Disseny i construcció d'un motor 3D en temps real per la Gameboy Advance

Annexos

per Jordi Espada Brau

Enginyeria Tècnica en Informàtica de Sistemes, pla 93

Escola Politècnica Superior
Universitat de Girona

Tutor : Gustavo Patow
Departament : Departament d'Informàtica i Matemàtica Aplicada

Taula de continguts

ANNEX 0: ENSAMBLADOR ARM **6**

A0.1 CONJUNT DE REGISTRES DE LA CPU 6
A0.2 FLAGS DE CPU 8
A0.2.1 Registre d'Estat Actual de Programa(Current Program Status Register -CPSR-) *8*
A0.2.2 Registre salvaguardat del registre CPSR (Saved Program Status Registers -SPSR_<mode>-) *9*
A0.3 TIPUS DE CICLES 9
A0.4 INSTRUCCIONS ARM 10
A0.4.1 Instruccions de procés de dades *10*
- *Operacions de moure* *10*
- *Operacions lògiques* *10*
- *Operacions aritmètiques* *10*

A0.4.2 Intruccions de gestió de memòria *12*
- *Transferència ùnica* *12*
- *Transferència per blocs* *13*

A0.4.3 Salts, crides, mode CPSR i altres *14*
A0.5 PROGRAMACIÓ ENSAMBLADOR ARM, BASAT EN EL COMPILADOR GNU ASSEMBLER (GAS) 15
A0.5.1 Referència ràpida ensamblador ARM *15*
- Directives ensamblador GNU 15

A0.5.2 Crida d'una funció ensamblador des de C 17
A0.5.3 Utilització del procediment de crida estàndard (ATPCS) *18*

ANNEX 1: LA GAME BOY ADVANCE (GBA) **22**

A1.1 CARACTERÍSTIQUES 22
A1.2 LA CPU (ARM7TDMI) 23
A1.2.1 Registres de CPU *24*
- Registres R0-R12 (Registres de propòsit general) 24
- Registre R13 (SP) 24
- Registre R14 (LR) 24
- Registre R15 (PC) 25
- CPSR i SPSR (Registres d'estat de Programa) 25

A1.2.2 Modes de Processador *26*
A1.2.3 Estats de CPU *28*
A1.2.4 Tipus de CICLES *29*
A1.3. MEMÒRIA 30
A1.3.1 ROM del Sistema (System ROM): *31*
A1.3.2 Memoria Externa de Treball (EWRAM) *32*
A1.3.3 Memoria Interna de Treball (IWRAM) *32*
A1.3.4 IO RAM *32*
A1.3.5 Paleta RAM *33*
A1.3.6 Vídeo RAM *33*
A1.3.7 OAM *33*
A1.3.8 GAME PAK *34*
ROM *GAME PAK* 34
ROM *GAME PAK* IMATGE 1 34
ROM *GAME PAK* IMATGE 2: 34
CART RAM 34
A1.4 VISTA DELS GRÀFICS GBA PER HARDWARE 35
A1.4.1 Modes de vídeo *35*
A1.4.1.2 Modes basats en Rajoles (Modes 0,1 i 2) *35*
A1.4.1.2 Modes basats en bitmap (Modes 3,4 i 5) *37*
A1.4.2 Format de color *38*
A1.5 ACCÉS DIRECTE A MEMÒRIA (DMA) 39

ANNEX 3 EINES DEL DESENVOLUPAMENT 42

A2.1 KIT HAM 42
A2.1.1 El "makefile" 42
A2.1.2 Creació d'una ROM GBA 43
A2.1.3 Referència de funcions HAMLib 45
- Funcions d'inicialització HAMLib 45
- Funcions de control de fons 45
- Funcions de servei d'interrupció 47
- Funcions de manipulació de paleta 49
- Funcions mode mapa de bits 50
- Funcions mode rajola 52
- Funcions de finestra 55
- Controls de la GBA 56

A2.1.3 L'eina GFX2GBA 57
A2.1.3 Eines de temps 60
A2.2 L'EINA BIN2C V1.0 61
A2.3 L'EINA JGT V1.3 62
A2.3.1 Descripció 62
A2.3.2 Utilització 63
A2.3.3 El fitxer de configuració (GFX_Generate.cfg) 63
A2.3.4 Fitxers generats per JGT 63
A2.3.5 Carrega i gestió dels fitxers gràfics JGT 64

ANNEX 3: "MODE 7" A LA GBA 72

A3.0 OBJECTIU 72
A3.1 HISTÒRIA 72
A3.2 MODES RAJOLA A LA GBA: MODE 0,1 O 2 74
A3.3 REGISTRES DE ROTAT I/O ESCALAT 76
A3.4.1 REGISTRES DE PUNT DE REFERÈNCIA (REG_BGxX i REG_BgxY) 76
A3.4.2 REGISTRES DE ROTACIÓ/ESCALAT (BGxPA, BGxPB, BGxPC i BGxPD) 77
A3.4. ESTUDI DE LA RENDERITZACIÓ D'UN PLA 80
A3.4.1 La estructura del mapa 80
A3.4.2 El jugador 81
A3.4.3 Càlcul de components per la renderització d'un terra 83
Càlcul sense gir a l'espai 3D ($\alpha_j = 0º$) 84

Càlcul amb gir α_j a l'espai 3D 88
A3.4.4 Càlcul de components per la renderització d'un sostre 91
A3.4.5 Algorisme de renderitzat terra i sostre 92
A3.5 ADAPTACIÓ DEL "MODE 7" A LA GBA 93
A3.5.1 Rutina d'interrupció HBlank 93
Implementació de la rutina *HBlank* 93
Optimització de la rutina *HBlank* 95
A3.5.2 Rutina d'interrupció VBlank 98
A3.6 PREPARACIÓ DE UNA DEMOSTRACIÓ 99
A3.7 CAPTURES DE PANTALLA DE LA DEMOSTRACIÓ 102
A3.8 CONCLUSIONS 102
A3.9 TREBALLS FUTURS 103

ANNEX 4.1 ARITMÈTICA ENTERA 104

A4.1.1 ARITMÈTICA DE PUNT FIXA (APF) 104
A4.1.1 Definició de la APF 104
A4.1.2 Conversions APF 105
A4.1.3 Operacions aritmètiques en APF 105
- Operació aritmètica multiplicació 105
- Operació aritmètica divisió 105

A4.1.4 Enter de doble capacitat *106*
A4.1.5 Consells a l'hora d'utilitzar una APF *106*
A4.2 ALGORISME DE LA DIVISIÓ ENTERA AMB SIGNE 107
- La funció abs() 107
- La funció lmbd() 108

La funció subc() 109
- Operacions aritmètiques SHR, SHL, XOR i AND 109

ANNEX 5: FILTRE MIP-MAPPING 110

- Generació de MIP-MAPS de 8 bits 115
- Generació de MIP-MAPS de 16 bits 115
- Selecció NDD basat per l'àrea 117
- Selecció NDD basat en la compressió dels extrems 118
- Mostra-punt per-pixel MIP-MAPPING 121
- MIP-Mapping per-filtre basat en filtrat-Bilinear 121
- MIP-mapping per-pixel amb filtrat trilinear 122

ANNEX 6: TÈCNIQUES DE PALETA 124

A6.1 NEAREST COLOR 124
A6.2 REDUCCIÓ DEL COLOR A 8 BITS 127
A6.2 CREACIÓ D'UNA TAULA DE INTENSITATS 131
A6.3 CREACIÓ D'UNA TAULA DE TRANSPARÈNCIA 134

BIBLIOGRAFIA 136

Annex 0: Ensamblador ARM

A les seccions A0.1 a la A0.4, pretenem fer explicació dels registres i totes les instruccions que ens podem trobar dins una execució en mode ARM (*Advanced Risc Machines*), basat en el processador de la GameBoy Advance (GBA). A la secció A0.5, explicarem passos de declaracions, pas convencional de paràmetres, estructura d'un mòdul ensamblador ARM, etc; basat en el compilador GNUARM, que s'ha utilitzat pel desenvolupament d'aquest treball.

A0.1 Conjunt de registres de la CPU

La següent taula contempla els conjunt de registres del processador ARMTDMI disponibles als respectius modes.

