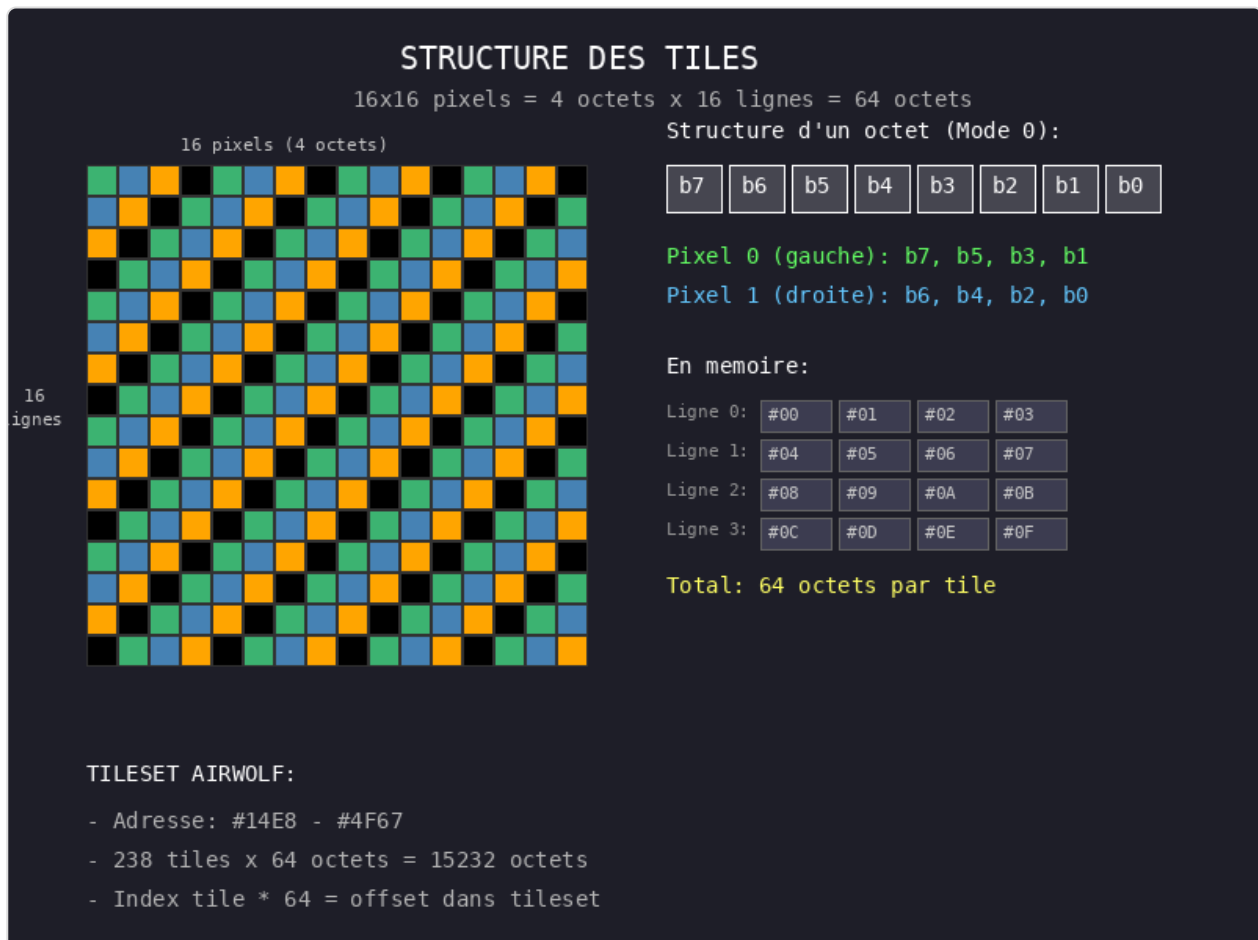


Figure 4 : Structure d'un tile - 16x16 pixels, 64 octets

Chaque tile mesure 16x16 pixels, soit 4 octets de large sur 16 lignes = 64 octets par tile. Le tileset complet occupe 15232 octets pour 238 tiles.

L'affichage d'un tile suit la logique de la memoire video entrelacee. La routine `DRAW_TILE` (#6E06) gere cette complexite :



```
DRAW_TILE:                ; #6E06
    LD B,#10                ; 16 lignes
.loop_y:
    PUSH BC
    PUSH HL
    LD B,#04                ; 4 octets par ligne
.loop_x:
    LD A,(DE)                ; Lire octet source
    LD (HL),A                ; Ecrire en VRAM
    INC DE
    INC HL
    DJNZ .loop_x
    POP HL
    ; Passage ligne suivante VRAM
    LD BC,#0800              ; +2048
    ADD HL,BC
    JR NC,.no_overflow
    LD BC,#3FB0              ; Correction debordement
    AND A
    SBC HL,BC
.no_overflow:
    POP BC
    DJNZ .loop_y
    RET
```

---

## Chapitre 5 : La memoire video entrelacee

### Anatomie de la VRAM

La memoire video de l'Amstrad CPC est structuree de maniere contre-intuitive. Les 16 Ko de VRAM (#C000-#FFFF) sont divises en 8 blocs de 2048 octets :

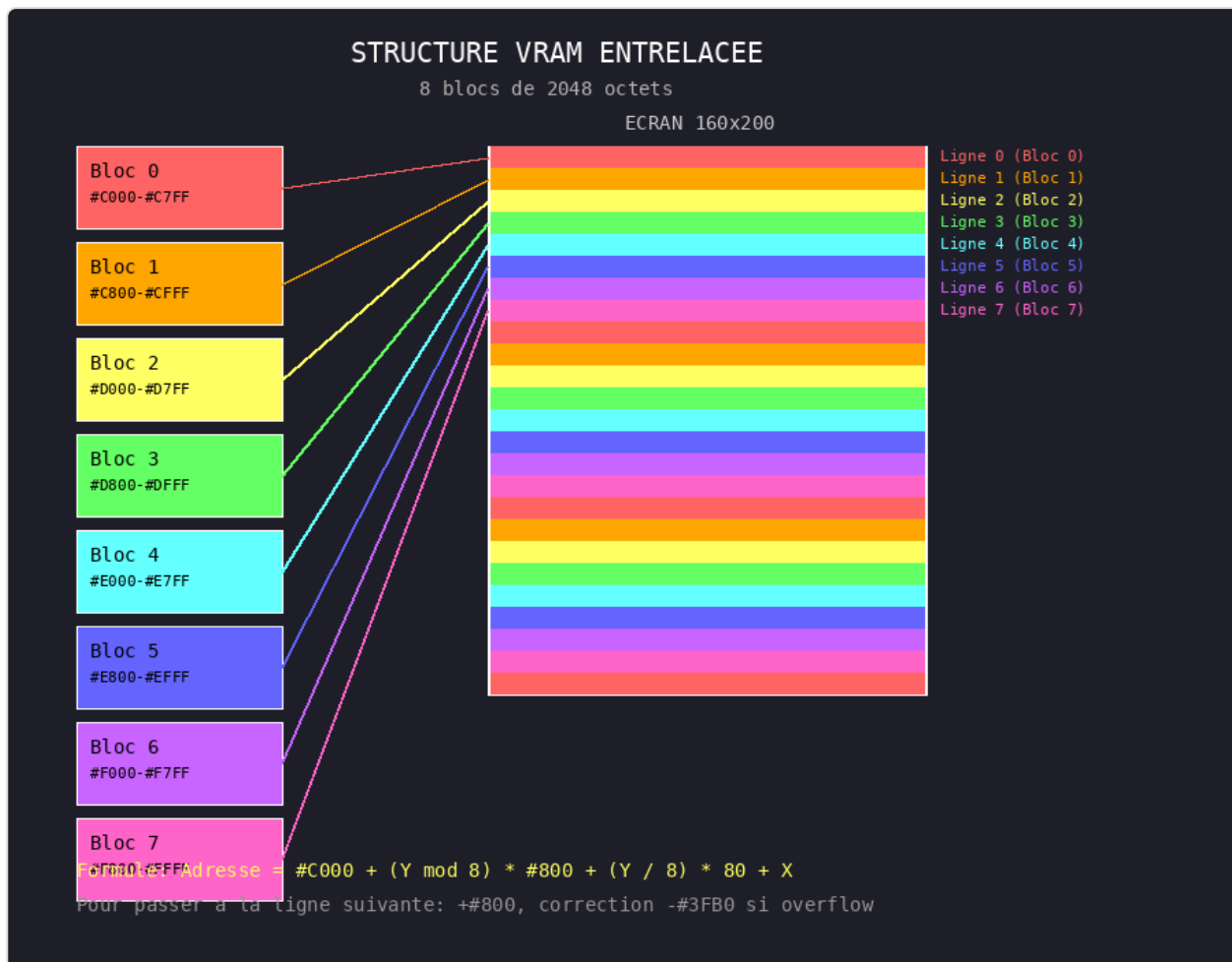


Figure 5 : L'entrelacement de la VRAM - 8 blocs pour 200 lignes

```

Bloc 0 : #C000-#C7FF (lignes 0, 8, 16, 24, 32...)
Bloc 1 : #C800-#CFFF (lignes 1, 9, 17, 25, 33...)
Bloc 2 : #D000-#D7FF (lignes 2, 10, 18, 26, 34...)
Bloc 3 : #D800-#DFFF (lignes 3, 11, 19, 27, 35...)
Bloc 4 : #E000-#E7FF (lignes 4, 12, 20, 28, 36...)
Bloc 5 : #E800-#EFFF (lignes 5, 13, 21, 29, 37...)
Bloc 6 : #F000-#F7FF (lignes 6, 14, 22, 30, 38...)
Bloc 7 : #F800-#FFFF (lignes 7, 15, 23, 31, 39...)

