AIRWOLF : AUTOPSIE D'UN JEU MAUDIT

Une plongée dans les entrailles du code d'un classique raté de l'Amstrad CPC

Figure 1 : 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.

TABLE DES MATIERES

Cinquième Partie : Les Collisions

Sixième Partie : Les Ennemis

Septième Partie : Le Son

Huitième Partie : Les Révélations

Neuvième Partie : La Renaissance

Dixième Partie : Réflexions

Annexes

Epilogue

PREMIERE PARTIE : LE CONTEXTE

Chapitre 1 : Un jeu, une légende

La série télévisée

Airwolf, c'est d'abord une série ameriçaine diffusee entre 1984 et 1987. Un hélicoptère de combat furtif, le plus avance au monde, est pilote par Stringfellow Hawke pour le compte d'une agence gouvernementale secrète. Action, suspense, et sûr-tout cet hélicoptère noir aux lignes élégantes : Airwolf devient une icone des annees 80.

L'adaptation vidéoludique

Comme toute licence populaire de l'époque, Airwolf est decline en jeu vidéo. 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 inçarne Stringfellow Hawke aux commandes de son hélicoptère dans un réseau de grottes labyrinthique. La mission : sauver des otages, détruire des cibles, et s'échapper avant l'écoulement du temps imparti.

Figure 2 : Le premier écran d'Airwolf sur Amstrad CPC - L'hélicoptère face aux premiers ennemis

La réputation

Rapidement, Airwolf acquiert une réputation particulière. Non pas celle d'un chef-d'oeuvre, mais celle d'un çauchemar. Les joueurs se heurtent à une difficulté extrême. La plupart n'arrivent pas à dépasser quatre écrans. Ceux qui persistent plus loin rencontrent systematiquement un plantage dans la zone finale.

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

Chapitre 2 : L'enquête commence

Une nuit d'insomnie

C'est lors d'une période de sante fragile que l'enquête debute. L'insomnie devenant insupportable, il fallait occuper l'esprit. Repondre à des questions absûrdes. Et celle-ci sûrgit : "A quoi ressemble le niveau 2 d'Airwolf ?"

Les outils du détéctive

L'émulateur WinAPE devient le laboratoire. Son debuggeur intégré permet de visualisér la mémoire en temps reel, de poser des breakpoints, de tracer l'exécution instruction par instruction. Le code assembleur Z80 se dévoile progressivement.

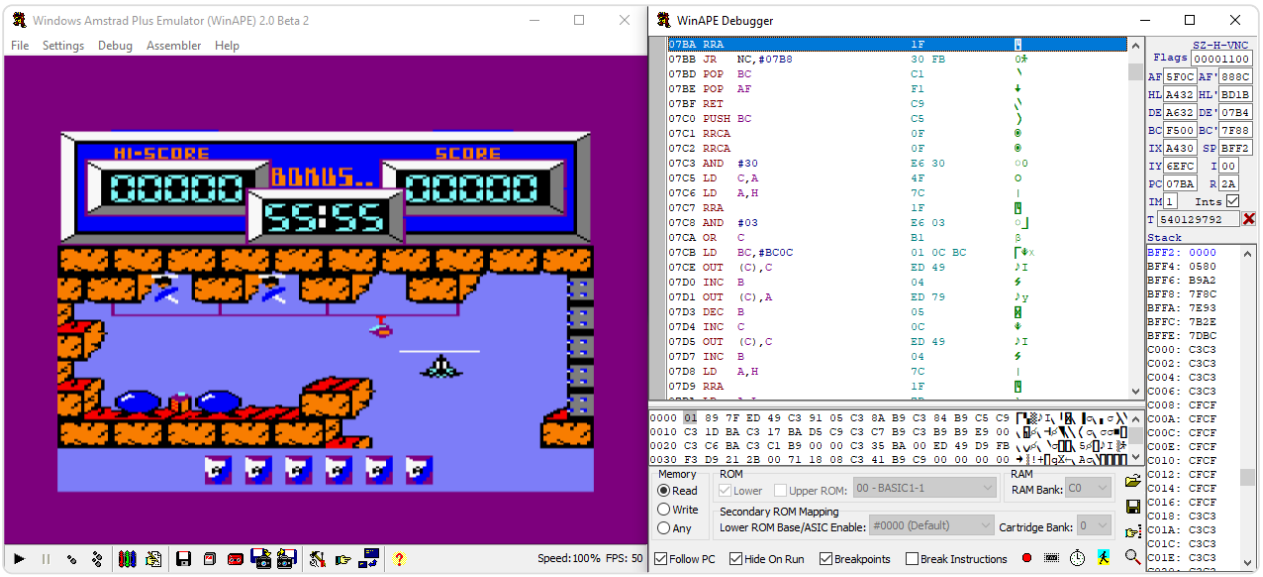

Figure 3 : L'émulateur WinAPE avec son debugger - A gauche le jeu en cours d'exécution, à droite le désassembléur montrant le code Z80, les registres, et le dump mémoire

Le debugger affiche en temps reel :
- Le code assembleur désassemblé avec les adresses (#07BA, #07BB...)
- Les registres du Z80 (AF, BC, DE, HL, IX, IY, PC, SP...)
- Les flags (Zero, Carry, etc.)
- Le dump hexadecimal de la mémoire
- La pile d'exécution (Stack)

C'est avec cet outil que chaque instruction sera tracée, chaque routine analysee.

La première découverte

Le "crash" de la zone finale n'est pas un bug. C'est une boucle infinie delibèree à l'adresse #698A :

```
    CP #08                  ; #6988 - Zone finale?
.infinite:
    JP Z,.infinite          ; #698A - Boucle étérnelle!
```

Quelqu'un à volontairement bloque le jeu dans `EXECUTE_BUTTON_ACTION` (#6971). Pourquoi ? La réponse se trouve plus profondement dans le code.

DEUXIEME PARTIE : L'ARCHITECTURE

Chapitre 3 : La cartographie mémoire

Organisation générale

L'Amstrad CPC dispose de 64 Ko de mémoire vive. Airwolf les utilisé avec une organisation révélatrice :