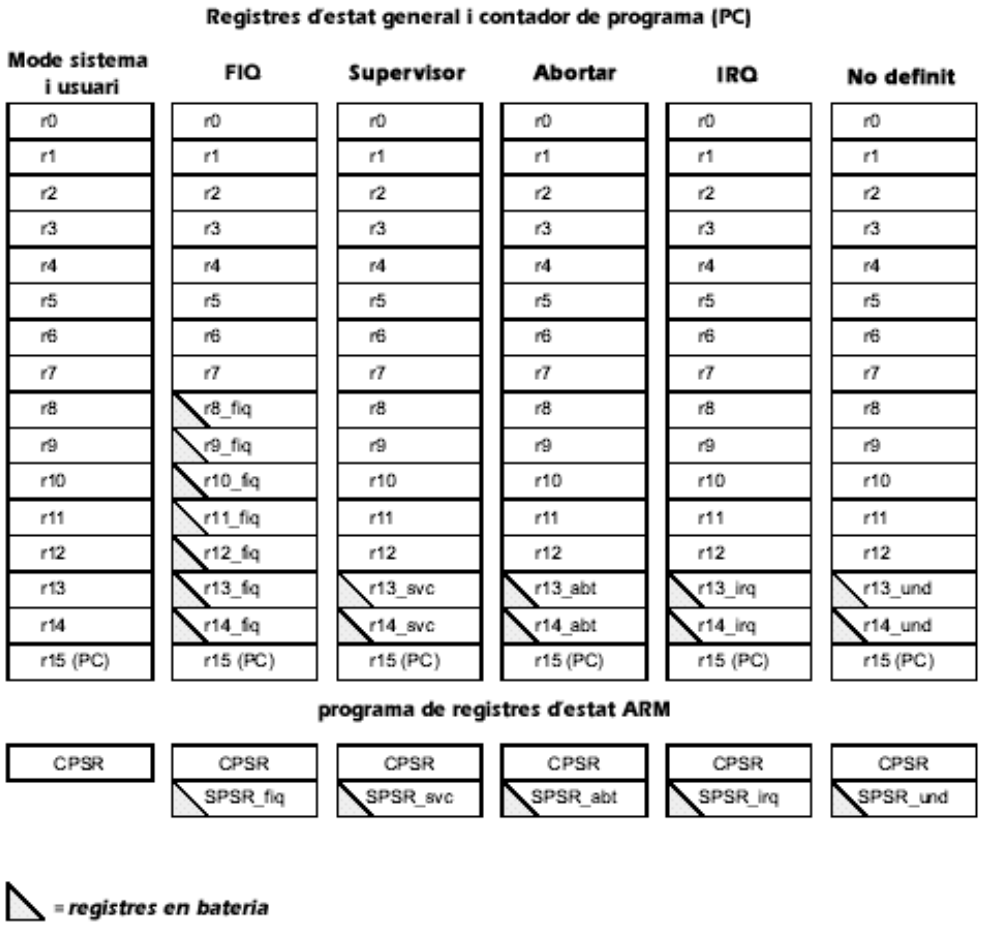


Figura A0.1 conjunt de registres del processador ARMTDMI

Si observem la taula, tenim un total de 37 registres de 32 bits cadascun: 31 registres generals (Rxx) i 6 registres d'estat (xPSR). Cal dir que, que alguns registres estan en bateria. Per exemple, cada mode té el seu propi registre R14 anomenat com: R14 en mode usuari, R14_fiq en mode interrupció, R14_svc en mode supervisor, etc. No obstant, hi ha registres que no estan en bateria. Per exemple, cada mode utilitza el mateix registre R0, per això si s'escriu el ell, també afectarà el seu contingut en els altres modes.

• Registres R0-R12 (Registres de propòsit general)

Aquests 13 registres són utilitzats per propòsit general en qualsevol circumstància. Cadascun té la mateixa funcionalitat i velocitat d'execució (*performance*), i per això, no hi ha un ràpid acumulador per operacions aritmètiques, i no hi ha un registre especial per adreçament de memòria.

D'altra banda, en mode THUMB només tenim 7 registres des de R0 a R7 que poden ser accedits lliurament, mentre que els 7 registres restants (R8-R12) poden ser accedits en algunes instruccions.

• Registre R13 (SP)

Aquest registre es utilitzat com punter de pila (*Stack Pointer* –SP-) en estat THUMB. En estat ARM, el registre R13 pot utilitzar-se com punter de pila o com un registre de propòsit general.

Com s'ha vist a la taula de la figura A0.1, el registre R13 està separat en cada mode. Quan el registre R13 s'utilitza com a punter pila cada controlador d'excepcions té d'utilitzar el seu propi punter de pila.

• Registre R14 (LR)

Aquest registre és utilitzat com ***registre d'enllaç*** (*Link Register* –LR-), quan es crida una subrutina amb la instrucció **embrancament amb enllaç** (*Branch with Link* –BL-). Aquest registre guarda l'adreça de retorn o valor de PC (comptador de programa), i per això només cal assignar el seu valor al PC quan vol retornar de la subrutina.

El guardat de l'adreça de retorn al registre LR és òbviament ràpid que empilar l'adreça de retorn a memòria, no obstant, només tenim un sol registre per cada mode, llavors l'usuari haurà de empilar manualment el seu contingut abans de fer noves crides de subrutines. El mateix passa quan succeeix una excepció, llavors hi ha una canvi de mode de processador i el valor de PC (R15) es salvat en LR.

Nota: En mode ARM, R14 pot ser utilitzat també com un registre de propòsit general, si dins una subrutina no és requerit el registre LR a l'hora del seu retorn.

• Registre R15 (PC)

R15 sempre s'utilitza com a comptador de programa (PC). Quan es vol llegir R15, normalment retorna el valor de PC+nn perquè les execucions d'instrucció està segmentades, on 'nn'depèn de la instrucció i l'estat de CPU (ARM o THUMB)

• CPSR i SPSR (Registres d'estat de Programa)

Els codis de condició actuals (flags) i el control de CPU són guardats en el registre CPSR. Quan succeeix una excepció, el vell CPSR és salvat a al registre SPSR del respectiu mode d'excepció (molt semblant quan el PC és guardat a LR).

A0.2 Flags de CPU

A0.2.1 Registre d'Estat Actual de Programa(Current Program Status Register -CPSR-)

A sota tenim presentat el registre CPSR i a continuació una explicació detallada dels bits que el componen.

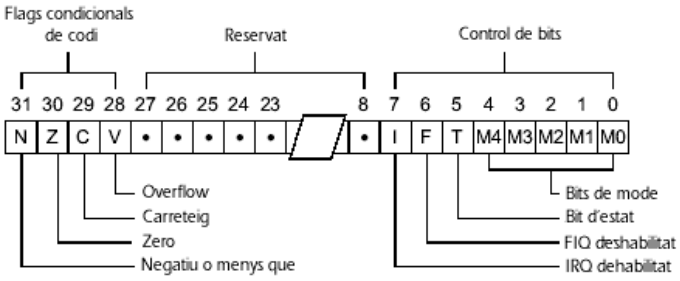


Figura A0.2 Paraula d'estat (CPSR)

Bit 31-28: Flags condicionals de codi (N,Z,C,V)

Aquests bits reflecteixen el resultat de les operacions aritmètiques o lògiques. En mode ARM, és pot decidir si es vol que una instrucció modifiqui els bits de condició o no afegint el sufix {S} al final d'una instrucció.
També, en mode ARM, totes les operacions poden ser executades condicionalment depenen del l'estat dels bits de condició, com per exemple MOVEQ (moure si és igual). Aquesta instrucció serà executada si el bit Z és igual a 1. En estat THUMB, només les instruccions de salt poden executar-se condicionalment.

Bit 27-8: Bits reservats

Aquests bits estan reservats per futures implementacions. Per futures compatibilitats, l'usuari mai hauria de canviar l'estat d'aquests bits.

Bit 7-0: Control Bits (I,F,T,M4-M0)

Aquests bits canviaran quan succeeixi una excepció. En mode privilegiats (modes no-usuari) aquests han de ser canviats manualment. Els bits d'interrupció I i F són utilitzats per deshabilitar les interrupcions IRQ i FIQ respectivament (posant un "1" significa deshabilitar).

El Bit T senyalitza l'estat actual de la CPU (0 = ARM, 1 = THUMB), aquest bit mai haurà de ser canviat manualment, l'intercanvi entre ambdós estats només pot ser efectuat mitjançant la instrucció *embrancament i canvi* (*Branch and exchange* -BX-)

Els bits de mode (M4-M0) expliquen l'actual mode operatiu.

```
   Binary Hex Dec  Expl.
 10000b 10h 16 - Usuari (no-privilegiat)
 10001b 11h 17 - FIQ
 10010b 12h 18 - IRQ
 10011b 13h 19 - Supervisor (SWI)
 10111b 17h 23 - Abortar
 11011b 1Bh 27 - No definit
 11111b 1Fh 31 - Sistema (mode privilegiat 'Usuari')
```

No és permès l'escriptura d'altres valors en els bits de mode.

A0.2.2 Registre salvaguardat del registre CPSR (Saved Program Status Registers -SPSR_<mode>-)

Addicionalment, per el registre CPSR, tenim 5 registres SPSR per cada mode operatiu: SPSR_fiq, SPSR_svc, SPSR_fiq, SPSR_irq i SPSR_und.

Quan la CPU entra en una excepció, el registre d'estat de programa (CPSR) serà copiat al respectiu registre SPSR_<mode>. Cal dir que, només hi ha un sol SPSR per cada mode, per això les excepcions d'igual consideració dins l'actual excepció seran permeses només si el controlador d'excepció guarda el contingut actual del SPSR a memòria.

Per exemple, per una excepció IRQ: s'entra en mode IRQ, i el registre CPSR és copiat al registre SPSR_irq. Si el controlador d'interrupció vol habilitar més interrupcions IRQ dins l'actual, llavors primer s'haurà d'empilar el valor SPSR_irq abans de permetre-ho.