```

## La formule de calcul

Pour trouver l'adresse d'une ligne Y :

```

Si Y < 8:
    Adresse = #C000 + (Y * #800) + (X / 2)
Sinon:
    Adresse = #C000 + ((Y mod 8) * #800) + ((Y / 8) * 80) + (X / 2)

```

## Les tables pre-calculees

Plutot que de calculer ces adresses en temps reel, Airwolf utilise des tables pre-calculees. La routine `BUILD_HELICO_START_LINE_ARRAY` (#8064) remplit ces tables :

```
VRAM_LINE_START_ARRAY      EQU #A400  ; Pour l'helicoptere  
VRAM_LINE_START_ARRAY_OLD  EQU #A500  ; Position precedente  
VRAM_LINE_START_ARRAY_C    EQU #A600  ; Auxiliaire
```

Ces tables contiennent les adresses de debut de chaque ligne pour la zone d'affichage de l'helicoptere. Une optimisation cruciale pour les performances.

---

# TROISIEME PARTIE :

## L'HELICOPTERE

### Chapitre 6 : L'anatomie d'Airwolf

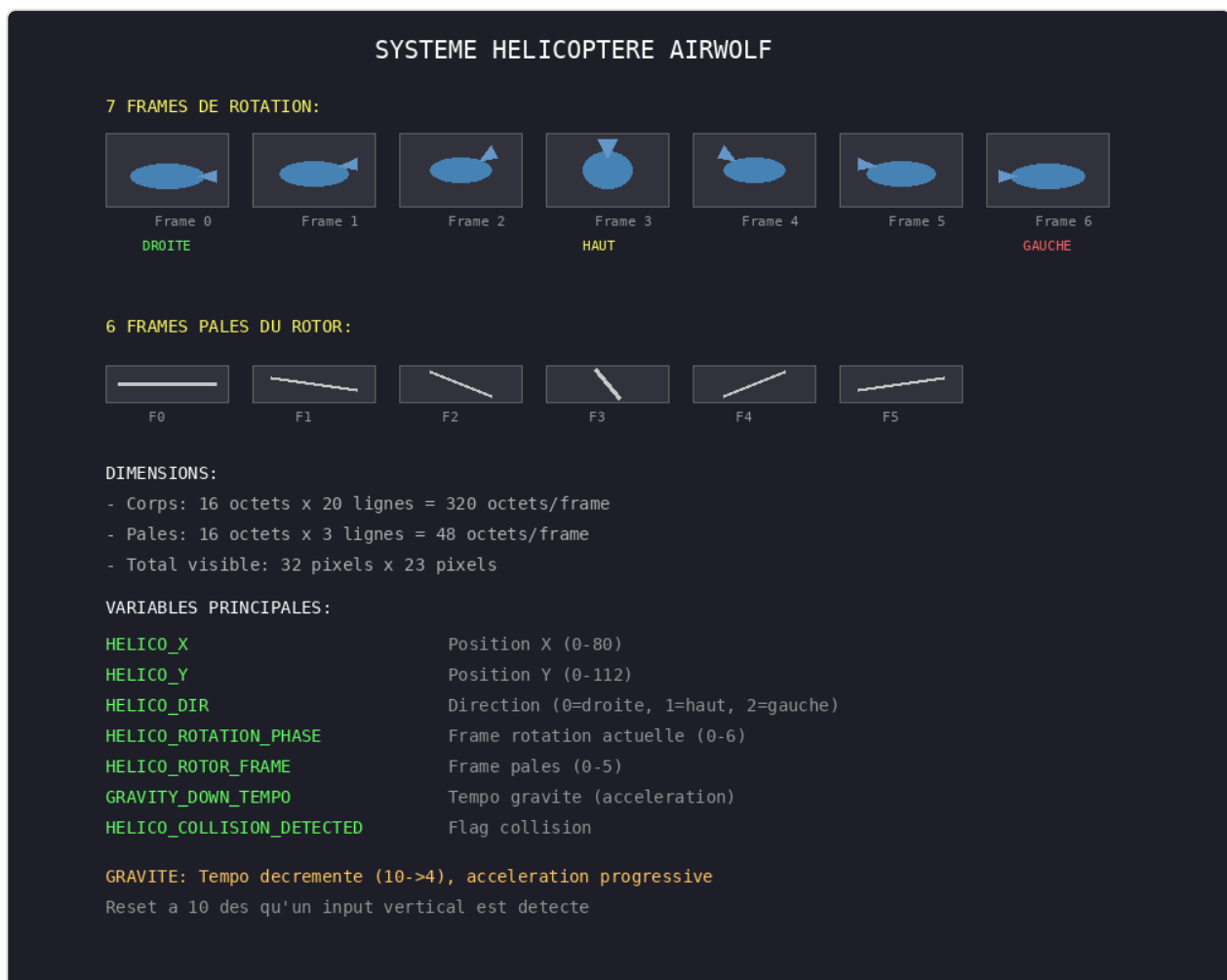


Figure 6 : Le systeme complet de l'helicoptere - rotation, pales, variables

### Les variables de l'helicoptere

L'etat de l'helicoptere est maintenu par un ensemble de variables situees dans la zone #79C0-#79F0 :

```

; Position
HELICO_X:          db ?      ; Coordonnee X (0-80 octets)
HELICO_Y:          db ?      ; Coordonnee Y (0-112 pixels)
HELICO_ADDR_VRAM:  dw ?      ; Adresse courante en VRAM

; Orientation et animation
HELICO_DIR:        db ?      ; Direction (0=droite, 1=haut, 2=gauche)
HELICO_ROTATION_PHASE: db ?    ; Phase de rotation (0-6)
HELICO_ROTOR_FRAME: db ?      ; Frame animation pales (0-5)
HELICO_SPRITE_ADDR: dw ?      ; Adresse sprite courant

; Physique
GRAVITY_DOWN_TEMPO: db ?      ; #79C4 - Tempo acceleration gravite
GRAVITY_DOWN_CPT:  db ?      ; #79D2 - Compteur gravite

; Etat
HELICO_COLLISION_DETECTED: db ? ; Flag collision
NB_LIFE_SHIELD:     db 6      ; #68AE - Nombre de vies

```

## Les dimensions

L'helicoptere occupe une zone de 16 octets (32 pixels) de large sur 23 pixels de haut :

```

HELICO_WIDTH  EQU 16    ; Largeur en octets
HELICO_HEIGHT EQU 23    ; Hauteur en pixels (20 corps + 3 pales)

```

## Chapitre 7 : Le systeme de rotation

### Sept frames, trois directions

L'helicoptere peut pointer dans trois directions : droite, haut, ou gauche. La transition entre ces directions utilise 7 frames d'animation :

```

Frame 0: Helicoptere pointe a droite (==>)
Frame 1: Legere rotation vers le haut
Frame 2: Rotation intermediaire
Frame 3: Helicoptere pointe vers le haut (==^)
Frame 4: Rotation vers la gauche
Frame 5: Rotation intermediaire
Frame 6: Helicoptere pointe a gauche (<==)

```

## L'algorithme de rotation

La rotation fonctionne par pas de 3 sur 7 frames. Elle est gérée dans la boucle principale `GAME_LOOP` (#7ABB) :

```
; Direction 0 (droite) -> Frame cible 0
; Direction 1 (haut)   -> Frame cible 3
; Direction 2 (gauche) -> Frame cible 6

HELICO_PROCESS_ROTATION:
    LD A,(HELICO_DIR)          ; Direction souhaitée (0, 1 ou 2)
    ADD A,A                    ; x2
    ADD A,(HELICO_DIR)         ; x3 (0->0, 1->3, 2->6)
    LD B,A                     ; B = frame cible

    LD A,(HELICO_ROTATION_PHASE); Frame actuelle
    CP B
    RET Z                      ; Déjà à la bonne frame

    JR C,.rotate_forward       ; Frame actuelle < cible
.rotate_backward:
    DEC A                      ; Rotation arrière
    JR .store_phase
.rotate_forward:
    INC A                      ; Rotation avant
.store_phase:
    LD (HELICO_ROTATION_PHASE),A

    ; Calculer l'adresse du sprite
    ; Chaque frame fait 320 octets (16 * 20)
    LD DE,320
    ; ... multiplication et ajout à l'adresse de base
```

La rotation est progressive : si le joueur passe de droite à gauche, l'hélicoptère traverse toutes les frames intermédiaires, donnant une impression de mouvement fluide.