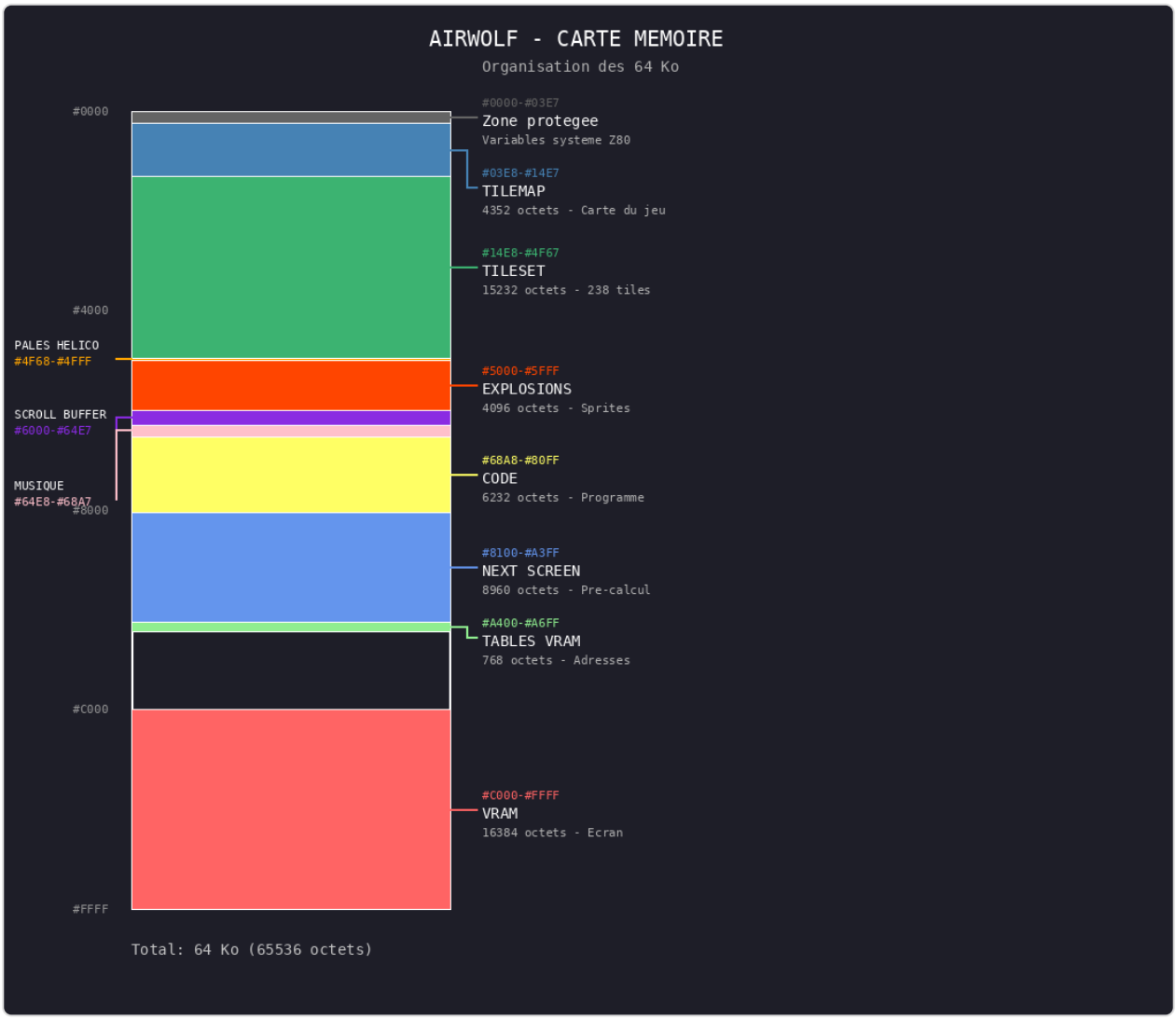

Figure 4 : Organisation complète des 64 Ko de mémoire d'Airwolf

```
Adresse     Taille    Contenu
--------------------------------
#0000-#03E7   1000    Zone protégée (variables système)
#03E8-#14E7   4352    TILEMAP - Carte du jeu
#14E8-#4F67  15232    TILESET - Les graphismes des tiles
#4F68-#4FFF    152    Sprites des pales d'hélicoptère
#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 écran suivant (pre-calcul)
#A400-#A6FF    768    Tables d'adresses VRAM
#C000-#FFFF  16384    Memoire vidéo
```

Le monde en une seule çarte

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

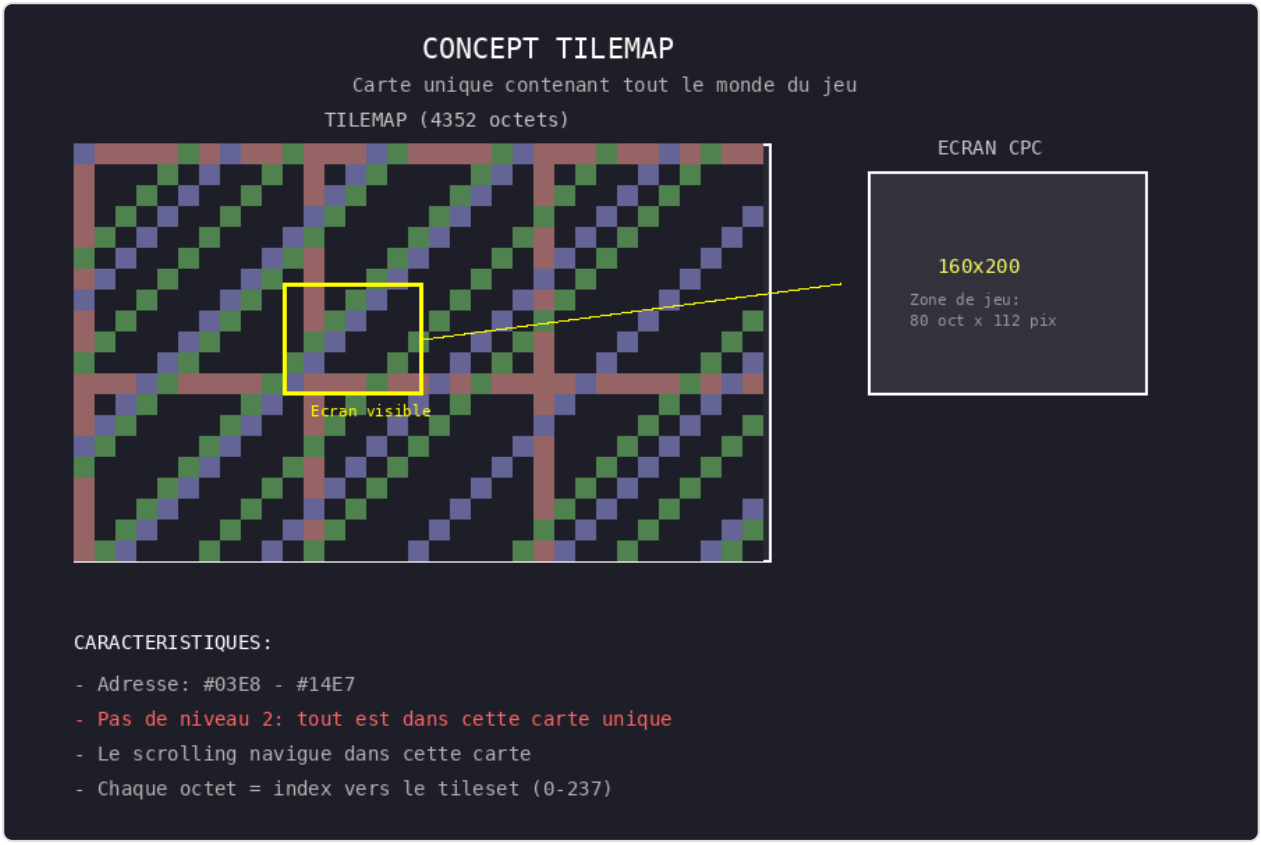

Figure 5 : Le concept de tilemap - une çarte unique contenant tout le monde

La tilemap définit l'agencement des tiles (blocs graphiques de 16x16 pixels) qui composent le décor. Chaque octet est un index vers le tileset, la bibliothèque de 238 tiles différents stockée à partir de #14E8.

Pas de chargement dynamique

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

La tilemap complète

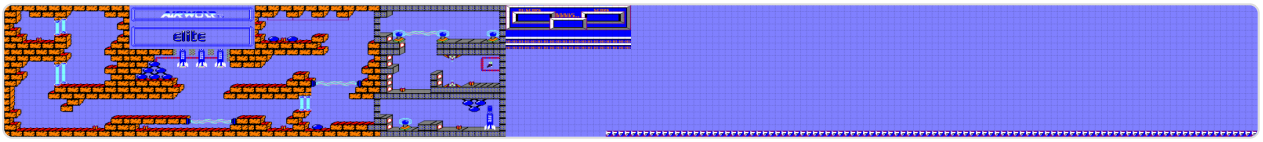

Figure 6 : La çarte complète du monde d'Airwolf - Tout le jeu tient dans cette unique tilemap

Cette vue d'ensemble révèle la structure du niveau :
- **A gauche** : L'entrée, les premiers écrans avec les obstacles de base
- **Au centre** : Le labyrinthe principal avec les ennemis
- **A droite** : Les zones avancees avec les boutons et la zone finale
- **Zone bleue** : L'espace navigable (ciel des çavernes)

Le joueur navigue dans cette çarte en scrollant de zone en zone. Les variables `VIEW_INDEX` (#6970), `VIEW_X` et `VIEW_Y` détérminent quelle portion de la tilemap est visible à l'écran.