A0.3 Tipus de CICLES

Els tipus de cicles que podem trobar durant l'execució d'operacions ARM, són tres: **Cicle no seqüencial (N), Cicle seqüencial (S) i cicle Intern (I)**.

N – Cicle no seqüencial

Sol·licitud d'un transferència des de/a una adreça la qual **NO** està relacionada a l'adreça utilitzada a l'anterior cicle. En la GBA s'anomena 1er accés. El temps d'execució de 1 cicle no seqüencial (1N), equival a un tic[1] de rellotge més un accés de estat d'espera no seqüencial.

S – Cicle seqüencial

Sol·licitud d'una transferència des de/a una adreça la qual és la mateixa l'adreça utilitzada al cicle anterior. Per això, per carregar/guardar blocs consecutius des de/a memòria de 16 o 32 bits (veure a0.4.2, transferència per blocs -STM/LDM –), el primer accés serà no seqüencial, però els pròxims seran seqüencials. Aquest tipus d'accés a la GBA s'anomena segon accés.
El temps d'execució per un cicle seqüencial (1S) és de 1 tic de rellotge més un accés de estat d'espera seqüencial.

I – Cicle intern

Passa quan la CPU està ocupada i, per tant, no es requereix cap sol·licitud de transferència a/des de memòria. El temps d'execució de 1 cicle intern (1I), equival a un tic[1] de rellotge (sense esperes d'accés).

.

[1]A la GBA, un tic equival a $59'59 \cdot 10^{-6}$ segons

A0.4 Instruccions ARM

La següent secció pretén fer una ràpida referència de les instruccions i significats de cada instrucció que té integrat el processador ARMT7DMI en mode ARM.

A0.4.1 Instruccions de procés de dades

- **Operacions de moure**

Instrucció	Cicles	Flags afectats	Explicació d'operació
MOV{cond}{S} Rd,Op2	1S+x+y	NZc-	Rd = Op2
MVN{cond}{S} Rd,Op2	1S+x+y	NZc-	Rd = NOT Op2

- **Operacions lògiques**

Instrucció	Cicles	Flags afectats	Explicació d'operació
AND{cond}{S} Rd,Rn,Op2	1S+x+y	NZc-	Rd = Rn AND Op2
TST{cond} Rn,Op2	1S+x	NZc-	Void = Rn AND Op2
EOR{cond}{S} Rd,Rn,Op2	1S+x+y	NZc-	Rd = Rn XOR Op2
TEQ{cond} Rn,Op2	1S+x	NZc-	Void = Rn XOR Op2
ORR{cond}{S} Rd,Rn,Op2	1S+x+y	NZc-	Rd = Rn OR Op2
BIC{cond}{S} Rd,Rn,Op2	1S+x+y	NZc-	Rd = Rn AND NOT Op2

- **Operacions aritmètiques**

Instrucció	Cicles	Flags afectats	Explicació d'operació
ADD{cond}{S} Rd,Rn,Op2	1S+x+y	NZCV	Rd = Rn+Op2
ADC{cond}{S} Rd,Rn,Op2	1S+x+y	NZCV	Rd = Rn+Op2+Cy
SUB{cond}{S} Rd,Rn,Op2	1S+x+y	NZCV	Rd = Rn-Op2
SBC{cond}{S} Rd,Rn,Op2	1S+x+y	NZCV	Rd = Rn-Op2+Cy-1
RSB{cond}{S} Rd,Rn,Op2	1S+x+y	NZCV	Rd = Op2-Rn
RSC{cond}{S} Rd,Rn,Op2	1S+x+y	NZCV	Rd = Op2-Rn+Cy-1
CMP{cond} Rn,Op2	1S+x	NZCV	Void = Rn-Op2
CMN{cond} Rn,Op2	1S+x	NZCV	Void = Rn+Op2
MUL{cond}{S} Rd,Rm,Rs	1S+mI	NZx-	Rd = Rm*Rs
MLA{cond}{S} Rd,Rm,Rs,Rn	1S+mI+1I	NZx-	Rd = Rm*Rs+Rn
UMULL{cond}{S} RdLo,RdHi,Rm,Rs	1S+mI+1I	NZx-	RdHiLo = Rm*Rs
UMLAL{cond}{S} RdLo,RdHi,Rm,Rs	1S+mI+2I	NZx-	RdHiLo = Rm*Rs+RdHiLo
SMULL{cond}{S} RdLo,RdHi,Rm,Rs	1S+mI+1I	NZx-	RdHiLo = Rm*Rs
SMLAL{cond}{S} RdLo,RdHi,Rm,Rs	1S+mI+2I	NZx-	RdHiLo = Rm*Rs+RdHiLo

A l'aparat de cicles:

Sumar *x=1I* cicles si Op2 és un registre que efectua un previ desplaçament aritmètic.
Sumar *y=1S+1N* cicles si Rd=R15.
En les multiplicacions, *m* depèn de com els bits més significants de Rs siguin 0s o 1s.

- **Camp {S}**

En mode ARM, quasi totes les instruccions de procés de la informació poden actualitzar els bits de condició al registre CPSR de manera opcional, afegint el sufix {S}.

- **Camp de condició ({cond})**

En mode ARM, totes les instruccions poden ser executades depenen de l'estat dels bits de condició (C,N,Z,V) al registre CPSR. Amb els següents sufixos de condició {cond}, es pot executar les instruccions de forma condicional, com per exemple B*EQ* (Salta *si igual*) o MOV*MI* (Moure *si té signe*)

{cond}	Flags	Significat
EQ	*Z=1*	*Igual*
NE	*Z=0*	*No igual*
CS	*C=1*	*Menys o igual (sense signe)*
CC	*C=0*	*Menys que (sense signe)*
MI	*N=1*	*Negatiu*
PL	*N=0*	*Positiu o zero*
VS	*V=1*	*Overflow*
VC	*V=0*	*No overflow*
HI	*C=1 i Z=0*	*Més gran (sense signe)*
LS	*C=0 o Z=1*	*Menys o igual que (sense signe)*
GE	*N=V*	*Més gran o igual que*
LT	*N<>V*	*Més petit que*
GT	*Z=0 i N=V*	*Més gran que*
LE	*Z=1 o N<>V*	*Menys o igual que*
AL	-	*Executar sempre*
Reservat, no s'utilitza	-	-

- **Operant 2 (Op2)**

L'operant 2 descrit en les taules d'instruccions lògiques i algunes d'aritmètiques, ha de ser un registre o immediat amb possibilitat de ser desplaçat amb les operacions LSR (Desplaçament $\log_2$ a la dreta), LSL(Desplaçament $\log_2$ a l'esquerra), ASR (Desplaçament Aritmètic a la Dreta), ROR (Rotació de registre a la Dreta).

Exemple,

```
MOV        r0,r1, LSL #2     @ Equival a fer r0 ← r1 << 2
```

A0.4.2 Intruccions de gestió de memòria

• Transferència ùnica

Instrucció	Cicles	Flags afectats	Explicació d'operació
LDR{cond}{B}{T} Rd,<Address>	1S+1N+1I +y	----	Rd=[Rn+/-<offset>]
LDR{cond}H Rd,<Address>	1S+1N+1I +y	----	Càrrega de mitja paraula sense signe
LDR{cond}SB Rd,<Address>	1S+1N+1I +y	----	Càrrega d'un byte amb signe
LDR{cond}SH Rd,<Address>	1S+1N+1I +y	----	Càrrega de mitja de paraula amb signe
STR{cond}{B}{T} Rd,<Address>	2N	----	[Rn+/-<offset>]=Rd
STR{cond}H Rd,<Address>	2N	----	Guardat de mitja paraula
SWP{cond}{B} Rd,Rm,[Rn]	1S+2N+1I	----	Rd=[Rn], [Rn]=Rm

Sumar y=1S+1N si Rd=PC.

- **Camp {B}**

  S'afegeix el camp {B} si és vol escriure/llegir a/des de memòria una informació de 8 bits (1 byte).

- **Camp {T}**

  S'afegeix el camp {T} si és vol tenir un accés a memòria en mode no privilegiat.

- **El valor d'adreça (<Address>)**

El valor d'adreça es pot donar de maneres diferents:

1. Un valor immediat i en hexadecimal que representa l'adreça.
2. Si el registre Rn conté un valor de adreça a llegir o escriure:
2.1 ***[Rn]*** : és realitza la lectura o escriptura a l'adreça que apunta el registre Rn i amb offset 0.

Especificació d'adreçament pre-indexada:

2.2 ***[Rn,<#{+/-}offset>]{!}*** : és realitza la lectura o escriptura a l'adreça que apunta el registre Rn més/menys ({+/-}) un offset immediat. Si s'afegeix el sufix {!}, s'actualitzarà el registre Rn amb aquest offset al haver acabat la instrucció.

2.3 ***[Rn,{+/-}Rm{,<shift>}]{!}*** : és realitza la lectura o escriptura a l'adreça que apunta el registre Rn més/menys ({+/-}) l'offset contingut al registre Rm. El registre Rm pot ser desplaçat per un immediat (<shift>). De igual manera, si s'afegeix el sufix {!}, s'actualitzarà el registre Rn amb aquest offset al haver acabat la instrucció.