## Chapitre 8 : Les pales du rotor

### Animation indépendante

Les pales du rotor tournent indépendamment de l'orientation de l'hélicoptère. Six frames d'animation créent l'illusion du mouvement. Les données des pales sont stockées à l'adresse `BASE_PALLE_HELICO` (#4F68) :

```

Frame 0: ====      (pales horizontales)
Frame 1: ===       (debut rotation)
Frame 2: ==        (milieu rotation)
Frame 3: =         (quasi verticales)
Frame 4: ==        (continuation)
Frame 5: ===       (retour)
[Retour Frame 0]

```

## Le code d'animation

La variable `PALLE_FRAME` (#7963) stocke la frame courante :

```

ANIMATE_HELICO_ROTOR:
    ; Incrementer frame
    LD A,(PALLE_FRAME)      ; #7963
    INC A
    CP 6                    ; 6 frames
    JR NZ,.no_loop
    XOR A                   ; Retour a 0
.no_loop:
    LD (PALLE_FRAME),A

    ; Calculer adresse sprite pales
    ; Chaque frame de pale fait 48 octets (16 * 3 lignes)
    LD HL,BASE_PALLE_HELICO ; #4F68
    LD DE,48
    ; ... multiplication par frame

    RET

```

Les pales sont dessinees separement, au-dessus du corps de l'helicoptere, permettant une superposition correcte avec le decor.

## Chapitre 9 : La gravite

### Un systeme d'acceleration

Airwolf simule une gravite realiste avec acceleration. Quand le joueur ne donne pas d'input vertical, l'helicoptere commence a tomber, de plus en plus vite. Ce systeme est gere dans `GAME_LOOP` (#7ABB) :

```

APPLY_GRAVITY:                ; Dans GAME_LOOP #7ABB
    ; Verifier si input vertical
    LD A,H                    ; Etat joystick
    AND #03                   ; Bits haut/bas
    JR NZ,.reset_gravity      ; Input -> pas de gravite

    ; Decrementer compteur
    LD A,(GRAVITY_DOWN_CPT)    ; #79D2
    DEC A
    LD (GRAVITY_DOWN_CPT),A
    JR NZ,.no_fall             ; Pas encore temps de tomber

    ; Accelerer
    LD A,(GRAVITY_DOWN_TEMPO)   ; #79C4
    LD (GRAVITY_DOWN_CPT),A    ; Recharger compteur
    DEC A
    LD (GRAVITY_DOWN_TEMPO),A   ; Tempo plus court = chute plus rapide

    ; Vitesse maximale
    CP #01
    JR NZ,.apply_fall
    LD A,#04
    LD (GRAVITY_DOWN_TEMPO),A   ; Plafonner a tempo 4

.apply_fall:
    SET 1,H                    ; Forcer input DOWN
    RET

.reset_gravity:
    LD A,#0A
    LD (GRAVITY_DOWN_TEMPO),A   ; Reset tempo initial
    LD (GRAVITY_DOWN_CPT),A
    RET

```

## L'effet ressenti

- Sans input : l'helicoptere commence a tomber lentement
- Le tempo diminue : 10, 9, 8, 7... la chute accelere
- Minimum atteint a 4 : vitesse terminale
- Un seul appui vers le haut : la gravite se reinitialise

Cette mecanique cree une sensation de poids et de pilotage exigeant.



## Chapitre 10 : Le systeme de tir

### Trois directions de tir

L'helicoptere peut tirer dans trois directions, correspondant a son orientation. La routine `FIRE_NEW_SHOOT` (#7898) gere l'initialisation d'un nouveau tir :

```
FIRE_NEW_SHOOT:                ; #7898
    ; Verifier qu'aucun tir n'est actif
    LD A,(NB_SHOOT_FIRED)      ; #79EE
    CP #01
    RET Z                      ; Un seul tir a la fois

    ; Direction = direction helicoptere
    LD A,(HELICO_DIR)
    LD (SHOOT_DIR),A

    ; Position de depart selon direction
    ; Direction 0 (droite): devant l'helicoptere
    ; Direction 1 (haut): au-dessus
    ; Direction 2 (gauche): a l'arriere

    LD A,#32                    ; Duree de vie: 50 frames
    LD (SHOOT_LIFE_CPT),A

    LD A,#01
    LD (NB_SHOOT_FIRED),A
    RET
```

### Detection des impacts

La routine `TEST_SHOOT_COLLISION` (#7C8D) detecte les collisions avec des pixels speciaux :

```
TEST_SHOOT_COLLISION:      ; #7C8D
    LD A,(HL)               ; Pixel ecran

    CP #0F                  ; Pixel solide
    JR Z,.hit_wall

    CP #F3                  ; Bouton destructible
    CALL Z,EXECUTE_BUTTON_ACTION ; #6971

    CP #B9                  ; Paroi gauche
    JR Z,.hit_wall

    CP #F6                  ; Paroi droite
    JR Z,.hit_wall

    CP #00                  ; Vide
    CALL NZ,SET_SHOOT_HIT_DETECTED
```

Certaines couleurs déclenchent des actions : détruire une cloison via **EXECUTE\_BUTTON\_ACTION** (#6971), activer un interrupteur, ou simplement être absorbées par le decor.

---

# QUATRIEME PARTIE : LE SCROLLING

## Chapitre 11 : La philosophie du defilement

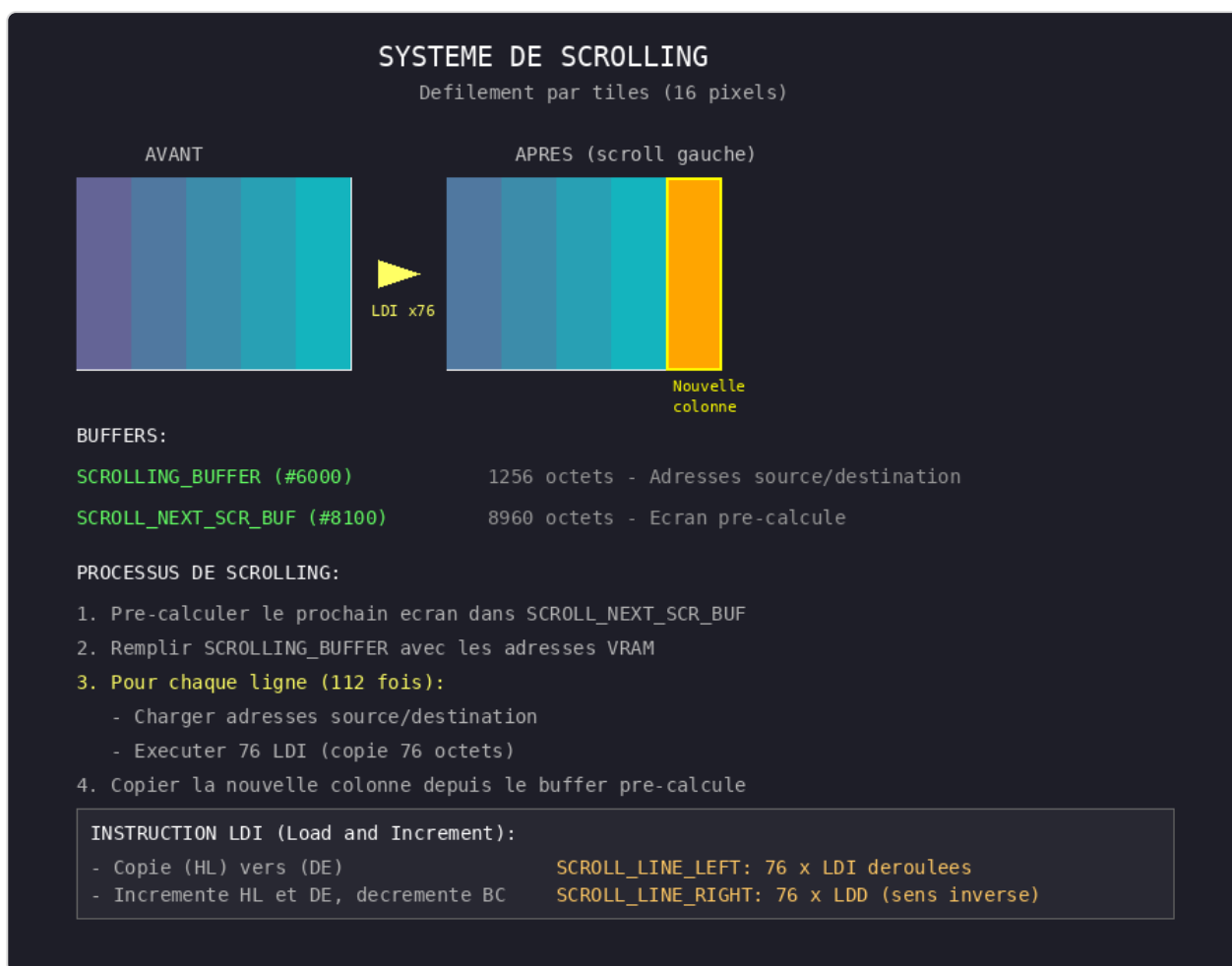


Figure 7 : Le systeme de scrolling par tiles

### Un scrolling par tiles

Airwolf n'utilise pas de scrolling pixel par pixel. Le monde defile par tiles entieres de 16 pixels. Cette approche simplifie enormement la gestion mais cree un effet saccade caracteristique.