Chapitre 4 : Le mode graphique

Mode 0 : le choix de la couleur

Airwolf utilisé 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 7 : Les 27 couleurs hardware de l'Amstrad CPC

La palette d'Airwolf

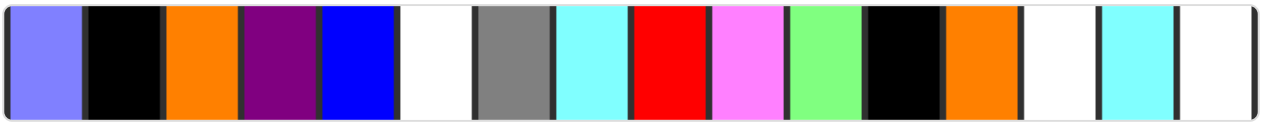

Figure 8 : La palette de 16 couleurs utilisée par Airwolf

La palette choisie pour Airwolf est révélatrice. On y trouve :
- **Bleu/violet** : Le fond (ciel des çavernes)
- **Orange** : Les briques des murs et certains ennemis
- **Cyan** : Les éléments interactifs (boutons, parois destructibles)
- **Rouge** : Les ennemis et éléments de danger
- **Noir** : Le fond neutre (pas de collision)

La presence de **deux nuances d'orange** tres proches est intrigante. Elles apparaissent dans les murs ET dans certains ennemis, suggerant que le développeur avait peut-être prevu de rendre les ennemis destructibles - une fonctionnalité jamais implementee.

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

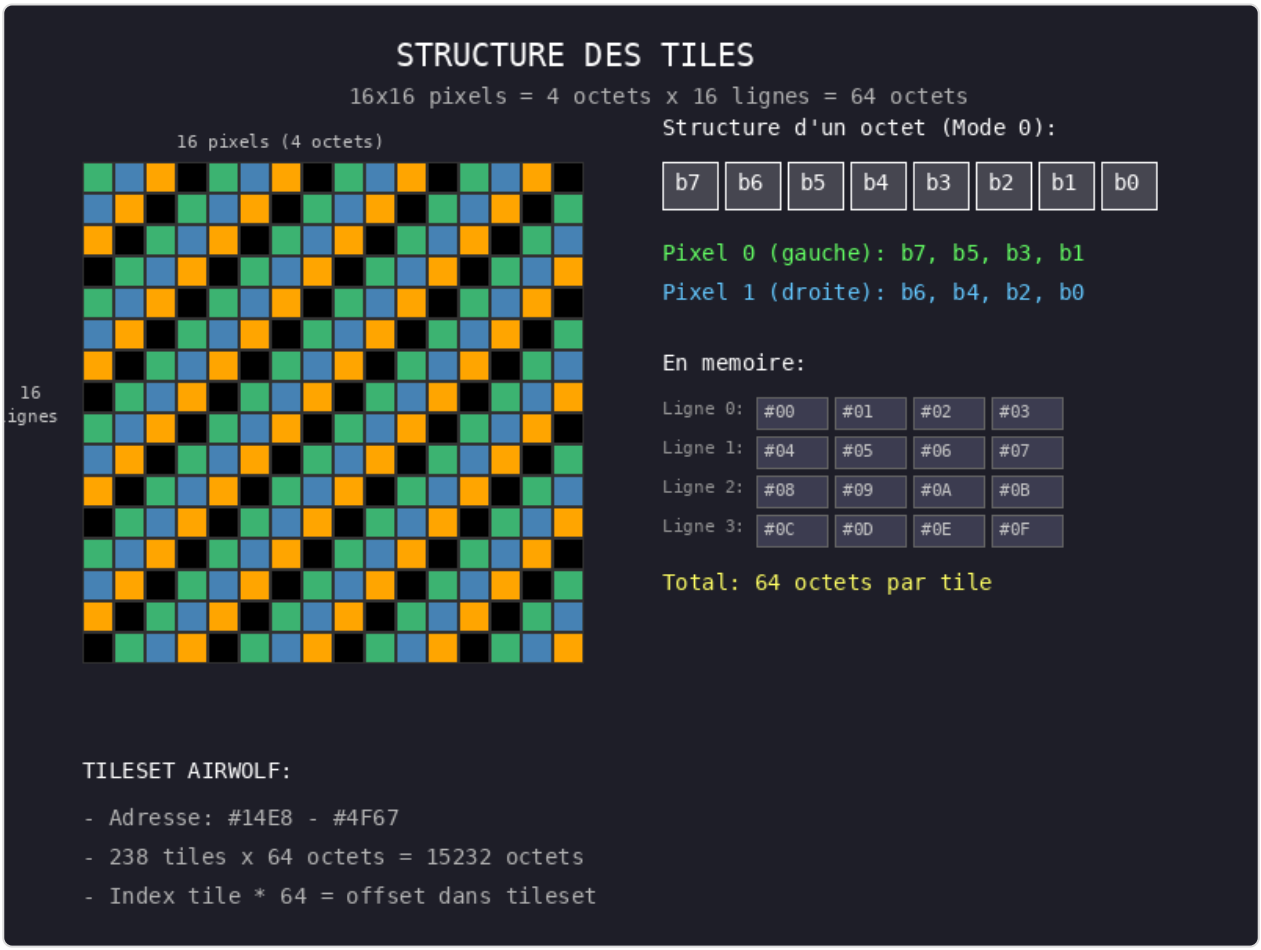

Figure 9 : Structure d'un tile - 16x16 pixels, 64 octets

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

Le tileset complet

Figure 10 : Les 238 tiles du jeu - Murs, plateformes, ennemis, décorations

Le tileset révèle la richesse graphique du jeu :
- **Première ligne** : Briques de base, variations de murs
- **Lignes 2-4** : Plateformes, coins, éléments de décor
- **Lignes 5-8** : Ennemis animes (robots, tourelles, flammes)
- **Lignes 9-12** : Boutons, portes, éléments interactifs
- **Dernière ligne** : Tiles speciales (fin de niveau, bonus)

On remarque la cohèrence graphique malgre la contrainte des 16 couleurs. Les tiles d'ennemis utilisént les mêmes orange que les murs - indice possible d'une fonctionnalité de destruction abandonee.

L'affichage d'un tile suit la logique de la mémoire vidéo entrelacée. La routine `DRAW_TILE` (#6E06) gère cette complexité :

```
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 mémoire vidéo entrelacée

Anatomie de la VRAM

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

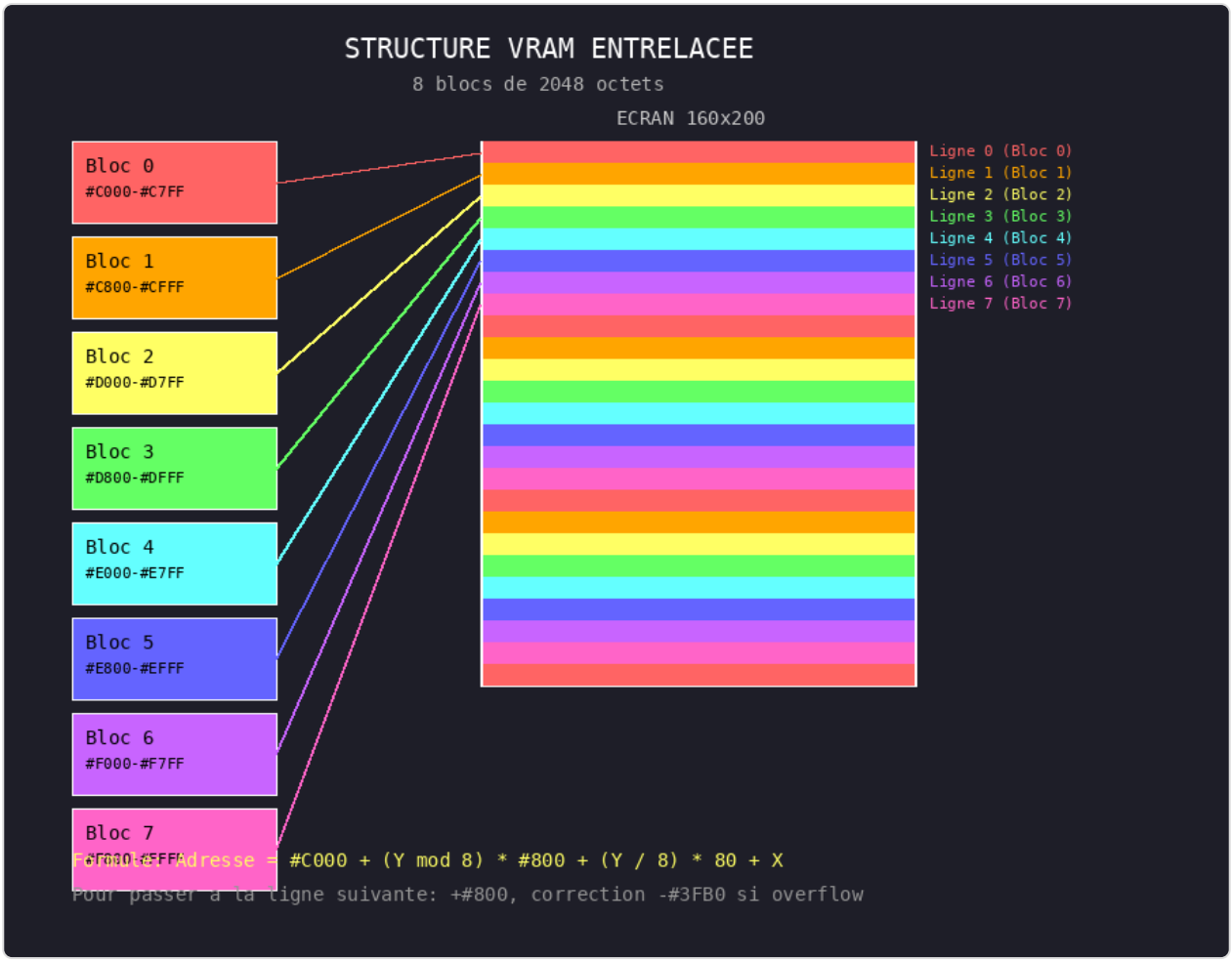

Figure 11 : L'entrelacément de la VRAM - 8 blocs pour 200 lignes