Especificació d'adreçament post-indexada:

2.4 ***[Rn],<#{+/-}offset>*** : és realitza la lectura o escriptura a l'adreça que apunta el registre *Rn* respecte un offset i al acabar, actualitza el valor Rn pel l'offset especificat.

2.5 ***[Rn],{+/-}Rm{,<shift>}*** : és realitza la lectura o escriptura a l'adreça que apunta el registre Rn respecte l'offset contingut al registre Rm. El registre Rm pot ser desplaçat per un immediat. Al acabar, actualitza el valor Rn pel l'offset Rm especificat.

• Transferència per blocs

Instrucció	Cicles	Flags afectats	Explicació d'operació
LDM{cond}{amod} Rn{!},<Rlist>{^}	nS+1N+1I +y	----	Carrega multiple sense signe
STM{cond}{amod} Rn{!},<Rlist>{^}	(n-1)S+2N	----	Guardat Multiple

Sumar y=1S+1N si *Rd* és en llista (<Rlist>).

• Llista de registres <Rlist>

En instruccions de transferència per blocs, s'assigna una llista de registres consecutius continguts entre "{" i "}". LDM i STM realitzen carregues o escriptures de paraules (32 bits) a/des de paraules de memòria, als registres assignats.

Exemple:

```
LDM   R0,{R1,R2,R3}   @ Carrega el valor de tres paraules als registres de la
                      @ llista (R1 a R3) consecutives a l'adreça apuntada per R0.
```

O també:

```
LDM    R0,{R1-R3}
```

• Camp {amod}

En operacions per transferència per blocs, podem actualitzar el registre adreça base (*Rn*) incrementant-lo o decrementar-lo igual al numero de lectures o escriptures a/des de memòria als registres dins la llista (<*Rlist*>). Això és possible amb els sufixos IB, IA, DB i DA.

```
IB  :     Increment Before  (Incrementa abans)
IA  :     Increment After   (Incrementa després)
DB  :     Decrement Before  (Decrementa abans)
DA  :     Deecrement After  (Decrementa després)
```

Alternativament, FD, ED, FA, EA poden ser utilitzats, molt comunament per simplificar neumònics a l'hora de fer transferències de pila.

```
ED  :     Empty stack, Descending  (Pila buida, descendeix)
FD  :     Full stack, Descending   (Pila plena, descendeix)
EA  :     Empty stack, Ascending   (Pila buida, ascendeix)
FA  :     Full stack, Ascending    (Pila plena, ascendeix)
```

• Camp {!}

Actualitza el registre Rn amb el numero de escriptures/lectures realitzades.

A0.4.3 Salts, crides, mode CPSR i altres

Instrucció	Cicles	Flags afectats	Explicació d'operació
B{cond} label	2S+1N	----	PC=$+8+/-32M
BL{cond} label	2S+1N	----	PC=$+8+/-32M, LR=$+4
BX{cond} Rn	2S+1N	----	PC=Rn, THUMB/ARM (Rn bit0)
MRS{cond} Rd,Psr	1S	----	Rd=Psr
MSR{cond} Psr{_field},Op	1S	----	Psr[field]=Op
SWI{cond} Imm24bit	2S+1N	----	PC=8, ARM Svc mode, LR=$+4
NOP	1S	----	R0=R0

- **label (etiqueta)**

  Per instruccions de tipus B (*Branch*), etiqueta la qual es vol saltar. Si B salta de forma incondicional (sense res al camp {*cond*}), llavors es suposarà que salta a una subrutina.
  Per subrutines, normalment es fa servir la instrucció BL (*Branch & Link*), que guarda l'adreça de retorn al registre R14. Per retornar de la subrutina, només cal moure el registre R14 al registre de R15 (registre PC).

- **camp {_field}**

  Especifica una màscara de bits de la paraula PSR destí que s'han de protegir. Per exemple, CPSR_fsxc (aka CPSR aka CPSR_all) deshabilita totes les proteccions.

- **Imm24bit**

  Quan s'invoca una interrupció software (SWI) desde estat ARM, s'ha d'especificar SWI NN*10000h, on NN indica el numero d'interrupció.

A0.5 Programació ensamblador ARM, basat en el compilador GNU Assembler (GAS)

A0.5.1 Referència ràpida ensamblador ARM

Un sumari de comandes útils i expressions per l'arquitectura utilitzant l'ensamblador GNU es presenta breument en la porció concloent d'aquesta secció. Cada línia ensamblador té el següent format,

```
[<etiqueta>:[intrucció o directiva>] @ comentari
```

A diferencia del ensamblador ARM, l'ensamblador GNU no requereix indentar instruccions i directives. Les etiquetes són reconegudes per *colon* següent en ves de cada posició al començament de la línia. Un exemple es veu a continuació, un simple programa que defineix una suma que retorna la suma del dos argument (veure a0.5.3 per més informació sobre el pas de paràmetres ensamblador) de entrada:

```
.section .text, "x"

add:
        ADD r0, r0, r1  @ Suma els arguments de entrada (r0,r1) a r0
        MOV pc, lr      @ retorn de la subrutina

@ fi de programa
```

Llistat a0.1

- ## Directives ensamblador GNU

Seguidament es mostra per ordre alfabètic un llistat del les comandes directives ensamblador GNU més usuals.

GNU Assembler Directive	Description
`.ascii "<cadena>"`	Insereix una cadena com informació dins el mòdul ensamblador.
`.asciz "<cadena>"`	Com `.ascii`, però acaba la informació inicialització seguit d'un zero.
`.baling <potencia_de_dos> {,<omplenat_de_valors>{,<max_omplenat>}}`	Alinea l'adreça a `<potencia_de_2>`. L'ensamblador alinea per l'afegit de bytes del valor `<omplenat_de_valors>` o un apropiat defecte. L'alineació no podrà ser possible més de `<max_omplenat>` bytes.
`.byte <byte1> {,<byte2>} ...`	Insereix una llista de valors de 8 bits com informació dins del ensamblador.
`.code <numero_de_bits>`	Indica que el tamany de instrucció a partir d'aquesta directiva serà del tamany `<number_of_bits>`. Aquest valor a de ser 16 per ensamblador Thumb i 32 per ensamblador ARM.
`.else`	Cal utilitzar-se amb combinació les directives `.if` i `.endif` (similar a `else` de C)
`.endif`	Indica el final d'un bloc de codi condicional (veure `.if`, `.ifdef` i `.ifndef`)
`.endm`	Finalitza una definició de macro (veure `.macro`)
`.endr`	Finalitza un bucle de repetició (veure `.rept` i `.irp`)

.equ <nom simbol>, <valor>	Aquesta directiva defineix un valor en un símbol amb nom, equivalent a la directiva #define de C.
.err	Causa un error durant el compilat.
.global <symbol>	Aquesta directiva declara un símbol perquè pugui ser cridat per un mòdul extern.
.hword <short1> {,<short2>}...	Insereix una llista de valors de 16 bits com informació dins el mòdul ensamblador.
.if <logical_expression>	Fa un bloc de codi condicional. Per finalitza el bloc cal utilitzar la directiva .endif. Veure també la directiva .else
.ifdef <symbol>	Inclou un bloc de codi si <simbol> es definit. Cal finalitzar el bloc amb la directiva .endif
.include "<filename>"	Inclou el codi font indicat. (equivalent al #include de C.)
.irp <param> {,<val_1>}{,<val_2>}...	Repeteix un bloc de codi, un cop per cada valor en la llista de valors. Cal marcar el final de bloc amb la directiva .endr. En el bloc de codi repetit, cal utilitzar \<param> per substituir el valor associat en la llista de valors.

.macro <name> {<arg_1>}{,<arg_2>}...{,<arg_N}	Defineix una macro anomenada <name> amb N paràmetres. La macro definida ha de acabar amb la directiva .endm. Per sortir de la macro més aviat que .endm, cal utilitzar la directiva .exitm. Per llegir els paràmetres calen ser precedides per "\". Per exemple:
	
	.macro SHIFLEFT a, b
	. if \b < 0
	MOV \a,\a, ARL #-\b
	.exitm
	.endif
	
	MOV \a, \a, LSL #\b
	
	.endm

.rept <numero_de_cops>	Repeteix un bloc de codi donat el numero de cops. Cal acabar el bloc amb la directiva .endr.
<nom_registre> .req <nom_registre>	Aquesta directiva anomena un registre.
.section <nom_seccio> {,"<flags>"}	Comença un nou codi o secció de dades. Les seccions en GNU s'anomenen .text, una secció de codi, .data, una secció de dades inicialitzada, i .bss, una secció de dades sense inicialitzar. Aquestes seccions tenen uns flags per defecte, i el linkador entendrà aquests noms per defecte. A continuació s'expliquen els flags permesos a la directiva .section, per fitxers en format ELF: <flag> significat a secció permesa w secció escribible x secció executable
.set <nom_variable>, {valor_variable}	Aquesta directiva posa els valors s'una variable.
.space <numero_de_bytes> {,<omplenat_bytes>}	Reserva el numero de bytes donats. Els bytes son omplenats amb 0s o per <omplenat de bytes> si s'especifica.
.word <word1>{,<word2>} ...	Insereix una llista de valors de 32 bits com informació.