## La zone de jeu

La zone de jeu occupe :

- Largeur : 80 octets (160 pixels)
- Hauteur : 112 pixels (7 tiles)

Le HUD (vies, timer, score) occupe le reste de l'ecran et n'est jamais scrolle.

## Les quatre directions

Le joueur peut scroller dans les quatre directions cardinales en atteignant les bords de la zone visible.

---

# Chapitre 12 : Le scrolling horizontal

## Le buffer de preparation

Avant de scroller, le jeu pre-calcule la nouvelle colonne de tiles dans un buffer :

```
SCROLL_NEXT_SCR_BUF EQU #8100 ; Buffer 8960 octets
```

## L'algorithme du scrolling gauche

La routine `SCROLL_LEFT` (#7360) gere le defilement vers la gauche :

```

SCROLL_LEFT:                                ; #7360
    ; 1. Remplir le buffer avec les adresses source/destination
    CALL FILL_SCROLLING_BUFFER_LEFT

    ; 2. Pour chaque ligne de l'ecran
    LD B,112                                ; 112 lignes
.loop_line:
    PUSH BC

    ; Charger les adresses depuis le buffer
    LD HL,(SCROLL_SRC_ADDR)
    LD DE,(SCROLL_DST_ADDR)

    ; Copier 76 octets vers la gauche
    ; (laisse 4 octets pour la nouvelle colonne)
    CALL SCROLL_LINE_LEFT                    ; #738F

    ; Mettre a jour le buffer
    CALL UPDATE_SCROLLING_BUFFER

    POP BC
    DJNZ .loop_line

    ; 3. Copier la nouvelle colonne depuis le pre-calcul
    CALL COPY_NEW_COLUMN_LEFT
    RET

```

## La copie par LDI

Le coeur du scrolling utilise l'instruction LDI (Load and Increment) en boucle deroulee. La routine `SCROLL_LINE_LEFT` (#738F) :

```

SCROLL_LINE_LEFT:                            ; #738F
    ; 76 instructions LDI deroulees
    LDI     ; Copie (HL) vers (DE), incremente les deux, decremente BC
    LDI
    LDI
    ; ... 76 fois au total
    RET

```

76 octets = 80 octets de zone de jeu - 4 octets de nouvelle colonne.

## Scrolling droite : LDD

Pour le scrolling vers la droite, la routine `SCROLL_LINE_RIGHT` (#7292) utilise LDD (Load and Decrement) en partant de la fin :

```

SCROLL_LINE_RIGHT:                ; #7292
    ; Source et destination pointent vers la fin de la ligne
    ; 76 instructions LDD deroulees
    LDD        ; Copie (HL) vers (DE), decremente les deux
    LDD
    LDD
    ; ... 76 fois
    RET

```

## Chapitre 13 : Le scrolling vertical

### Copier des bandes horizontales

Le scrolling vertical fonctionne par bandes de 8 pixels (correspondant aux blocs VRAM). La routine `NEXT_VIEW_TO_SCREEN_DOWN` (#724E) gere le scrolling vers le bas :

```

NEXT_VIEW_TO_SCREEN_DOWN:        ; #724E
    LD A,#0E                      ; 14 iterations (112/8 lignes)
.loop:
    PUSH AF
    CALL COPY_VRAM_TO_NEXT_VIEW_SCROLL_DOWN ; #74B8
    CALL COPY_NEXT_VIEW_TO_BOTTOM_SCREEN    ; #70C0
    POP AF
    DEC A
    JR NZ,.loop
    RET

```

La routine equivalente pour le scrolling vers le haut est `NEXT_VIEW_TO_SCREEN_UP` (#726B).

### La gestion de l'entrelacement

Le scrolling vertical est plus complexe car il doit gerer l'entrelacement de la VRAM. Copier "une ligne vers le bas" signifie en realite :

1. Bloc 0 -> Bloc 1 (lignes 0,8,16... -> lignes 1,9,17...)
2. Bloc 1 -> Bloc 2
3. ...
4. Bloc 7 -> Bloc 0 du caractere suivant

## Chapitre 14 : Le pre-calcul de l'ecran suivant

### Dessiner dans l'ombre

Avant de scroller, le jeu dessine le prochain ecran complet dans un buffer invisible. La routine `DRAW_SCROLLING_NEXT_SCREEN` se charge de cette preparation :

```
DRAW_SCROLLING_NEXT_SCREEN:
    LD HL,SCROLL_NEXT_SCR_BUF    ; #8100
    ; Pour chaque tile du prochain ecran
    LD A,#07                      ; 7 lignes de tiles
.loop_y:
    PUSH AF
    LD A,#14                      ; 20 colonnes
.loop_x:
    PUSH AF
    ; Calculer l'index du tile dans la tilemap
    ; Chercher le graphisme dans le tileset
    ; Copier les 64 octets du tile dans le buffer
    ; Incrementer la position
    POP AF
    DEC A
    JR NZ,.loop_x
    POP AF
    DEC A
    JR NZ,.loop_y
    RET
```

### Avantage de cette methode

Le pre-calcul permet :

1. Un scrolling plus rapide (simple copie, pas de calculs)
  2. Pas de clignotement ni d'artefacts
  3. Separation claire entre preparation et affichage
-