```
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 utilisé des tables pre-calculees. La routine `BUILD_HELICO_START_LINE_ARRAY` (#8064) remplit ces tables :

```
VRAM_LINE_START_ARRAY       EQU #A400  ; Pour l'hélicoptère
VRAM_LINE_START_ARRAY_OLD   EQU #A500  ; Position précédente
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'hélicoptère. Une optimisation cruciale pour les performances.

TROISIEME PARTIE : L'HELICOPTERE

Chapitre 6 : L'anatomie d'Airwolf

Figure 12 : Le système complet de l'hélicoptère - rotation, pales, variables

Les variables de l'hélicoptère

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

```
; Position
HELICO_X:               db ?    ; Coordonnée X (0-80 octets)
HELICO_Y:               db ?    ; Coordonnée 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 gravité
GRAVITY_DOWN_CPT:       db ?    ; #79D2 - Compteur gravité

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

Les dimensions

L'hélicoptère 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 système de rotation

Sept frames, trois directions

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

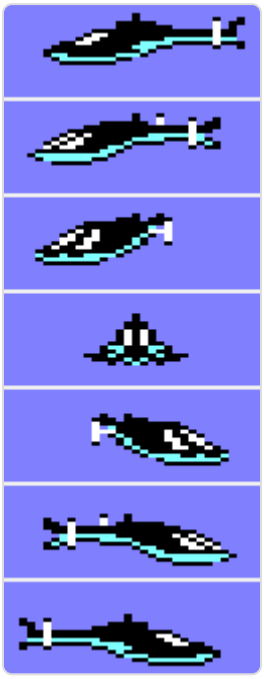

Figure 13 : Les 7 frames de rotation de l'hélicoptère - De droite (haut) à gauche (bas) en passant par la position frontale

```
Frame 0: Hélicoptère pointe à droite (===>)
Frame 1: Legère rotation vers le haut
Frame 2: Rotation intermediaire
Frame 3: Hélicoptère pointe vers le haut (===^)
Frame 4: Rotation vers la gauche
Frame 5: Rotation intermediaire
Frame 6: Hélicoptère pointe à gauche (<===)
```

Chaque frame est un sprite distinct stocké à partir de l'adresse #4F68 (BASE_PALLE_HELICO). Les sprites montrent non seulement la rotation du fuselage mais aussi l'angle du rotor principal et du rotor de queue.

L'algorithme de rotation

La rotation fonctionne par pas de 3 sur 7 frames. Elle est gèree 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 souhaitee (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                       ; Deja 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
    ; ... multipliçation 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 intermediaires, donnant une impression de mouvement fluide.

Chapitre 8 : Les pales du rotor

Animation independante

Les pales du rotor tournent independamment de l'orientation de l'hélicoptère. Six frames d'animation créént 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) stocké 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 à 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
    ; ... multipliçation par frame

    RET
```

Les pales sont dessinées séparément, au-dessus du corps de l'hélicoptère, permettant une superposition correcte avec le décor.

Chapitre 9 : La gravité

Un système d'acceleration

Airwolf simule une gravité réaliste avec acceleration. Quand le joueur ne donné pas d'input vertical, l'hélicoptère commence à tomber, de plus en plus vite. Ce système est gère 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 gravité

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

    ; Accelèrer
    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 à 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'hélicoptère commence à tomber lentement
- Le tempo diminue : 10, 9, 8, 7... la chute accelère
- Minimum atteint à 4 : vitesse terminale
- Un seul appui vers le haut : la gravité se reinitialisé

Cette meçanique créé une sensation de poids et de pilotage exigeant.

Chapitre 10 : Le système de tir

Trois directions de tir

L'hélicoptère peut tirer dans trois directions, correspondant à son orientation. La routine `FIRE_NEW_SHOOT` (#7898) gère 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 à la fois

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

    ; Position de départ selon direction
    ; Direction 0 (droite): devant l'hélicoptère
    ; Direction 1 (haut): au-dessus
    ; Direction 2 (gauche): à l'arrière

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

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

Détection des impacts

La routine `TEST_SHOOT_COLLISION` (#7C8D) détécte les collisions avec des pixels spe-ciaux :

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

    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 declenchent des actions : détruire une cloison via `EXECUTE_BUTTON_ACTION` (#6971), activer un interrupteur, ou simplement être absorbees par le décor.

QUATRIEME PARTIE : LE SCROLLING

Chapitre 11 : La philosophie du défilement

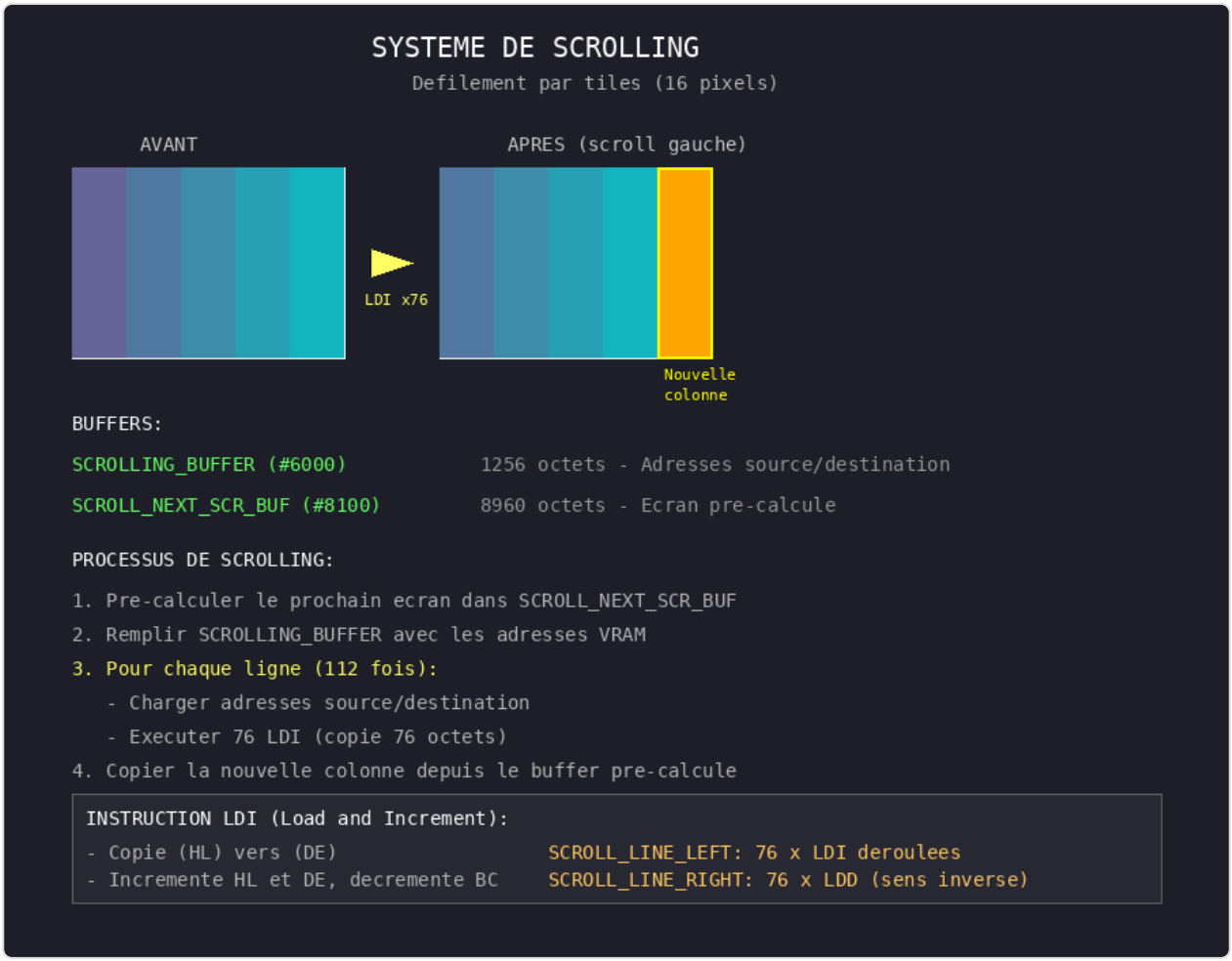

Figure 14 : Le système de scrolling par tiles

Un scrolling par tiles

Airwolf n'utilisé pas de scrolling pixel par pixel. Le monde defile par tiles entières de 16 pixels. Cette approche simplifie enormêment la gestion mais créé un effet sacçade caractéristique.

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'écran et n'est jamais scrolle.

Les quatre directions

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

Chapitre 12 : Le scrolling horizontal

Le buffer de préparation

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) gère le défilement 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'écran
    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 à 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 utilisé 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), incrémenté 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) utilisé 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) gère le scrolling vers le bas :