A0.5.2 Crida d'una funció ensamblador des de C

Per cridar a la funció *add* vist al llistat a0.1, caldria declarar el símbol *add* com extern,

```
.section .text, "x"

.global add @ defineix la etiqueta add de linkat extern

add:
        ADD r0, r0, r1  @ Suma els arguments de entrada (r0, r1) a r0
        MOV pc, lr      @ retorn de la subrutina

@ fi de programa
```

I en el mòdul C, declarar el prototipus de funció *add* extern,

```
extern int add(int num1, int num2);

void main(void)
{
        int r;

        r = add(2,2);
}
```

A0.5.3 Utilització del procediment de crida estàndard (ATPCS)

Segons les normes de procediment estandart de crides de rutina de ensamblador ARM (*Procedure Call Standard* -ATPCS-), els 4 primers registres (r0-r3) es fan servir pel pas de paràmetres o per el retorn de valors. Si n'hi ha més de 4 parametres de entrada, cal fer el seu pas de la resta de parametres per mitjà de la pila (sp) que, en ATPCS, s'especifica un *Full stack, Descending*.

Existeixen unes normes que cal complir a l'hora de implementar una funció ensamblador ARM o THUMB, que especifiquen la manera de fer el pas de paràmetres, la utilització de cada registre.

- **Normes registres entre rutines**

Per complir amb el ATPCS cal seguir les següents normes entre registres:

• Cal utilitzar els registers r0-r3 per fer el pas de valor de paràmetres en les rutines, i també es fan servir per fer el retorn d'un resultat si la rutina representa una funció. Entre crides de subrutines pots utilitzar els registres r0-r3 per qualsevol propòsit, sense necessitat de ser salvaguardats abans de ser sobrescrits.

• Els registres r4-r12 mantenen valors de variable locals de rutina. En mode thumb, moltes instruccions només poden fer servir els registres r4-r7 com a variables locals. A diferencia dels registres r0-r3, aquests han de ser salvaguardats si són sobrescrits durant l'execució de la rutina.

• Registre r13, és el punter de pila (stack pointer –sp-). No es pot utilitzar sota cap propòsit. El valor mantingut en sp a la sortida de la rutina ha de ser el mateix valor que quan va entrar.

• El registre r14 es el registre que conté l'adreça de tornada (link register –lr-). Si salves l'adreça de tornada, llavors es pot utilitzar el registre r14, per altres propòsits entre crides.

• Registre r15 es el comptador de programa (program counter –pc-). Aquest registre, no pot ser sobrescrit sota cap propòsit.

• Pas de paràmetres

Existeix dos tipus de pas de parametres, segons ATPCS: rutines variàdiques i no variàdiques.

Una rutina amb un número variable de arguments se'n diu que és *variadic*. Una rutina amb un número fix de arguments és *no variadica*. Les regles són diferents en el passat de paràmetres de rutines variadica i no variadica. Aquesta secció descriurem la base estàndard de pas de paràmetres enters.

- Rutines variadiques

Els valors de parametres són passat a rutines variadiques en registres enters a1-a2, i a la pila si es necessari (a1-a4 és sinónim dels registres r0-r3 respectivament).

L'ordre de les paraules utilitzades es tal si els valors de paràmetre que s'han guardat consecutivament en tamany de paraula (32 bits) i llavors s'han transferit a:

1. a1-a4, a1 primer.
2. La pila, l'adreça més baixa primer. (Això significa que els arguments estaran empilats a la pila en ordre invers[1])

- Rutines no variadiques

Els valors de paràmetre a nivell de maquina són passats com una rutina no variadica si:

1. Els primers 4 enters són assignats a a1-a4.
2. Els valors restants són empilats a la pila en ordre invers.

[1] En ATPCS, especifica *Full stack, Descending*.

- ## Valor de retorn

Una acció no retorna valor, una funció retorna:

• Un valor enter a traves de a1.
• De dos a 4 valors enters en a1-a2, a1-a3 or a1-a4.
• Un valor enter llarg (64 bits) ha de ser retornat indirectament, per memòria.

- ## Exemple pràctic d'una crida a funció ensamblador ARM amb més de 4 paràmetres

En vista les normatives implicades en ATPCS, una funció ARM:

1. Disposa de 16 registres, 2 dels quals són intocables (sp i pc).
2. Els altres, els registres r0 al r3 poden ser sobrescrits sense cap problema, però els demés (del r4 al r12) han de ser salvaguardats mitjançant la pila.
3. Pel pas de paràmetres, s'utilitzen els registres r0-r3 pels 4 primers. A partir del 5é paràmetre es passa per la pila.

Tot seguit mostrem un exemple pràctic i senzill per una rutina ensamblador ARM i que fa la suma de 6 paràmetres de entrada. Tot el pas de paràmetres es farà per mitja un punter de estructura C. Per això, només necessitarem un sol registre (r0) en el pas de paràmetres i no caldrà fer la carrega dels demés paràmetres per pila (sp).

```
typedef struct{
        int num1;
        int num2;
        int num3;
        int num4;
        int num5;
        int num6;
}tParametresADD;


extern int add(tParametresADD *);

void main(void)
{
        tParametresADD PADD;
        int Resultat;

        PADD.num1 = 0;
        PADD.num2 = 1;
        PADD.num3 = 2;
        PADD.num4 = 3;
        PADD.num5 = 4;
        PADD.num6 = 5;

        Resultat = Add(&PADD); { Resultat =0+1+2+3+4+5;}

}
```

I la funció ensamblador carregaria els 6 paràmetres com,

```
.section .text, "x"
.code    32

.global add                     @ defineix la etiqueta add de linkat extern

add:
@
        STMFD   sp!,{r4-r5}     @ Guarda els registres ja que seràn sobrescrits. A la vegada
                                @ actualitza sp (!) en Full stack, Descending, segons el
                                @ el convingut ATPCS.

        LDMIA   r0,{r0-r5}      @ Carrega els parametres als registres r0 al r5. R0 conté
                                @ el punter de l'estructura PADD declarada en el mòdul main.
                                @ Les carregues consecutives dels registres r0-r5 són de
                                @ tamany enter (32), per això, els camps de l'estructura
                                @ tParametresADD són enters.


        ADD r0, r0, r1
        ADD r0, r0, r2
        ADD r0, r0, r3
        ADD r0, r0, r4
        ADD r0, r0, r5


        LDMFD   sp!,{r4-r5}     @ Restaura els registres r4-r5. Ara sp, té el mateix valor
                                @ que quan va entrar a la rutina.
        MOV     pc, lr          @ retorn de la subrutina

@ fi de programa
```

Annex 1: La Game Boy Advance (GBA)

La GBA és una consola portàtil destinada a l'execució de videojocs de la casa Nintendo. Actualment hi ha tres versions d'aquesta consola: Game Boy Advance, Game Boy Advance SP i Game Boy Advance Micro (figura A1.1). La GBA Micro és la més reduïda de les tres, però, realment, la part més important que fa aquest sistema portable, deixant de banda el pes i les dimensions reduïdes, són els components de baix consum que té integrats.

Figura A1.1 *De esquerra a dreta Game Boy Advance, Game Boy Advance SP i Game Boy Advance Micro*

Comencem doncs, en explicar les característiques més importants d'aquest sistema[1], detallant-les més endavant.

A1.1 Característiques

Un resum de característiques més importants basat en la consola GBA són les següents,

CPU	ARM7TDMI 32 bit RISC per GBA o Z80/8080 8bit per Game Boy o Game Boy Color
Memòria	32 Kbyte + 96 Kbyte Video RAM (VRAM) interna a la CPU i 256 Kbyte de memòria externa a la CPU.
Pantalla	TFT Color LCD reflectiva (40.8 mm x 61.2 mm)
Resolució	240 x 160 pixels
Energia	Amb 2 bateries alkalinas AA o un pack especial de bateria.
Durabilitat	2 AA= 15 hores pack especial de bateria = 10 hores
Dimensions	82mm x 144.5 x 24.5mm
Pes	140 gr (aproximats)
Capacitat	fins a 256 Mbits (32 Mbytes)
Colors	Fins a 32,768 colores (16 bits)

Figura A1.2 *Característiques més importants de la GBA*

[1] Les informacions i referències tècniques que s'explicaran, estan basades sobre la consola GBA (figura A1.1 esquerra). Aquestes informacions s'han tret de [ARM01], [GT03],[TPP01A], [TPP01B] i [CVHS02].

La part més important a destacar en aquesta consola és que té funcions especials per controlar tots els dispositius del hardware GBA. Algunes d'aquestes funcions són per gràfics i acceleren moltes tasques 2D alliberant la CPU de fer importants operacions en 2D.

La manera d'accedir a aquestes funcions és a través de la memòria entrada/sortida mapejada (memory-mapped Input-Output). La Memòria IO mapejada és una manera de comunicar-se amb el hardware escrivint i/o llegint valors a l'adreça especifica. Aquesta adreça està mapejada a les funcions del Hard intern.

Una segona manera és a través de les funcions BIOS, les quals estan incloses a la memòria interna ROM de la GBA. Utilitzant interrupcions per Soft (SoftWare Interrupts –**SWI-**) és possible accedir a rutines preprogramades permanents dins el sistema ROM. Aquestes rutines accedeixen al hardware via memòria IO mapejada.