# CINQUIEME PARTIE : LES COLLISIONS

## Chapitre 15 : La detection integree

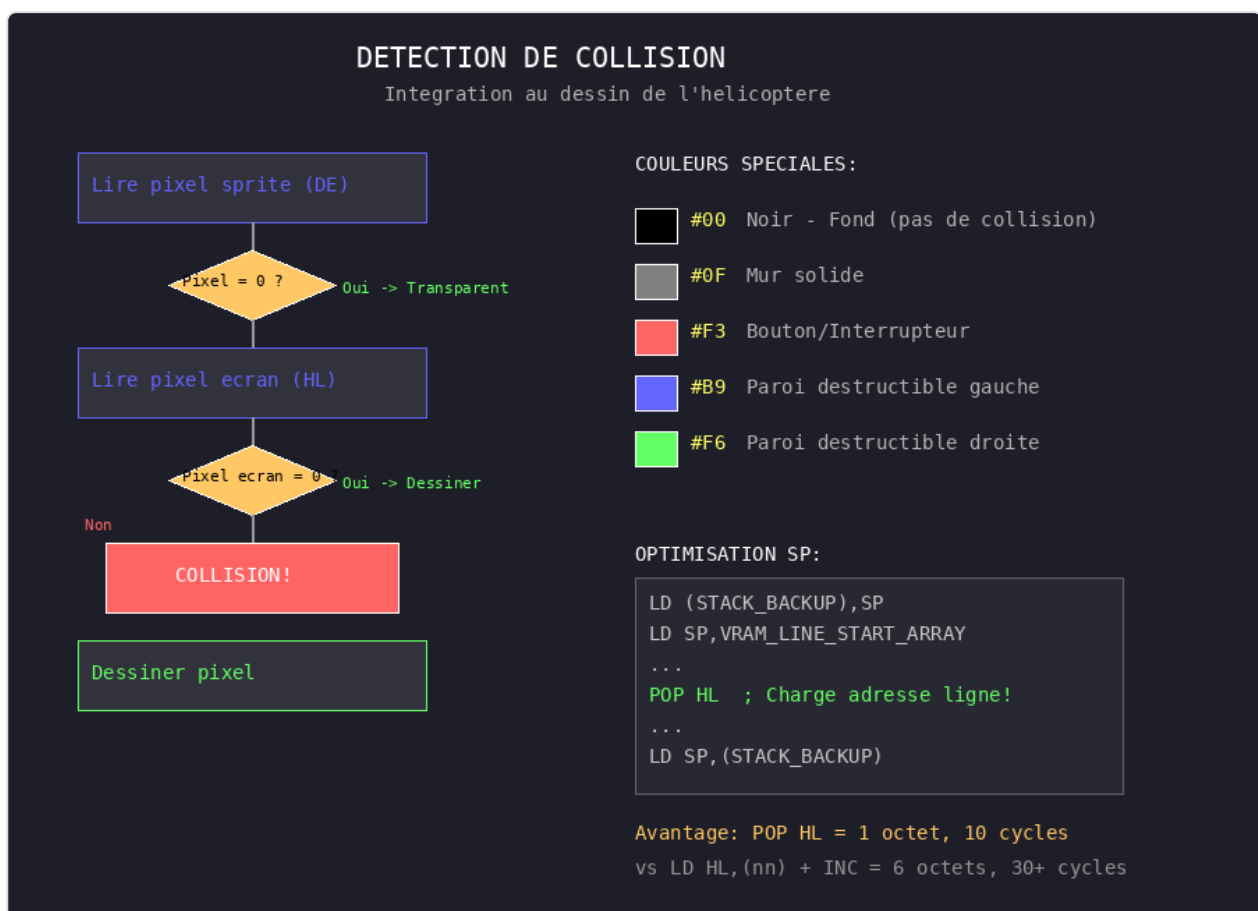


Figure 8 : Detection de collision integree au dessin

### Collision pendant le dessin

L'une des techniques les plus elegantes d'Airwolf : la detection des collisions est integree directement dans la routine d'affichage de l'helicoptere `TEST_AND_DRAW_HELICO` (#7E8A).



Plutot que :

1. Dessiner l'helicoptere
2. Tester les collisions

Le jeu fait les deux en un seul passage dans `DRAW_HELICO` (#7F1C) :

```

DRAW_HELICO:                                ; #7F1C
    ; Pour chaque pixel du sprite
    LD A,(DE)                                ; Pixel du sprite helicoptere
    OR A
    JR Z,.transparent                        ; Si 0, pixel transparent

    LD A,(HL)                                ; Pixel de l'ecran
    OR A
    JR Z,.draw                              ; Si 0, ecran vide -> dessiner

    ; Pixel sprite non-nul ET pixel ecran non-nul = collision!
    LD A,#01
    LD (HELICO_COLLISION_DETECTED),A
    JR .next_pixel

.draw:
    LD A,(DE)
    LD (HL),A                                ; Dessiner le pixel
.transparent:
.next_pixel:
    INC DE
    INC HL

```

## Collision pixel-parfaite

Cette methode produit une detection au pixel pres. Si un seul pixel non-transparent de l'helicoptere chevauche un pixel non-noir du decor, une collision est detectee et la routine `HELICO_LOSE_LIFE` (#68BB) est appelee.

# Chapitre 16 : Les couleurs magiques

## La palette comme systeme de collision

Une decouverte fascinante : la palette contient des couleurs dupliques qui servent de marqueurs pour la logique du jeu :

```

Index #00 : Noir (fond, pas de collision)
Index #0F : Mur solide
Index #F3 : Bouton/Interrupteur
Index #B9 : Paroi destructible gauche
Index #F6 : Paroi destructible droite

```

Visuellement identiques au noir ou aux couleurs adjacentes, ces pixels speciaux sont invisibles mais detectables par le code.

## Un design emergent

Ce systeme suggere une evolution du design pendant le developpement. Les collisions par couleur sont simples a implementer mais rigides. Elles obligent les graphistes a peindre avec des couleurs specifiques, creant une dependance forte entre art et code.

# Chapitre 17 : L'optimisation par la pile

## Le detournement du Stack Pointer

Pour afficher l'helicoptere rapidement, Airwolf detourne le registre SP (Stack Pointer) dans `DRAW_HELICO` (#7F1C) :

```

DRAW_HELICO:                ; #7F1C
    LD (STACK_BACKUP),SP    ; Sauvegarder la pile

    LD SP,VRAM_LINE_START_ARRAY ; #A400 - SP pointe vers la table d'adresses

    LD B,#14                ; 20 lignes
.loop_y:
    POP HL                  ; Charge l'adresse de la ligne!

    ; Dessiner les 16 pixels de la ligne
    ; (boucle deroulee)

    DJNZ .loop_y

    LD SP,(STACK_BACKUP)    ; Restaurer la pile
    RET

```

## Pourquoi ca marche

`POP HL` charge HL avec les 2 octets pointés par SP, puis incremente SP de 2. En pointant SP vers une table d'adresses pre-calculees par `BUILD_HELICO_START_LINE_ARRAY` (#8064), chaque POP charge instantanement l'adresse de la ligne suivante.

C'est plus rapide que :

```
LD HL, (TABLE_ADDR)
INC HL
INC HL
LD (TABLE_ADDR), HL
```

## Le danger

Cette technique est dangereuse : toute interruption pendant ce code corromprait la table d'adresses au lieu de la pile. Les interruptions doivent etre desactivees ou le code doit etre infaillible.

---

# Chapitre 18 : L'effacement securise

## Ne jamais effacer ce qui ne nous appartient pas

Quand l'helicoptere se deplace, il faut effacer son ancienne position. Mais attention : on ne doit effacer que les pixels qu'on a dessines, pas le decor !

```
ERASE_HELICO:
    LD SP,VRAM_LINE_START_ARRAY_OLD ; #A500 - Adresses de l'ancienne position
    LD IX,VRAM_LINE_START_ARRAY     ; #A400 - Adresses de la nouvelle
    LD IY,SPRITE_DATA                ; Donnees du sprite

    LD B,#14                          ; 20 lignes
.loop_y:
    POP HL                            ; Ancienne adresse

    ; Pour chaque pixel
    LD A,(IY+0)                       ; Pixel du sprite
    OR A
    JR Z,.skip                        ; Transparent -> ne pas effacer

    ; Comparer ancienne et nouvelle position
    ; Si le pixel fait maintenant partie du decor, ne pas effacer

    XOR A
    LD (HL),A                         ; Effacer (mettre a noir)

.skip:
    INC HL
    INC IY
    ; ...
```

---

# SIXIEME PARTIE : LES ENNEMIS

## Chapitre 19 : La structure des entites

STRUCTURE DES ENNEMIS			PATTERNS:
Offset	Description	Taille	
#00-01	Adresse VRAM initiale	2	ANIM_RL_45 D-G, 45 frames
#02-03	Pointeur pattern animation	2	
#04-05	Adresse sprite de base	2	ANIM_RULD Carre
#06	Largeur (octets)	1	ANIM_STATIC Immobile
#07	Hauteur (pixels)	1	
#08	Tempo mouvement initial	1	Format pattern:
#09	Compteur tempo mouvement	1	
#0A-0B	Adresse sprite courante	2	db DIRECTION, DUREE db DIRECTION, DUREE db #FF ; Fin/loop
#0C	Nombre frames total	1	
#0D	Frame courante	1	DOUBLE ANIMATION:
#0E	Tempo animation	1	
#0F	Compteur tempo animation	1	Animation MOUVEMENT (pattern)
#10	Flag actif (0=inactif)	1	
#11-12	Adresse VRAM courante	2	Animation GRAPHIQUE (sprites)
#13	Duree mouvement initial	1	
#14	Compteur duree mouvement	1	Les deux animations sont independantes et fonctionnent en parallele
#15-16	Pointeur pattern courant	2	
#17	Flag reset position	1	

Figure 9 : Structure de 24 octets definissant chaque ennemi

## 24 octets par ennemi

Chaque ennemi est defini par une structure de 24 octets, geree par la routine `ANIMATE_ENEMY` (#6F5B) :

Offset	Taille	Role
#00-#01	2	Adresse VRAM initiale
#02-#03	2	Pointeur pattern animation
#04-#05	2	Adresse sprite de base
#06	1	Largeur en octets
#07	1	Hauteur en pixels
#08	1	Tempo mouvement initial
#09	1	Compteur tempo mouvement
#0A-#0B	2	Adresse sprite courante
#0C	1	Nombre de frames total
#0D	1	Frame courante
#0E	1	Tempo animation
#0F	1	Compteur tempo animation
#10	1	Flag actif (0=inactif)
#11-#12	2	Adresse VRAM courante
#13	1	Duree mouvement initial
#14	1	Compteur duree mouvement
#15-#16	2	Pointeur pattern courant
#17	1	Flag reset position

## Le double systeme d'animation

Chaque ennemi a deux animations independantes :

1. **Animation graphique** : cycle des frames du sprite
2. **Animation de mouvement** : pattern de déplacement