```
NEXT_VIEW_TO_SCREEN_DOWN:       ; #724E
    LD A,#0E                    ; 14 itérations (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'entrelacément

Le scrolling vertical est plus complexe çar il doit gèrer l'entrelacément de la VRAM. Copier "une ligne vers le bas" signifie en réalité :

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 çaractère suivant

Chapitre 14 : Le pre-calcul de l'écran suivant

Dessiner dans l'ombre

Avant de scroller, le jeu dessiné le prochain écran complet dans un buffer invisible. La routine `DRAW_SCROLLING_NEXT_SCREEN` se charge de cette préparation :

```
DRAW_SCROLLING_NEXT_SCREEN:
    LD HL,SCROLL_NEXT_SCR_BUF   ; #8100
    ; Pour chaque tile du prochain écran
    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 méthode

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 préparation et affichage

CINQUIEME PARTIE : LES COLLISIONS

Chapitre 15 : La détection intégrée

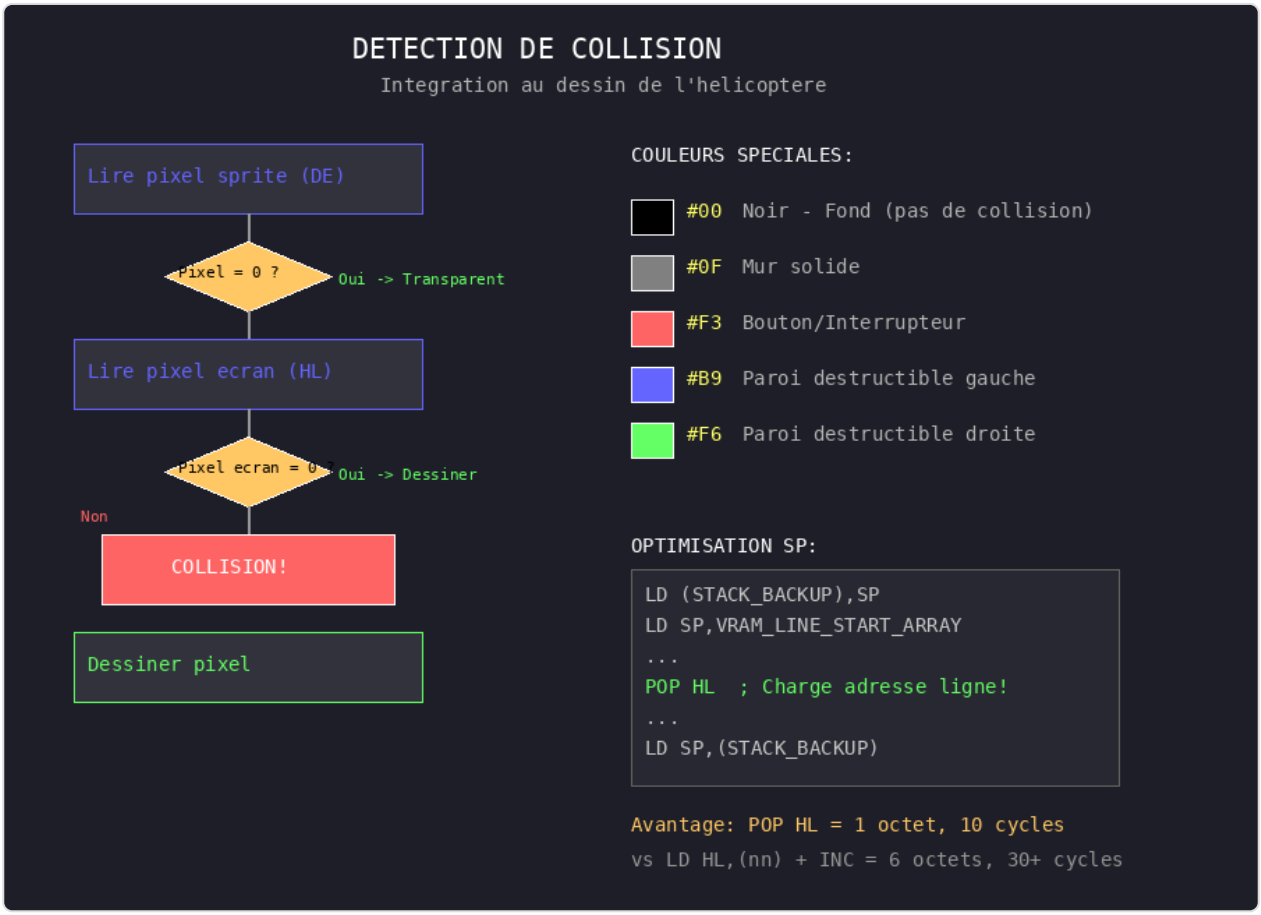

Figure 15 : Détéction de collision intégrée au dessin

Collision pendant le dessin

L'une des techniques les plus élégantes d'Airwolf : la détection des collisions est intégrée directement dans la routine d'affichage de l'hélicoptère `TEST_AND_DRAW_HELICO` (#7E8A).

Plutot que :
1. Dessiner l'hélicoptère
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 hélicoptère
    OR A
    JR Z,.transparent       ; Si 0, pixel transparent

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

    ; Pixel sprite non-nul ET pixel écran 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 méthode produit une détection au pixel pres. Si un seul pixel non-transparent de l'hélicoptère chevauche un pixel non-noir du décor, une collision est détectée et la routine `HELICO_LOSE_LIFE` (#68BB) est appelee.

Chapitre 16 : Les couleurs magiques

La palette comme système de collision

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

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

Le cyan : la couleur des secrets

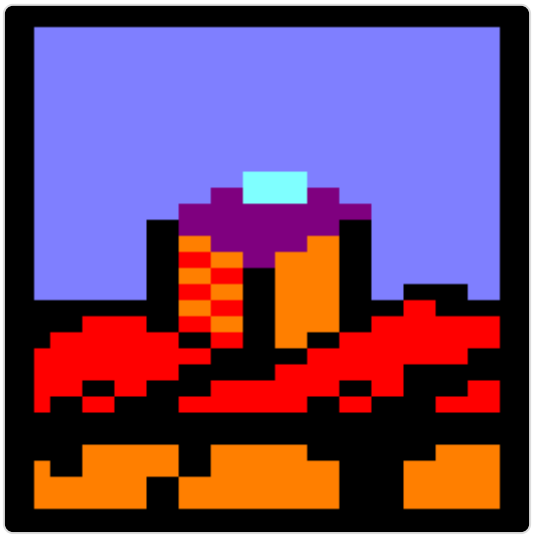

Figure 16 : Un bouton/interrupteur - Le bloc cyan en haut est détécte par le code de collision

Le **cyan** est la couleur cle du jeu. C'est elle qui détérmine ce qui peut être détruit ou active. Observez le bouton ci-dessus : le petit bloc cyan au sommet est le point de contact. Quand un tir touche un pixel cyan, la routine `TEST_SHOOT_COLLISION` (#7C8D) declenche l'action correspondante.

La destruction pixel par pixel

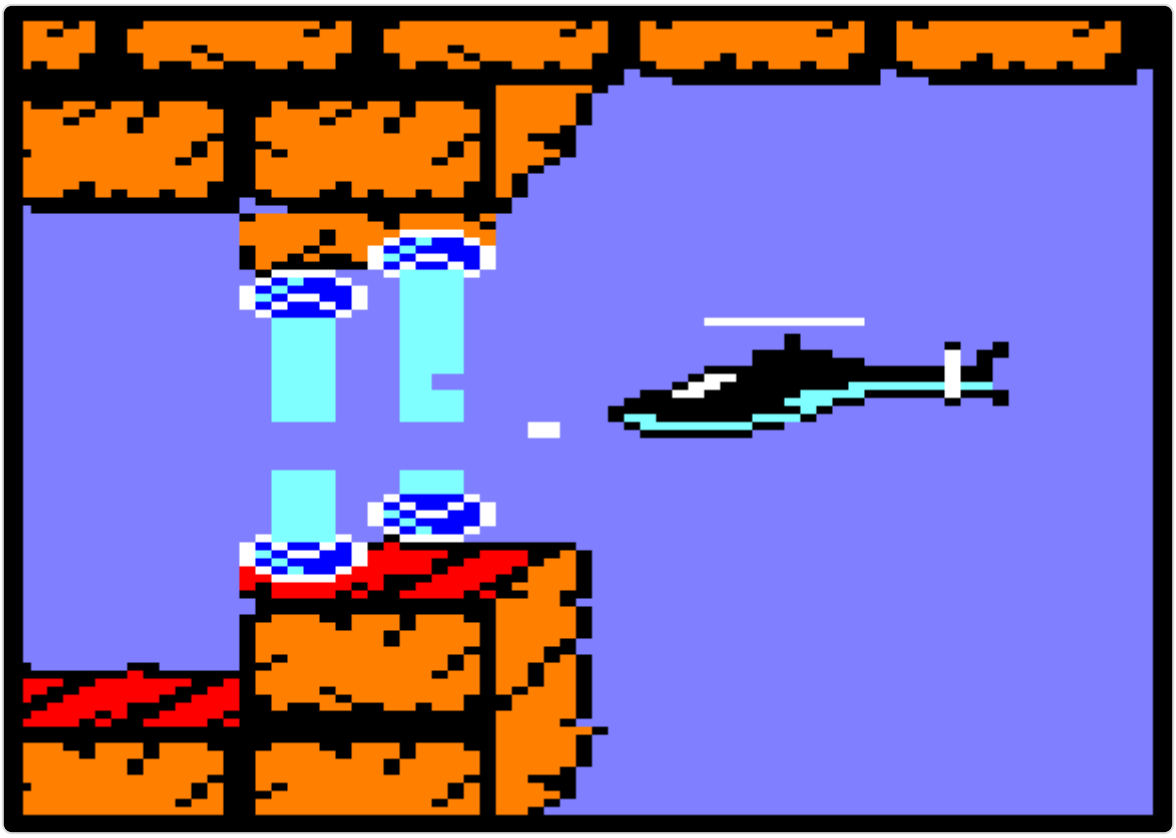

Figure 17 : Destruction des parois - Les colonnes cyan sont les points faibles des murs

Les parois destructibles contiennent des colonnes de pixels cyan. Chaque tir qui les touche effacé progressivement ces colonnes, creant l'illusion d'une destruction progressive. C'est elegant mais limité : seuls les pixels exactement de cette couleur peuvent être détruits.

Le mystère de l'orange

Fait troublant : les **deux nuances d'orange** de la palette apparaissent à la fois dans :