Els programes que s'executen sobre la plataforma GBA normalment estan continguts en un *Game Pak*. Un *Game Pak* consisteix principalment en un cartutx estàndard i reemplaçable a la GBA. Aquest cartutx, conté una regió de memòria ROM i, opcionalment, una regió de memòria RAM (en la forma de SRAM, Flash ROM, o EEPROM, utilitzades principalment per guardar informació dins el joc).

Dins la ROM del *Game Pak*, s'hi guarda el codi d'execució i dades del joc.

A1.2 LA CPU (ARM7TDMI)

La GBA disposa d'un processador RISC (Reduced Instruction Set Computer) ARM7TDMI a 16.78 MHz, dissenyat per ARM (Advanced RISC Machines), i pensat per tenir un alt rendiment d'execució (*performance*) i baix consum. Aquesta CPU disposa de fins a 16 registres quan es treballa en mode usuari.

ARMTDMI és un processador de 32 bits quan actua en un estat denominat *ARM*. Tanmateix, pot ser canviat a l'estat *THUMB*, el qual permet controlar un especial subconjunt d'instruccions de 16 bits que mapegen parts de les instruccions de 32 bits. El cicle d'execució de cada instrucció es caracteritza per una ràpida descodificació i execució segmentada en tres etapes:

a. Busqueda de la instrucció a memòria (fetch): .
b. Descodificació d'instrucció (Decoder).
c. Execució d'instrucció (Execution).

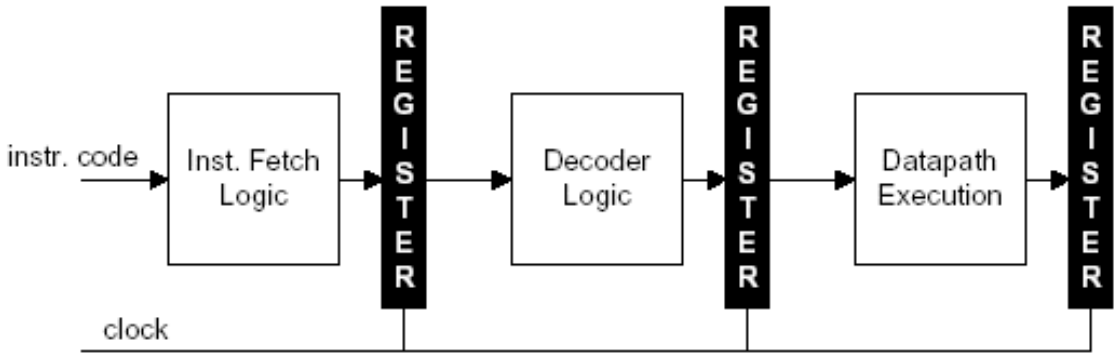


Figura A1.3 ***I****nstrucció segmentada en l'execució*

Aquesta segmentació té 3 avantatges:

- Guardar els resultats de cada etapa en registres.
- La segmentació permet que una instrucció sigui executada mentre que la següent instrucció sigui descodificada i la pròxima instrucció sigui obtinguda de memòria (fetch), tot això, a la mateixa vegada.
- Com a conseqüència, el període de rellotge (clock) resulta més curt que sense la segmentació.

Donat que una instrucció necessita 3 etapes abans ser executada, el comptador de programa (PC) sempre apunta dos instruccions per endavant de la instrucció actualment executant-se.

A1.2.1 Registres de CPU

16 registres de 32 bits són visibles en mode usuari en qualsevol temps donat, no obstant, hi ha 20 registres en bateria (veure secció a1.2.2) els quals son intercanviats sempre que la CPU canvia el mode de sistema. Els registres visibles en mode usuari són els següents:

• Registres R0-R12 (Registres de propòsit general)

Aquests 13 registres són utilitzats per propòsit general en qualsevol circumstància. Cadascun té la mateixa funcionalitat i velocitat d'execució (*performance*), i per això, no hi ha un ràpid acumulador per operacions aritmètiques, i no hi ha un registre especial per adreçament de memòria.

D'altra banda, en mode THUMB només tenim 7 registres des de R0 a R7 que poden ser accedits lliurament, mentre que els 7 registres restants (R8-R12) poden ser accedits en algunes instruccions.

• Registre R13 (SP)

Aquest registre es utilitzat com punter de pila (*Stack Pointer* –SP-) en estat THUMB. En estat ARM, el registre R13 pot utilitzar-se com punter de pila o com un registre de propòsit general.

Com s'ha vist a la taula de la figura A2.1, el registre R13 està separat en cada mode. Quan el registre R13 s'utilitza com a punter pila cada controlador d'excepcions té d'utilitzar el seu propi punter de pila.

• Registre R14 (LR)

Aquest registre és utilitzat com ***registre d'enllaç*** (*Link Register* –LR-). Quan es crida una subrutina amb la instrucció ***embrancament amb enllaç*** (*Branch with Link* –BL-). Aquest registre guarda l'adreça de retorn o valor de PC (comptador de programa), i per això només cal assignar el seu valor al PC quan vol retornar de la subrutina.

El guardat de l'adreça de retorn al registre LR és òbviament més ràpid que empilar l'adreça de retorn a memòria, no obstant, només tenim un sol registre per cada mode, llavors l'usuari haurà de empilar manualment el seu contingut abans de fer noves crides de subrutines. El mateix passa quan succeeix una excepció, llavors hi ha una canvi de mode de processador i el valor de PC (R15) es salvat en LR.

Nota: En mode ARM, R14 pot ser utilitzat també com un registre de propòsit general, si dins una subrutina no és requerit el registre LR a l'hora del seu retorn.

• Registre R15 (PC)

R15 sempre s'utilitza com a comptador de programa (PC). Quan es vol llegir el R15, normalment retorna el valor de PC+nn perquè les execucions d'instrucció està segmentades, on 'nn'depèn de la instrucció i l'estat de CPU (ARM o THUMB)

• CPSR i SPSR (Registres d'estat de Programa)

Els codis de condició actuals (flags) i el control de CPU són guardats en el registre CPSR. Quan succeeix una excepció, el vell CPSR és salvat a al registre SPSR del respectiu mode d'excepció (molt semblant quan el PC és guardat a LR).

- Registre d'Estat Actual de Programa (Current Program Status Register -CPSR-)

A sota tenim presentat el registre CPSR i a continuació una explicació detallada dels bits que el componen.

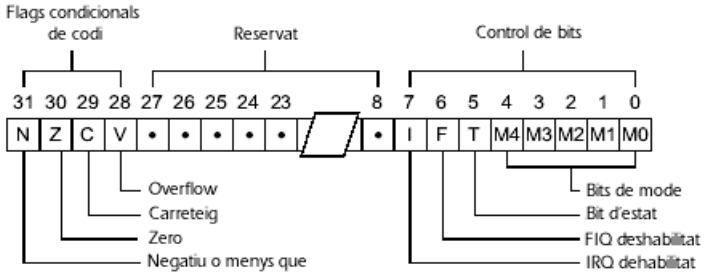


Figura A1.4 *Paraula d'estat (CPSR)*

Bit 31-28: Flags condicionals de codi (N,Z,C,V)

Aquests bits reflecteixen el resultat de les operacions aritmètiques o lògiques. En mode ARM és pot decidir si es vol que una instrucció modifiqui els bits de condició o no afegint el sufix {S} al final d'una instrucció.

També, en mode ARM, totes les operacions poden ser executades condicionalment depenen del l'estat dels bits de condició, com per exemple MOVEQ (moure si és igual). Aquesta instrucció serà executada si el bit Z és igual a 1. En estat THUMB, només les instruccions d'embrancament (salts) poden executar-se condicionalment.

Bit 27-8: Bits reservats

Aquests bits estan reservats per futures implementacions. Per futures compatibilitats, l'usuari mai hauria de canviar l'estat d'aquests bits.

Bit 7-0: Control Bits (I,F,T,M4-M0)

Aquests bits canviaran quan succeeixi una excepció. En mode privilegiats (modes no-usuari) aquests de ser canviats manualment. Els bits d'interrupció I i F són utilitzats per deshabilitar les interrupcions IRQ i FIQ respectivament (posant un "1" significa deshabilitar).

El Bit T senyalitza l'estat actual de la CPU (0 = ARM, 1 = THUMB), aquest bit mai haurà de ser canviat manualment, l'intercanvi entre ambdós estats només pot ser efectuat mitjançant la instrucció *embrancament i canvi* (*Branch and exchange* -BX-)

Els bits de mode (M4-M0) expliquen l'actual mode operatiu (veure secció A1.2.2).

```
Binary Hex Dec  Expl.
10000b 10h 16 - Usuari (no-privilegiat)
10001b 11h 17 - FIQ
10010b 12h 18 - IRQ
10011b 13h 19 - Supervisor (SWI)
10111b 17h 23 - Abortar
11011b 1Bh 27 - No definit
11111b 1Fh 31 - Sistema (mode privilegiat 'Usuari')
```

No és permès l'escriptura d'altres valors en els bits de mode.