## Chapitre 20 : Les patterns de mouvement

### Definition des patterns

Les mouvements sont definis comme des sequences de paires (direction, duree) :

```

ANIM_RL_45:                                ; Droite-Gauche, 45 frames chaque
    db INPUT_RIGHT, 45
    db INPUT_LEFT, 45
    db #FF                                ; Marqueur de fin/loop

ANIM_RULD_20302030:                        ; Carre
    db INPUT_RIGHT, 20
    db INPUT_UP, 30
    db INPUT_LEFT, 20
    db INPUT_DOWN, 30
    db #FF

ANIM_STATIC:                              ; Immobile
    db 0, 255
    db #FF

```

### L'algorithme d'animation

La routine `ANIMATE_ENEMY` (#6F5B) gere le mouvement de chaque ennemi :

```

ANIMATE_ENEMY:                                ; #6F5B
    ; Verifier si actif
    LD A,(IX+#10)
    OR A
    RET Z

    ; Decrementer tempo mouvement
    LD A,(IX+#09)
    DEC A
    LD (IX+#09),A
    RET NZ                                     ; Pas encore temps de bouger

    ; Recharger tempo
    LD A,(IX+#08)
    LD (IX+#09),A

    ; Lire direction du pattern
    LD L,(IX+#15)
    LD H,(IX+#16)
    LD A,(HL)

    CP #FF                                     ; Fin du pattern?
    JR Z,.loop_pattern

    ; Appliquer la direction
    CALL MOVE_SPRITE                          ; #6F34

    ; Decrementer duree
    LD A,(IX+#14)
    DEC A
    LD (IX+#14),A
    RET NZ

    ; Passer a l'etape suivante du pattern
    LD L,(IX+#15)
    LD H,(IX+#16)
    INC HL
    INC HL                                     ; Sauter direction+duree
    LD (IX+#15),L
    LD (IX+#16),H

    ; Charger nouvelle duree
    INC HL
    LD A,(HL)
    LD (IX+#14),A
    RET

.loop_pattern:
    ; Retour au debut du pattern
    LD L,(IX+#02)
    LD H,(IX+#03)
    LD (IX+#15),L
    LD (IX+#16),H
    RET

```



La routine `ANIMATE_ALL_ENEMIES` (#6FE1) appelle `ANIMATE_ENEMY` pour chaque ennemi actif.

## Chapitre 21 : L'animation graphique

### Le cycle des frames

Indépendamment du mouvement, le sprite de l'ennemi change régulièrement :

```
ANIMATE_ENEMY_SPRITE:
    ; Decrementer tempo animation
    LD A,(IX+#0F)
    DEC A
    LD (IX+#0F),A
    RET NZ

    ; Recharger tempo
    LD A,(IX+#0E)
    LD (IX+#0F),A

    ; Passer a la frame suivante
    LD A,(IX+#0D)
    INC A
    CP (IX+#0C)                ; Depasse nb frames?
    JR NZ,.no_loop
    XOR A                      ; Retour frame 0
.no_loop:
    LD (IX+#0D),A

    ; Calculer nouvelle adresse sprite
    ; Nouvelle adresse = base + (frame * taille_frame)
    ; taille_frame = largeur * hauteur
    ; ...

    RET
```

### Types d'ennemis

Le jeu contient plusieurs types d'ennemis avec des comportements distincts. Les sprites sont stockés dans le tileset :

- **Statiques** : tourelles fixes (T\_RADAR\_UP #3DA8, T\_RADAR\_DOWN #3EA8)
- **Patrouilleurs** : mouvement horizontal (T\_BLUE\_BOT #17A8, T\_TANK\_BOT #3BA8)

- **Erratiques** : patterns complexes (T\_EYE\_BOT #1C68, T\_FLAME #18E8)
-

# SEPTIEME PARTIE : LE SON

## Chapitre 22 : Le PSG AY-3-8912

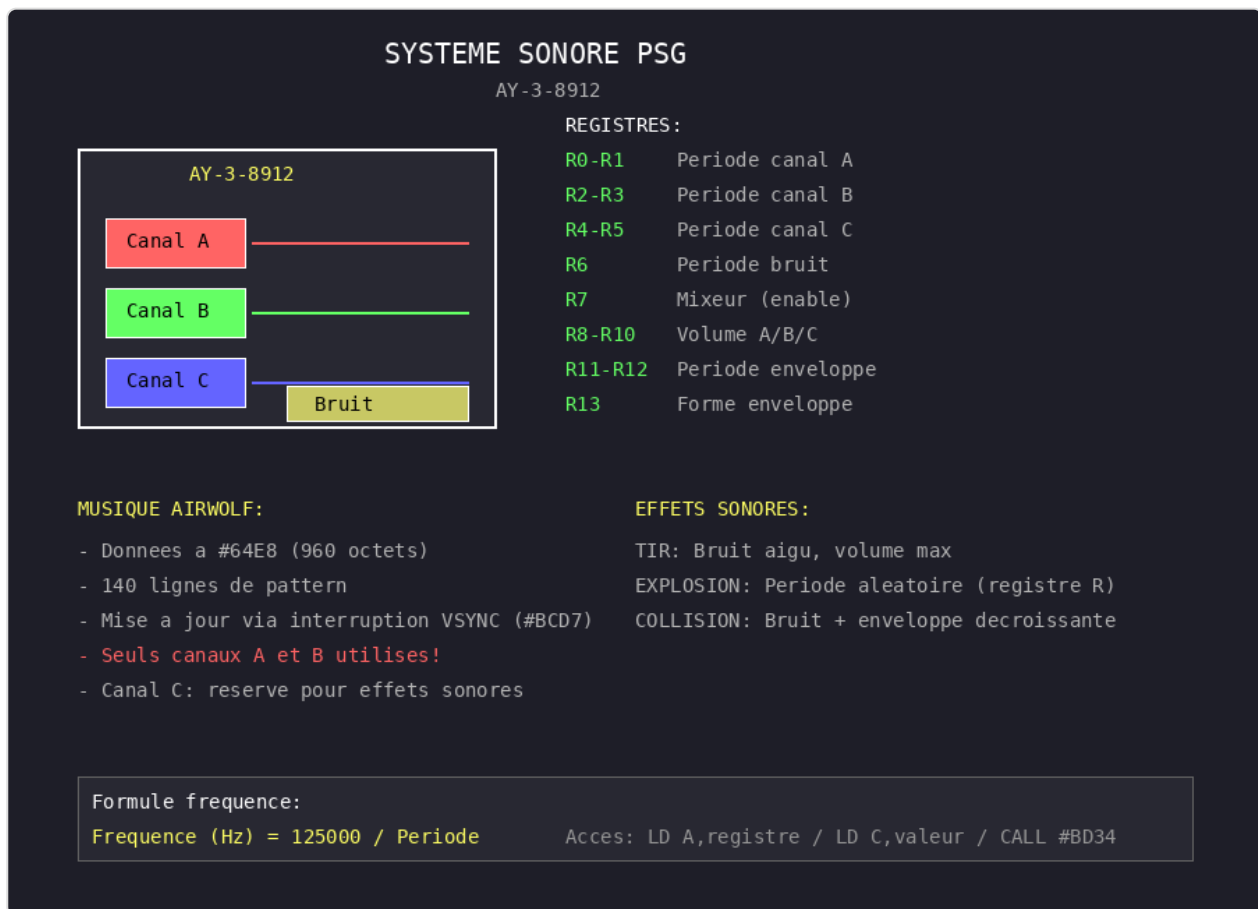


Figure 10 : Le circuit sonore PSG AY-3-8912

## Architecture sonore

L'Amstrad CPC utilise le circuit AY-3-8912, offrant :

- 3 canaux melodiques
- 1 generateur de bruit partageable
- Enveloppes materielles

## Les registres

Registre	Fonction
R0-R1	Periode canal A
R2-R3	Periode canal B
R4-R5	Periode canal C
R6	Periode bruit
R7	Mixeur (active/desactive canaux)
R8-R10	Volume canaux A/B/C
R11-R12	Periode enveloppe
R13	Forme enveloppe

## Acces via firmware

```
; Ecrire valeur C dans registre A
LD A,registre
LD C,valeur
CALL #BD34          ; SOUND DIRECT REG
```

# Chapitre 23 : La musique

## Structure des donnees

La musique est stockee comme une sequence de valeurs de registres a partir de **MUSIC\_DATA\_BASE** (#64E8) :

```
MUSIC_DATA_BASE      EQU #64E8    ; Debut des donnees
MUSIC_PTR:           dw 0          ; #6AB3 - Pointeur courant
MUSIC_PATTERN_LINE_CPT: db 0      ; #6AB2 - Compteur de lignes
NB_MUSIC_PATTERN_LINES EQU 140     ; #8C lignes par pattern

; Format: periode_A_low, periode_A_high, periode_B_low, periode_B_high
; 0 = pas de changement
```

## La routine de mise a jour

La routine `UPDATE_SOUND_REGISTERS` (#6B09) est appelee par `UPDATE_MUSIC` (#6B5D) a chaque frame via l'interruption VSYNC :

```
UPDATE_SOUND_REGISTERS:      ; #6B09
    LD HL,(MUSIC_PTR)        ; #6AB3

    ; Canal A
    LD A,(HL)
    OR A
    JR Z,.skip_a             ; #6B1F - 0 = pas de changement

    LD C,A
    LD A,#00                  ; Registre periode A low
    CALL #BD34
    INC HL

    LD C,(HL)
    LD A,#01                  ; Registre periode A high
    CALL #BD34

.skip_a:                     ; #6B1F
    ; Canal B similaire...

    ; Avancer le pointeur
    ; Gerer le bouclage
    ; ...
```

L'initialisation est faite par `INIT_MUSIC` (#6ABE) qui configure aussi l'interruption via `INIT_MUSIC_INTERRUPT` (#6B4F).