- Les briques des murs
- Certains sprites d'ennemis

Cette dupliçation suggère que le développeur avait envisage un système ou les tirs pourraient aussi détruire les ennemis en détéctant ces couleurs spécifiques. Cette fonctionnalité n'a jamais été implementee - les ennemis restent invulnerables.

Un design emergent

Ce système suggère une evolution du design pendant le développement. Les collisions par couleur sont simples à implementer mais rigides. Elles obligent les graphistes à peindre avec des couleurs spécifiques, creant une dependance forte entre art et code.

Chapitre 17 : L'optimisation par la pile

Le detournement du Stack Pointer

Pour afficher l'hélicoptère 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 ça marche

`POP HL` charge HL avec les 2 octets pointes par SP, puis incrémenté 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 dangèreuse : toute interruption pendant ce code corromprait la table d'adresses au lieu de la pile. Les interruptions doivent être désactivees ou le code doit être infaillible.

Chapitre 18 : L'effacement sécurisé

Ne jamais effacér ce qui ne nous appartient pas

Quand l'hélicoptère se déplacé, il faut effacér son ancienne position. Mais attention : on ne doit effacér que les pixels qu'on à dessinés, pas le décor !

```
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 effacér

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

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

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

SIXIEME PARTIE : LES ENNEMIS

Chapitre 19 : La structure des entités

Figure 18 : Structure de 24 octets définissant chaque ennemi

24 octets par ennemi

Chaque ennemi est défini par une structure de 24 octets, gèree 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

Offset	Taille	Role
#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 durée mouvement
#15-#16	2	Pointeur pattern courant
#17	1	Flag reset position

Le double système d'animation

Chaque ennemi à deux animations independantes :
1. **Animation graphique** : cycle des frames du sprite
2. **Animation de mouvement** : pattern de déplacément

Chapitre 20 : Les patterns de mouvement

Definition des patterns

Les mouvements sont définis comme des séquences de paires (direction, durée) :

```
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) gère 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 durée
    LD A,(IX+#14)
    DEC A
    LD (IX+#14),A
    RET NZ

    ; Passer à l'étape suivante du pattern
    LD L,(IX+#15)
    LD H,(IX+#16)
    INC HL
    INC HL                          ; Sauter direction+durée
    LD (IX+#15),L
    LD (IX+#16),H

    ; Charger nouvelle durée
    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

Independamment du mouvement, le sprite de l'ennemi change regulièrement :

```
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 à 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