- **Registre salvaguardat del registre CPSR (Saved Program Status Registers -SPSR_<mode>-)**

Addicionalment, per el registre CPSR, tenim 5 registres SPSR per cada mode operatiu (veure secció A1.2.2) : SPSR_fiq, SPSR_svc, SPSR_fiq, SPSR_irq i SPSR_und.

Quan la CPU entra en una excepció, el registre d'estat de programa (CPSR) serà copiat al respectiu registre SPSR_<mode>. Cal dir que, només hi ha un sol SPSR per cada mode, per això les excepcions d'igual consideració dins l'actual excepció seran permeses només si el controlador d'excepció guarda el contingut actual del SPSR a memòria.

Per exemple, per una excepció IRQ: s'entra en mode IRQ, i el registre CPSR és copiat al registre SPSR_irq. Si el controlador d'interrupció vol habilitar més interrupcions IRQ dins l'actual, llavors primer s'haurà d'empilar el valor SPSR_irq abans de permetre-ho.

A1.2.2 Modes de Processador

El processador ARM7TDMI té set modes els quals 6 són de sistema: Usuari, Sistema, IRQ, FIQ, SuperVisor, d'Abort i Indefinit.

El mode per defecte és el mode usuari. Certs esdeveniments fan disparar el canvi d'un mode. En alguns modes, s'intercanvia un conjunt de registres del mode entrant i reemplaçant l'actual conjunt de registres fins que es surt del mode.

Els modes operatius en que es pot trobar el processador es descriuen a continuació:

Mode usuari: Aquest és el mode per defecte.

Mode sistema: Aquest es el mode usuari privilegiat pel sistema operatiu. En aquest mode, es poden fer tant crides de sistema com crides de BIOS.

IRQ: S'entra a aquest mode quan una sol·licitud de interrupció (Interrupt ReQuest) s'ha disparat. Qualsevol interrupció en la GBA serà cridada en mode IRQ.

FIQ: Aquest mode és entrat quan és disparà una Sol·licitud Ràpida de Interrupció (Fast Interrupt reQuest). Com que totes les interrupcions de hardware en la GBA generen interrupcions, per defecte, aquest mode queda sense utilitzar, no obstant, aquest mode pot ser activat manualment amb la instrucció "msr".

SVC: Mode Supervisor (SuperVisor mode). S'entra quan s'executa una d'interrupció per software (Software Interrupt -SWI-). La GBA entra en aquest estat quan es crida la BIOS via SWI.

ABT: Mode Avortar (ABorT mode). S'entra després que una dada o instrucció avorta en l'etapa de fetch.

UND: Mode Indefinit (UNDefined mode). S'entra quan una instrucció no definida s'ha executat.

Registres en bateria (Banked registers)

El processador ARM7tdmi conté un conjunt de registres en bateria (veure Figura A1.5) que intercanvia els registres del mode usuari per els d'un mode privilegiat (mode sistema) quan entra. Quan el mode privilegiat surt, desfà l'intercanvi restaurant els valor de registres del mode usuari.

En mode IRQ disposem dels registres en bateria r13_irq i r14_irq que són intercanviats per els registres r13 i r14 respectivament. El registre CPSR es salvarà en el registre SPSR_irq.

En mode FIQ disposem dels registres en bateria r8_fiq, r9_fiq, r10_fiq, r11_fiq, r12_fiq, r13_fiq, r14_fiq, i SPSR_fiq. Igual que passa en el mode IRQ, aquest registres s'intercanvien amb els corresponents al mode usuari quan es dispara aquest mode.

En mode SVC, disposem dels registres en bateria r13_svc, r14_svc i SPSR_svc.

En mode ABT disposem dels registres en bateria r13_abt, r14_abt i SPSR_abt.

En mode UND disposem dels registres en bateria r13_und, r14_und i SPSR_und.

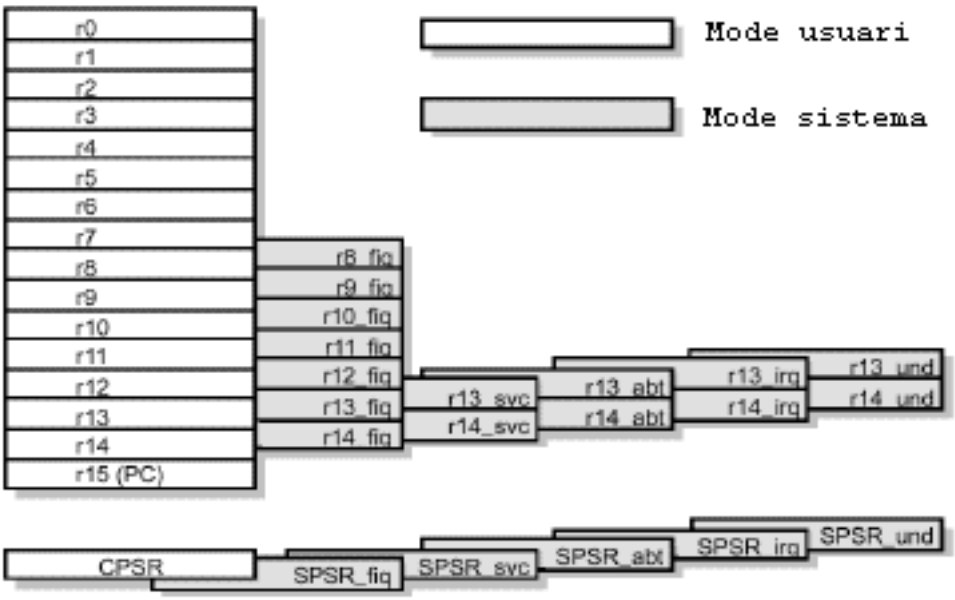


Figura A1.5 *Registres en bateria equivalents pels diferents modes de processador*

A1.2.3 Estats de CPU

El processador ARM7tdmi té quatre possibles estats dins la GBA:

- Estat *THUMB* (GBA): En aquest estat, la CPU executa instruccions alineades de 16 bits.
- Estat *ARM* (GBA)*:* És l'estat per defecte. Executa instruccions alineades de 32 bits.
- Estat *CGB*, es configura el processador ARM per simular una CPU Z80/8080 de 8bit a 4.2MHz o 8.4MHz per compatibilitzar amb Game Boy Color (GBC).
- Estat *DMG*, es configura el processador ARM per simular una CPU Z80/8080 8bit CPU a 4.2MHz per compatilitzar amb Game Boy (GB).

Dels quatre estats que pot tenir la CPU, en aquest projecte comprendrem només els estats *ARM* i *THUMB*, ja que són els estats que exclusius pel mode operatiu GBA.
Els estats *ARM-THUMB*, no poden operar simultàniament a la GBA, tanmateix poden alternar-se amb la instrucció *BX* (Branch and Exchance) oferint una sèrie de avantatges i inconvenients que tot seguit comentem,

Avantatges i desavantatges quan s'executa en estat *THUMB*

Hi ha dos grans avantatges a l'hora d'utilitzar l'estat THUMB:

- Ràpida execució, aproximadament un 160 %, quan s'utilitza un sistema de memòria amb un bus de 16 bits (tal com codis d'instrucció guardats a la memòria externa).
- Grandària de codi reduït, aproximadament, a un 65 % respecte el codi ARM.

Els desavantatges són:

- No té codis d'instrucció multifuncionals com en l'estat ARM, degut aquest motiu algunes vegades és requerit utilitzar més d'una instrucció per tenir el resultat semblant al d'una instrucció en estat ARM.
- Moltes instruccions només permeten accedir directament en els 8 primers registres (r0-r7).

Avantatges i desavantatges quan s'executa en estat *ARM*

Bàsicament, hi ha dos avantatges en estat ARM:

- Cada operació simple s'executa a la màxima velocitat quan s'utilitza un sistema de memòria de 32 bits com la memòria interna (veure secció A1.3.3).
- Totes les instruccions permeten accedir als 16 registres de forma directa.

Desavantatges:

- Les operacions en estat ARM no són suficient efectives quan és treballa en un sistema de memòria de 16 bits.
- El codi de programa ocupa molt espai.

Recomanacions

A causa de que el bus de la ROM del *GamePak* està limitat a 16 bits, les instruccions *ARM* (operacions de 32 bits) dins la ROM del *GamePak* no resulta un rendiment alt d'execució. Per això, és recomanable utilitzar instruccions *THUMB* (operacions de 16 bits) que permeten ser llegides en un sòl cicle de rellotge. Per altra banda, les instruccions *ARM* causen un gran grau d'execució quan es vol copiar dades de memòria externa a memòria interna Per tant, a l'hora de programar, és recomanable:

- Utilitzar estat *THUMB*, per execucions normals de codi de programa, per exemple, a memòria externa o dins el *GamePak*.
- Utilitzar estat *ARM* per subrutines especials que necessiten una ràpida execució (tal com rutines d'interrupció o algorismes complicats).

A1.2.4 Tipus de CICLES

Els tipus de cicles que podem trobar durant l'execució d'operacions ARM, són tres: **Cicle no seqüencial (N), Cicle seqüencial (S) i cicle Intern (I)**.