## Le canal oublie

L'article original mentionne une decouverte troublante : le player musical ignore complètement le troisieme canal sonore. Trois canaux sont disponibles, mais seuls deux sont utilises. Un indice supplementaire d'un developpement precipite.

# Chapitre 24 : Les effets sonores

## Le bruit d'explosion

La routine `HELICO_EXPLOSION_SOUND` (#7CB7) utilise le registre R du Z80 comme source de nombres pseudo-aleatoires :

```

HELICO_EXPLOSION_SOUND:      ; #7CB7
    LD A,R                    ; Valeur "aleatoire"
    LD C,A
    LD A,#06                  ; Periode bruit
    CALL #BD34

    LD A,#0D                  ; Forme enveloppe
    LD C,#00                  ; Decroissante
    CALL #BD34

```

## Le bruit de tir et collision

La routine **PLAY\_NOISE\_SOUND** (#6B79) joue les effets sonores :

```

PLAY_NOISE_SOUND:            ; #6B79
    LD A,#0C                  ; Volume canal C
    LD C,#0F                  ; Maximum
    CALL #BD34

    LD A,#06                  ; Periode bruit
    LD C,#08                  ; Bruit aigu
    CALL #BD34

    LD A,#01
    LD (NOISE_SOUND_PLAYED),A ; #6AB0
    RET

```

La routine **SFX\_OFF** (#6B96) desactive les effets sonores apres un certain temps.

---

# HUITIEME PARTIE : LES REVELATIONS

---

## | Chapitre 25 : Le niveau 2 n'existe pas

### La verite dans le code

L'exploration complete du code revele la verite : il n'y a pas de niveau 2. La tilemap unique contient l'integralite du monde jouable a partir de `BASE_TILEMAP` (#03E8). Les adresses de chargement, les routines de transition de niveau : inexistantes.

### Un monde clos

Le monde d'Airwolf est une carte fixe de tiles, pre-chargee en memoire. Le joueur la parcourt, active des interrupteurs via `EXECUTE_BUTTON_ACTION` (#6971), detruit des cibles. Mais il n'y a pas de "suite".

---

## Chapitre 26 : Le crash intentionnel

### LE MYSTERE DU CRASH

La boucle infinie intentionnelle

```
FINAL_ZONE_CHECK:
  LD A, (SCREEN_FLAGS)
  AND %00000011
  CP %00000011
  JR NZ, .infinite

.infinite:
  JR .infinite ; BLOQUE!
```

CE QUE CA SIGNIFIE:

1. Le jeu verifie 2 flags
2. La logique est incorrecte
3. Les flags ne peuvent jamais etre actives simultanement
4. Le jeu boucle indefiniment

=> CE N'EST PAS UN BUG  
=> C'EST INTENTIONNEL

**CONCLUSION:**

Les developpeurs savaient que le jeu ne pouvait pas etre termine. Plutot que corriger, ils ont cache le probleme avec cette boucle.

Figure 11 : La boucle infinie intentionnelle

## Les interrupteurs binaires

Le jeu utilise un systeme d'interrupteurs lies a des ecrans specifiques via `VIEW_INDEX` (#6970). Chaque interrupteur est un bit dans un octet de flags :

```
SCREEN_FLAGS:
  db %00000000    ; 8 flags pour 8 zones
```

## La logique defaillante

Dans la zone finale, deux interrupteurs doivent etre actives. Mais leur logique viole le design du reste du jeu. Plutot que de corriger le probleme, les developpeurs ont insere une boucle infinie :



```

FINAL_ZONE_CHECK:
    LD A,(SCREEN_FLAGS)
    AND %00000011      ; Les deux flags finaux
    CP %00000011      ; Les deux actives?
    JR NZ,.infinite     ; Non -> bloquer le jeu

    ; Code de fin de jeu jamais atteint

.infinite:
    JR .infinite        ; Le "crash"

```

## L'aveu d'echec

Cette boucle est un aveu. Les developpeurs savaient que le jeu ne pouvait pas etre termine correctement. Plutot que de livrer un produit non fonctionnel de maniere visible, ils ont opte pour un blocage silencieux que peu de joueurs atteindraient jamais.

---

# Chapitre 27 : Les vestiges du scrolling pixel

## Du code mort

Dans les entrailles du programme, des routines entieres ne sont jamais appelees. Parmi elles, `UNUSED_FULL_SCROLL_LEFT` (#7382) contient du code de scrolling pixel par pixel :

```

UNUSED_FULL_SCROLL_LEFT:      ; #7382
    ; Decale l'ecran d'un pixel
    ; Plus fluide mais plus lent
    CALL SCROLL_LEFT          ; #7360
    ; ...

```

## L'hypothese du changement de design

Ces routines suggerent un design initial different : un scrolling fluide, pixel par pixel, comme les meilleurs jeux de l'epoque. Abandonner ce systeme pour un scrolling par tiles indique soit des contraintes de performance, soit un changement d'equipe, soit les deux.

---

## Chapitre 28 : Plusieurs mains

### Les styles de code

L'analyse revelatrice montre des styles de programmation differents selon les sections :

- Certaines routines sont elegantes, optimisees (comme `DRAW_HELICO` #7F1C)
- D'autres sont verbeuses, repetitives
- Les conventions de nommage varient
- L'utilisation des registres est incoherente

### L'assemblage final

Airwolf n'est pas le travail d'un seul developpeur. C'est un patchwork, assemble a partir de contributions multiples, probablement sous pression temporelle pour respecter la licence.

---

# NEUVIEME PARTIE : LA RENAISSANCE

---

## Chapitre 29 : Airwolf Pico

### Reimaginer le jeu

L'exploration du code original a inspire une question : a quoi ressemblerait Airwolf s'il avait ete correctement concu ? La reponse prend la forme d'Airwolf Pico, une reimagination sur la plateforme PICO-8.

### Les corrections apportees

- Vrais niveaux multiples
  - Scrolling fluide
  - Detection de collision coherente
  - Fin de jeu accessible
  - Equilibrage de la difficulte
- 

## Chapitre 30 : Airwolf Reloaded CPC

### Retour aux sources

En collaboration avec l'artiste Titan, une version corrigees pour Amstrad CPC voit le jour. Airwolf Reloaded n'est pas un remake : c'est le jeu original, debugge et complete.

### Les modifications

- Correction de la boucle infinie finale

- Logique d'interrupteurs repensee
- Nouveaux graphismes par Titan
- Jeu completable de bout en bout

## **Preserver l'esprit**

L'objectif n'etait pas de creer un nouveau jeu, mais de reveler celui qui aurait du exister. Les mecaniques originales sont preservees, seuls les bugs bloquants sont corriges.

---

## DIXIEME PARTIE : REFLEXIONS

---

### Chapitre 31 : Archeologie numerique

#### Ce que le code raconte

Au-dela des octets et des registres, le code source d'Airwolf raconte une histoire humaine :

- Des developpeurs sous pression
- Des compromis forces
- Des ambitions revues a la baisse
- Une deadline impossible

#### Les indices du stress

Le code mort, les styles melanges, la boucle infinie : autant de traces d'un developpement chaotique. On devine les reunions de crise, les nuits blanches, les decisions douloureuses.

---

### Chapitre 32 : La valeur de la curiosite

#### Guerir par l'exploration

L'enquete sur Airwolf est nee de l'insomnie et de la maladie. Elle est devenue therapeutique. L'exploration methodique, la satisfaction des decouvertes, le sentiment de comprendre : autant de remedes inattendus.

#### La perseverance recompensee

Quarante ans apres sa sortie, Airwolf livre enfin ses secrets. Il aura fallu de la patience, des outils modernes, et surtout une curiosite tenace pour percer ses mysteres.