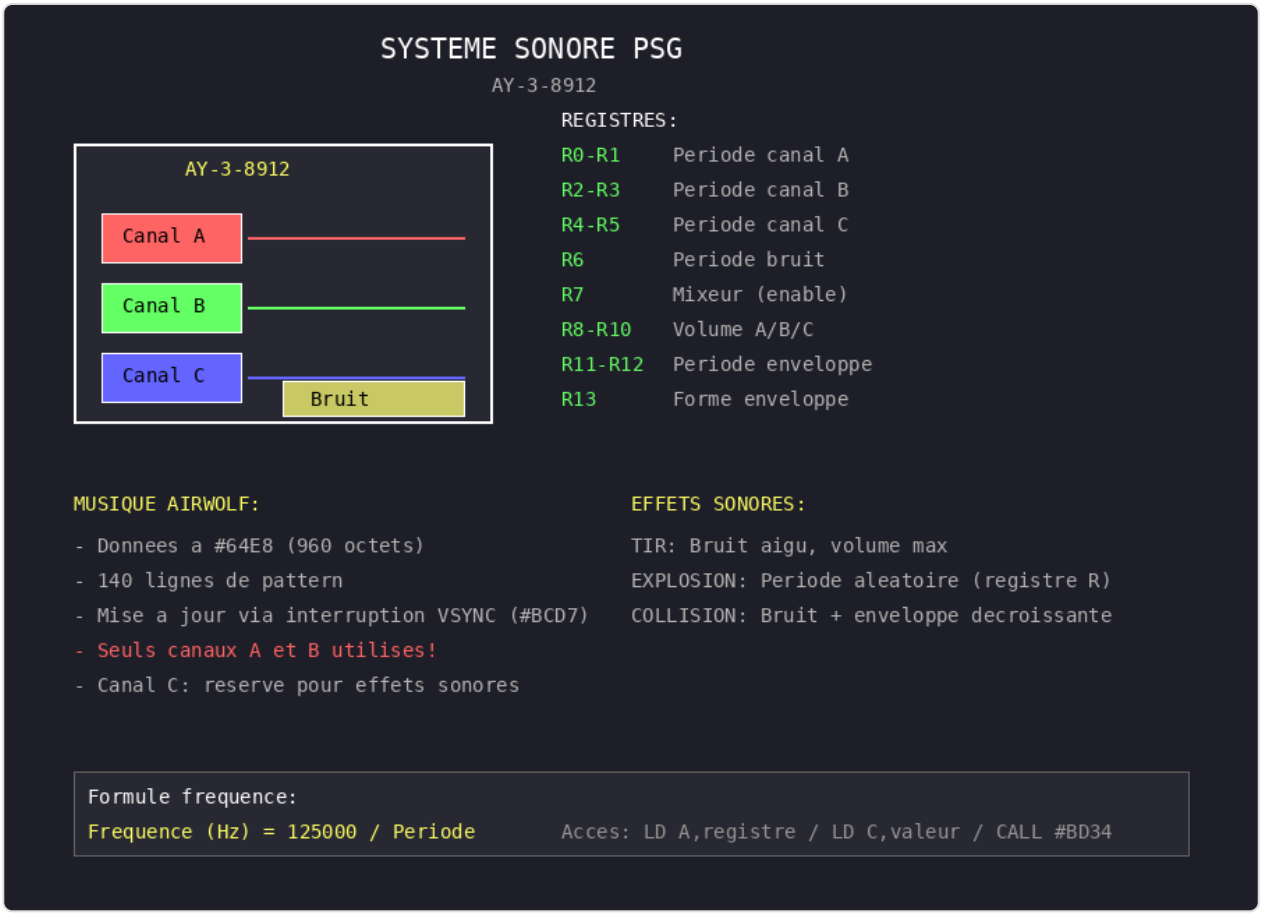

Figure 19 : Le circuit sonore PSG AY-3-8912

Architecture sonore

L'Amstrad CPC utilisé le circuit AY-3-8912, offrant :
- 3 çanaux mélodiques
- 1 generatéur de bruit partageable
- Enveloppes materielles

Les registres

Registre	Fonction
R0-R1	Periode çanal A
R2-R3	Periode çanal B

Registre	Fonction
R4-R5	Periode çanal C
R6	Periode bruit
R7	Mixeur (active/désactive çanaux)
R8-R10	Volume çanaux 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 données

La musique est stockée comme une séquence de valeurs de registres à partir de `MUSIC_DATA_BASE` (#64E8) :

```
MUSIC_DATA_BASE         EQU #64E8   ; Debut des données
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: période_A_low, période_A_high, période_B_low, période_B_high
; 0 = pas de changement
```

La routine de mise à jour

La routine `UPDATE_SOUND_REGISTERS` (#6B09) est appelee par `UPDATE_MUSIC` (#6B5D) à 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 période A low
    CALL #BD34
    INC HL

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

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

    ; Avancer le pointeur
    ; Gèrer le bouclage
    ; ...
```

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

Le çanal oublie

L'article original mentionne une découverte troublante : le player musical ignore complètement le troisième çanal sonore. Trois çanaux sont disponibles, mais seuls deux sont utilisés. Un indice supplementaire d'un développement precipite.

Chapitre 24 : Les effets sonores

Le bruit d'explosion

La routine `HELICO_EXPLOSION_SOUND` (#7CB7) utilisé 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 çanal 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) désactive les effets sonores après un certain temps.

Chapitre 24b : Les sprites d'explosion

L'animation de la mort

Quand l'hélicoptère entre en collision avec un obstacle, une animation d'explosion se declenche. Les sprites sont stockés à partir de l'adresse #5000 (BASE_EXPLOSION) :

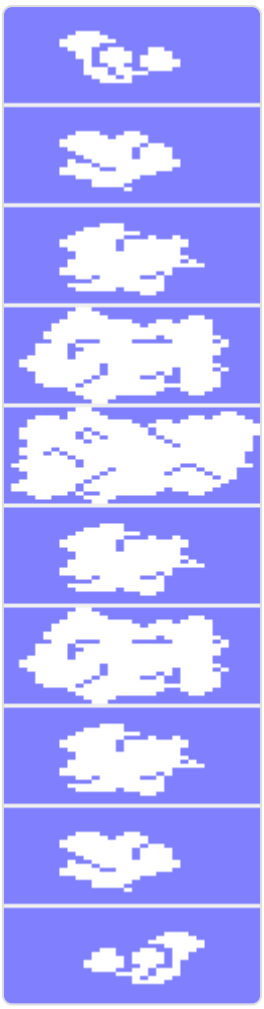

Figure 20 : Les frames d'animation de l'explosion - Du debut de l'impact à la dissipation

L'animation montre une progression réaliste :
1. Impact initial (petite tache)
2. Expansion rapide
3. Debris qui s'éparpillent
4. Fumee qui se dissipe

La routine `DRAW_EXPLOSION_SPRITE` (#79F2) gère l'affichage de ces frames en séquence, tandis que `HELICO_EXPLOSION_SOUND` (#7CB7) joue le son correspondant.

HUITIEME PARTIE : LES REVELATIONS

Chapitre 25 : Le niveau 2 n'existe pas

La vérité dans le code

L'exploration complète du code révèle la vérité : il n'y à pas de niveau 2. La tilemap unique contient l'integralite du monde jouable à 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 çarte fixe de tiles, pre-chargee en mémoire. Le joueur la parcourt, active des interrupteurs via `EXECUTE_BUTTON_ACTION` (#6971), détruit des cibles. Mais il n'y à pas de "suite".

Chapitre 26 : Le crash intentionnel

La zone 8 : l'écran maudit

L'écran final - Zone 8

Figure 21 : La zone 8 - L'écran que les joueurs n'atteignent jamais à çause de la boucle infinie

Voici l'écran fatidique ou `VIEW_INDEX` vaut 8. C'est une zone complexe avec plusieurs ennemis et obstacles. Mais si le joueur tente d'interagir avec un bouton ici, le jeu se figé indéfiniment.

Figure 22 : La boucle infinie intentionnelle

Les interrupteurs binaires

Le jeu utilisé un système d'interrupteurs lies à des écrans spécifiques 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

La boucle infinie se trouve dans `EXECUTE_BUTTON_ACTION` (#6971), la routine qui gère les interactions avec les interrupteurs du jeu. Le code vérifié `VIEW_INDEX` (#6970) - la position du joueur dans le monde :

```
EXECUTE_BUTTON_ACTION:              ; #6971
    LD A,(VIEW_INDEX)               ; #6971 - Charge la zone actuelle
    CP #01                          ; #6974 - Zone 1?
    JP Z,BUTTON_VIEW_1              ; #6976

    CP #03                          ; #6979 - Zone 3?
    JP Z,BUTTON_VIEW_3              ; #697B

    CP #09                          ; #697E - Zone 9?
    JP Z,BUTTON_VIEW_9              ; #6980

    CP #0B                          ; #6983 - Zone 11?
    JP Z,BUTTON_VIEW_11             ; #6985

    CP #08                          ; #6988 - Zone 8? (zone finale)
.infinite:
    JP Z,.infinite                  ; #698A - BOUCLE INFINIE!