## Chapitre 33 : Heritage technique

### Ce qu'Airwolf nous enseigne

Malgre ses defauts, Airwolf contient des techniques remarquables :

- L'utilisation de SP comme pointeur rapide ( `DRAW_HELICO` #7F1C)
- La detection de collision integree au rendu ( `TEST_AND_DRAW_HELICO` #7E8A)
- Les tables pre-calculees pour les adresses VRAM ( `BUILD_HELICO_START_LINE_ARRAY` #8064)
- Le systeme de patterns pour l'animation ( `ANIMATE_ENEMY` #6F5B)

Ces techniques restent pertinentes pour tout developpeur de jeux retro aujourd'hui.

---

# ANNEXES

---

## Annexe A : Palette de couleurs

Index	Couleur	Hardware
-----		
0	Noir	#00
1	Bleu	#04
2	Vert	#0B
3	Cyan	#0C
4	Rouge	#03
5	Magenta	#0D
6	Jaune	#06
7	Blanc	#0E
8-15	Variables	Selon ecran

## Annexe B : Map memoire complete

```
#0000-#003F : Variables systeme Z80
#0040-#005F : Zone de travail
#0060-#00FF : Code d'initialisation (BOOT)
#03E8-#14E7 : TILEMAP (BASE_TILEMAP)
#14E8-#4F67 : TILESET (BASE_TILESET - 238 tiles x 64 octets)
#4F68-#4FFF : Sprites pales (BASE_PALLE_HELICO - 6 frames x 48 octets)
#5000-#5FFF : Sprites explosion (BASE_EXPLOSION)
#6000-#64E7 : Buffer scrolling (SCROLLING_BUFFER)
#64E8-#68A7 : Donnees musicales (MUSIC_DATA_BASE)
#68A8      : Point d'entree (START_POINT)
#68AB      : LOSE_LIFE_TEMPO_CPT
#68AE      : NB_LIFE_SHIELD
#68BB      : HELICO_LOSE_LIFE
#6971      : EXECUTE_BUTTON_ACTION
#6ABE      : INIT_MUSIC
#6B09      : UPDATE_SOUND_REGISTERS
#6B5D      : UPDATE_MUSIC
#6B79      : PLAY_NOISE_SOUND
#6BE4      : DRAW_SPRITE_TILE
#6D37      : DECREMENT_TIMER
#6E06      : DRAW_TILE
#6E23      : DRAW_TILE_MAP
#6F34      : MOVE_SPRITE
#6F5B      : ANIMATE_ENEMY
#6FE1      : ANIMATE_ALL_ENEMIES
#7292      : SCROLL_LINE_RIGHT
#724E      : NEXT_VIEW_TO_SCREEN_DOWN
#726B      : NEXT_VIEW_TO_SCREEN_UP
#7360      : SCROLL_LEFT
#738F      : SCROLL_LINE_LEFT
#745D      : SCROLL_RIGHT
#7898      : FIRE_NEW_SHOOT
#7ABB      : GAME_LOOP
#7C8D      : TEST_SHOOT_COLLISION
#7CB7      : HELICO_EXPLOSION_SOUND
#7D89      : GAME_START
#7DA1      : MAIN_LOOP
#7DC2      : WAIT_FIRE_BUTTON_PRESS
#7E8A      : TEST_AND_DRAW_HELICO
#7F1C      : DRAW_HELICO
#8064      : BUILD_HELICO_START_LINE_ARRAY
#80AB      : CHEAT_TIMER_KEY
#80BB      : GRAVITY_KEY
#80CB      : PASSE_MURAILLE_KEY
#8100      : SCROLL_NEXT_SCR_BUF
#A400      : VRAM_LINE_START_ARRAY
#A500      : VRAM_LINE_START_ARRAY_OLD
#A600      : VRAM_LINE_START_ARRAY_C
#C000-#FFFF : Memoire video (8 blocs x 2048 octets)
```



## Annexe C : Codes de triche

Touche	Routine	Action
T	CHEAT_TIMER_KEY (#80AB)	Desactive le timer
G	GRAVITY_KEY (#80BB)	Desactive la gravite
C	PASSE_MURAILLE_KEY (#80CB)	Mode passe-muraille
ESC	-	Redemarrer

## Annexe D : Formules cles

### Adresse VRAM d'une ligne Y

```
Si Y < 8:
    Adresse = #C000 + (Y * 2048)
Sinon:
    Adresse = #C000 + ((Y mod 8) * 2048) + ((Y / 8) * 80)
```

### Index tile vers adresse graphique

```
Adresse = BASE_TILESET (#14E8) + (index * 64)
```

### Frequence sonore PSG

```
Frequence = 125000 / Periode
```

## Annexe E : Boucle principale du jeu



Figure 12 : La boucle principale GAME\_LOOP (#7ABB)

## **Annexe F : Index des routines**

Adresse	Routine	Description
#68A8	START_POINT	Point d'entree du programme
#68BB	HELICO_LOSE_LIFE	Gestion perte de vie
#6971	EXECUTE_BUTTON_ACTION	Actions des interrupteurs
#6ABE	INIT_MUSIC	Initialisation musique
#6B09	UPDATE_SOUND_REGISTERS	Mise a jour PSG
#6B5D	UPDATE_MUSIC	Routine musique frame
#6B79	PLAY_NOISE_SOUND	Jouer effet sonore
#6BE4	DRAW_SPRITE_TILE	Afficher sprite/tile
#6D37	DECREMENT_TIMER	Decrementation timer
#6E06	DRAW_TILE	Afficher un tile
#6E23	DRAW_TILE_MAP	Afficher tilemap complete
#6F34	MOVE_SPRITE	Deplacer un sprite
#6F5B	ANIMATE_ENEMY	Animer un ennemi
#6FE1	ANIMATE_ALL_ENEMIES	Animer tous les ennemis
#7292	SCROLL_LINE_RIGHT	Scroll une ligne droite
#724E	NEXT_VIEW_TO_SCREEN_DOWN	Scroll ecran vers bas
#726B	NEXT_VIEW_TO_SCREEN_UP	Scroll ecran vers haut
#7360	SCROLL_LEFT	Scroll vers gauche
#738F	SCROLL_LINE_LEFT	Scroll une ligne gauche
#745D	SCROLL_RIGHT	Scroll vers droite
#7898	FIRE_NEW_SHOOT	Nouveau tir
#7ABB	GAME_LOOP	Boucle principale
#7C8D	TEST_SHOOT_COLLISION	Test collision tir
#7CB7	HELICO_EXPLOSION_SOUND	Son explosion
#7D89	GAME_START	Demarrage jeu

Adresse	Routine	Description
#7DA1	MAIN_LOOP	Boucle menu
#7DC2	WAIT_FIRE_BUTTON_PRESS	Attente bouton fire
#7E8A	TEST_AND_DRAW_HELICO	Test et affiche helico
#7F1C	DRAW_HELICO	Afficher helicoptere
#8064	BUILD_HELICO_START_LINE_ARRAY	Calcul table VRAM
#80AB	CHEAT_TIMER_KEY	Cheat timer
#80BB	GRAVITY_KEY	Cheat gravite
#80CB	PASSE_MURAILLE_KEY	Cheat collision

## EPILOGUE

---

Airwolf restera dans l'histoire comme un jeu rate. Un produit de son époque, victime de la pression commerciale et des contraintes techniques. Mais son exploration révèle quelque chose de plus profond : derrière chaque ligne de code, il y a des humains. Des développeurs qui ont travaillé dur, qui ont fait des compromis, qui ont parfois échoué.

Quarante ans plus tard, nous pouvons enfin comprendre ce qui s'est passé. Non pas pour condamner, mais pour apprendre. Et peut-être, grâce à Airwolf Reloaded, pour offrir à ce jeu maudit la fin qu'il méritait.

"Le code ne ment jamais. Il raconte l'histoire de ceux qui l'ont écrit."

---

## FIN

---

Livre basé sur l'analyse technique du code source et l'article "Airwolf Reloaded" par Alain Le Guirec.

Decembre 2025

---

## Index des Figures

1. Figure 1 : Organisation complète des 64 Ko de mémoire d'Airwolf
2. Figure 2 : Le concept de tilemap - une carte unique contenant tout le monde
3. Figure 3 : Les 27 couleurs hardware de l'Amstrad CPC
4. Figure 4 : Structure d'un tile - 16x16 pixels, 64 octets
5. Figure 5 : L'entrelacement de la VRAM - 8 blocs pour 200 lignes
6. Figure 6 : Le système complet de l'hélicoptère

7. Figure 7 : Le systeme de scrolling par tiles
8. Figure 8 : Detection de collision integree au dessin
9. Figure 9 : Structure de 24 octets definissant chaque ennemi
10. Figure 10 : Le circuit sonore PSG AY-3-8912
11. Figure 11 : La boucle infinie intentionnelle
12. Figure 12 : La boucle principale GAME\_LOOP (#7ABB)