    CP #0C                          ; #698D - Zone 12?
    JP Z,BUTTON_VIEW_12             ; #698F
    RET                             ; #6992
```

L'adresse **#698A** contient le tristement celebre `JP Z,.infinite` qui créé une boucle sans fin. Quand `VIEW_INDEX` vaut 8 (la zone finale), le flag Z est active et le processeur saute à sa propre adresse indéfiniment.

L'aveu d'échec

Cette boucle à l'adresse #698A est un aveu. Les développeurs savaient que la zone 8 ne pouvait pas être gère correctement. Plutot que de livrer un produit qui plante de manière visible, ils ont opte pour un gel silencieux que peu de joueurs atteindraient jamais.

Dans la version corrigee (Airwolf Reloaded), ces 3 octets (`CA 8A 69`) ont été remplaces par 3 NOPs (`00 00 00`), permettant enfin au code de continuer.

Chapitre 27 : Les vestiges du scrolling pixel

Du code mort

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

```
UNUSED_FULL_SCROLL_LEFT:        ; #7382
    ; Deçale l'écran d'un pixel
    ; Plus fluide mais plus lent
    CALL SCROLL_LEFT            ; #7360
    ; ...
```

L'hypothese du changement de design

Ces routines suggèrent un design initial diffèrent : un scrolling fluide, pixel par pixel, comme les meilleurs jeux de l'époque. Abandonnér ce système pour un scrolling par tiles indique soit des contraintes de performance, soit un changement d'équipe, soit les deux.

Chapitre 28 : Plusieurs mains

Les styles de code

L'analyse révélatrice montre des styles de programmation différents selon les sections :
- Certaines routines sont élégantes, optimisées (comme `DRAW_HELICO` #7F1C)
- D'autres sont verbeuses, repetitives
- Les conventions de nommage varient
- L'utilisation des registres est incohèrente

L'assemblage final

Airwolf n'est pas le travail d'un seul développeur. C'est un patchwork, assemble à 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 à inspire une question : à quoi ressemblerait Airwolf s'il avait été correctement concu ? La réponse prend la forme d'Airwolf Pico, une réimagination sur la plateforme PICO-8.

Les corrections apportees

- Vrais niveaux multiples
- Scrolling fluide
- Détéction de collision cohèrente
- Fin de jeu accèssible
- Equilibrage de la difficulté

Chapitre 30 : Airwolf Reloaded CPC

Retour aux sources

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

Les modifiçations

- Correction de la boucle infinie finale
- Logique d'interrupteurs repensee
- Nouveaux graphismes par Titan
- Jeu completable de bout en bout

Préservér l'esprit

L'objectif n'était pas de créer un nouveau jeu, mais de révèler celui qui aurait du exister. Les meçaniques originales sont préservées, seuls les bugs bloquants sont corriges.

DIXIEME PARTIE : REFLEXIONS

Chapitre 31 : Archéologie numérique

Ce que le code raconte

Au-delà des octets et des registres, le code source d'Airwolf raconte une histoire humaine :
- Des développeurs sous pression
- Des compromis forces
- Des ambitions revues à la baisse
- Une deadline impossible

Les indices du stress

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

Chapitre 32 : La valeur de la curiosité

Guerir par l'exploration

L'enquête sur Airwolf est nee de l'insomnie et de la maladie. Elle est devenue thérapeutique. L'exploration méthodique, la satisfaction des découvertes, le sentiment de comprendre : autant de remèdes inattendus.

La perseverance recompensee

Quarante ans après sa sortie, Airwolf livre enfin ses secrets. Il aura fallu de la patience, des outils modernes, et sûrtout une curiosité tenace pour percer ses mystères.

Chapitre 33 : Héritage technique

Ce qu'Airwolf nous enseigne

Malgre ses défauts, Airwolf contient des techniques remarquables :
- L'utilisation de SP comme pointeur rapide (`DRAW_HELICO` #7F1C)
- La détection de collision intégrée au rendu (`TEST_AND_DRAW_HELICO` #7E8A)
- Les tables pre-calculees pour les adresses VRAM (`BUILD_HELICO_START_LINE_ARRAY` #8064)
- Le système de patterns pour l'animation (`ANIMATE_ENEMY` #6F5B)

Ces techniques restent pertinentes pour tout développeur de jeux rétro 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 écran
```

Annexe B : Map mémoire complète

```
#0000-#003F : Variables système 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'entrée (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 vidéo (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 gravité
C	PASSE_MURAILLE_KEY (#80CB)	Mode passe-muraille
ESC	-	Redémarrer

Annexe D : Formules clés

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 23 : La boucle principale GAME_LOOP (#7ABB)

Annexe F : Index des routines

Adresse	Routine	Description
#68A8	START_POINT	Point d'entrée 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 à 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 complète
#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 écran vers bas
#726B	NEXT_VIEW_TO_SCREEN_UP	Scroll écran 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

Adresse	Routine	Description
#7CB7	HELICO_EXPLOSION_SOUND	Son explosion
#7D89	GAME_START	Demarrage jeu
#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 hélicoptère
#8064	BUILD_HELICO_START_LINE_ARRAY	Calcul table VRAM
#80AB	CHEAT_TIMER_KEY	Cheat timer
#80BB	GRAVITY_KEY	Cheat gravité
#80CB	PASSE_MURAILLE_KEY	Cheat collision

EPILOGUE

Airwolf restera dans l'histoire comme un jeu raté. 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 à des humains. Des développeurs qui ont travaille dur, qui ont fait des compromis, qui ont parfois echoue.

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

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

FIN

Livre base sur l'analyse technique du code source et l'article "Airwolf Reloaded" par Alain Le Guirec.
Decembre 2025

Index des Figures

A propos de l'auteur

Alain Le Guirec est un passionné de rétro-computing et de préservation du patrimoine vidéoludique. C'est lors d'une période de convalescence, durant des nuits d'insomnie, qu'il s'est lance dans le reverse-engineering complet d'Airwolf sur Amstrad CPC.

Ce qui à commence comme une simple question - "A quoi ressemble le niveau 2 d'Airwolf ?" - s'est transformé en une enquête de plusieurs semaines, plongeant dans des milliers de lignes de code assembleur Z80. Cette exploration à révèle non seulement les secrets techniques du jeu, mais aussi l'histoire humaine derrière son développement chaotique.

De cette analyse sont nes deux projets :
- **Airwolf Pico** : une réimagination du jeu sur la plateforme PICO-8
- **Airwolf Reloaded by Titan** : une version corrigee et amelioree pour Amstrad CPC

L'article original relatant cette aventure à été publie sur LinkedIn en decembre 2025.

A propos de ce livre

Ce livre à été redige avec l'assistance de **Claude**, une intelligence artificielle d'Anthropic. A partir du code source reverse-engineère par Alain Le Guirec et de son article LinkedIn, Claude à structure, developpe et mis en forme l'analyse technique complète du jeu.

Les diagrammes et schemas ont été genères programmatiquement en Python, tandis que les çaptures d'écran et sprites proviennent directement du jeu original analyse sous WinAPE.

Cette collaboration homme-machine illustre comment l'IA peut amplifier et diffuser le travail de préservation du patrimoine informatique, transformant des notes techniques en documentation accèssible à tous.

> "La curiosité, aussi apparemment futile soit-elle, peut devenir essentielle."
>
> — Alain Le Guirec