N – Cicle no seqüencial

Sol·licitud d'un transferència des de/a una adreça la qual **NO** està relatada a l'adreça utilitzada a l'anterior cicle. En la GBA s'anomena primer accés. El temps d'execució de 1 cicle no seqüencial (1N), equival a un tick[2] de rellotge més un accés de estat d'espera no seqüencial.

S – Cicle seqüencial

Sol·licitud d'una transferència des de/a una adreça la qual és la mateixa l'adreça utilitzada al cicle anterior. Per això, per carregar/guarda blocs consecutius des de/a memòria de 16 o 32 bits, el primer accés serà no seqüencial, però els pròxims seran seqüencials. Aquest tipus d'accés a la GBA s'anomena segon accés.
El temps d'execució per un cicle seqüencial (1S) és de 1 tic de rellotge més un accés de estat d'espera seqüencial.

I – Cicle intern

Passa quan la CPU està ocupada i, per tant, no es requereix cap sol·licitud de transferència a/des de memòria. El temps d'execució de 1 cicle intern (1I), equival a un tic[1] de rellotge (sense esperes d'accés).

.

[2] A la GBA un tic equival a $59'59 \cdot 10^{-6}$ segons

A1.3. Memòria

Presentem les àrees de memòria contemplades per la CPU, amb l'ampla del bus de dades corresponent, la informació que permet llegir/guardar per cada tipus d'accés (8,16 o 32 bits). També, és descriuen els costs de cicles que calen per cada tipus accés,

Adreces	Tipus de memòria	Bus	Escriptura	Lectura	Cicles
0x00000000 0x00003FFF	ROM del sistema (16 Kbyte)	32	8 / 16 / 32	-	**1 / 1 / 1**
0x02000000 0x0203FFFF	Memòria externa de treball(256 Kbyte)	16	8 / 16 / 32	8 / 16 / 32	**3 / 3 / 6 ****
0x03000000 0x03007FFF	Memòria Interna de treball(32 Kbyte)	32	8 / 16 / 32	8 / 16 / 32	**1 / 1 / 1**
0x04000000 0x040003FF	IO RAM(1 Kbyte)	32	8 / 16 / 32	8 / 16 / 32	**1 / 1 / 1**
0x05000000 0x050003FF	Paleta RAM(1 Kbyte)	16	8 / 16 / 32	16 / 32	**1 / 1 / 2 ***
0x06000000 0x06017FFF	Vídeo RAM(96 Kbyte)	16	8 / 16 / 32	16 / 32	**1 / 1 / 2 ***
0x07000000 0x070003FF	OAM (1 Kbyte)	32	8 / 16 / 32	16 / 32	**1 / 1 / 1 ***

0x08000000 0x09FFFFFF	ROM *Game Pak* amb registre d'espera d'estat 0 (32 Mbytes)	16	8 / 16 / 32	-	**5 / 5 / 8 **/*****
0x0A000000 0x0BFFFFFF	Imatge de ROM *Game Pak* amb registre d'espera d'estat 1 (32 Mbytes)	16	8 / 16 / 32	-	**5 / 5 / 8 **/*****
0x0C000000 0x0DFFFFFF	Imatge de ROM *Game Pak* amb registre d'espera d'estat 2 (32 Mbytes)	16	8 / 16 / 32	16 / 32	**5 / 5 / 8 **/*****
0x0E000000 0x0E00FFFF	RAM *Game Pak* (0-64 Kbyte)	8	8 / - / -	8 / - / -	**5 / - / - ****
0x0FFFFFFF					

Regió de memòria interna
Regió de memòria externa
Regions sense utilitzar o imatges de memòria.

```
*  +1 cicle si la GBA accedeix a la memòria de vídeo al mateix temps
**  Espera d'estat per defecte.
*** Temps per separat en accessos seqüencials i accessos no seqüencials.
```

A1.3.1 ROM del Sistema (System ROM):

```
Rang d'adreçament: 0x00000000 - 0x0003FFF
Grandària: 16kb
Amplada de bus: 32 bit
```

Descripció

El rang d'adreces 0x0 - 0x00003FFF contenen les funcions de la BIOS, per la qual cosa són executables, però no llegibles.

A1.3.2 Memoria Externa de Treball (EWRAM)

```
Rang d'adreçament: 0x02000000-0x0203FFFF
Grandària:  256kb
Amplada de bus: 16 bit
Adreça imatge: Cada  262144 bytes des de l'adreça 0x02000000 a
l'adreça 0x02FFFFFF.
```

Descripció

Aquest espai és disponible per les dades i codi del nostre joc. Aquesta és l'àrea més gran que podem trobar integrada a la GBA, no obstant, la transferència a/des de la memòria externa és d'una grandària de 16 bits i per aquest motiu, és consumeix més cicles que els necessaris per accessos de 32 bits. Per això, és recomanable que el codi *ARM* estigui a la regió de memòria interna (Internal Work Ram –IWRAM-), ja que s'aconsegueix un alt nivell de rendiment que no pas en memòria externa. En canvi, en mode *THUMB*, és recomanable treballar-hi quan estem a memòria externa (External Work RAM –EWRAM-), ja que totes les operacions que s'executa en aquest mode són de 16 bits.

A1.3.3 Memoria Interna de Treball (IWRAM)

```
Rang d'adreçament: 0x03000000-0x03007FFF
Grandària:  32kb
Amplada de bus: 32 bit
Adreça imatge:  Cada  32768 bytes des de l'adreça 0x03000000 a
l'adreça 0x03FFFFFF
```

Descripció

Aquest espai esta disponible per ser utilitzat. Internament, està ubicada a la pastilla ARM7. Es la RAM més ràpida de totes dins la GBA degut al seu bus de 32 bit. A comparació dels busos ROM i EWRAM (que són de 16 bits), es guanya la màxima eficiència en l'execució quan el codi *ARM* (32 bits) resideix a memòria interna.

A1.3.4 IO RAM

```
Rang d'adreçament: 0x04000000-0x040003FF
Grandària:  1Kb
Amplada de bus:  Dades dobles de 32 bit
```

Adreça imatge: Només la paraula a l'adreça 0x04000800 es figura cada 65536 bytes des de l'adreça 0x04000000 a l'adreça 0x04FFFFFF.

Descripció

En aquesta regió conté els Registres IO Mapejats o del Circuit Específic Integrat d'Aplicació (Application Specific Integrated Circuit –ASIC-) a la GBA, on és utilitzada per controlar gràfics, so, DMA, i altres dispositius.

A1.3.5 Paleta RAM

```
Rang d'adreçament: 0x05000000-0x050003FF
Grandària:  1kb
Amplada de bus:  16 bit
Adreça imatge: Cada  1024 bytes des de l'adreça 0x05000000 a l'adreça
0x5FFFFFF.
```

Descripció

Aquesta àrea especifica els valors de color 16 bit per els modes paleta. Hi ha dos àrees per la paleta:

- Pel fons (background en anglès) a l'adreça 0x05000000
- Pels *sprites*[3] a l'adreça 0x05000200

Cadascun d'ells és indexada independentment i pot ser configurada, com una paleta de 256 colors, o com 16 paletes individuals de 16 colors cadascuna obtingui un màxim de 512 colors diferents sumant totes dos paletes.

A1.3.6 Vídeo RAM

```
Rang d'adreçament: 0x06000000-0x06017FFF
Grandària:  96kb
Amplada de bus: 16 bit
Adreça imatge: Els bytes e l'àrea compresa 0x06010000 - 0x06017FFF és
reproduïda des de l'adreça 0x06018000  a l'adreça  0x0601FFFF.  La
regió compres a l'adreça 0x06000000 - 0x06020000 figura  cada  131072
bytes des de l'adreça 0x06000000 a l'adreça 0x06FFFFFF.
```

Descripció

La memòria de vídeo (Vídeo RAM –VRAM-) és utilitzada per guardar el *frame buffer*[4] en mode gràfic de mapa de bits que ho permeten(veure apartat A1.4.1), i les dades i el mapa basat en rajoles per els modes text i pels modes d'escalats/rotat.

A1.3.7 OAM

```
Rang d'adreçament: 0x07000000-0x070003FF
Grandària:  1kb
Grandària de dades: 32 bit
```
Adreça imatge: cada 1024 bytes des de l'adreça 0x07000000 a l'adreça 0x07FFFFFF

En aquí aquesta regió de memòria hi tenim els atributs dels objectes (Object Attribute Memory –OAM-) , i s'utilitza per el control dels *sprites*.

---

[3] Els *sprites* són objectes que és mouen per la pantalla (objectes actius) i estan dotats d'unes propietats com la velocitat, vitalitat, etc. Aquests objectes poden moure's segons els moviments d'un altre objecte (objectes intel·ligents) o recorre uns moviments passats per paràmetre (objectes tontos).

[4] El *frame buffer* és el bùfer on hi ha la informació dels pixels a ser representats per pantalla.

A1.3.8 *GAME PAK*

Com s'ha comentat, el codi del joc es pot executar a la regió de memòria externa. D'altra banda, si s'insereix un cartutx *GamePak*, el joc/aplicació s'executarà des de a la ROM del cartutx, amb una capacitat màxima de 256 Mbits (32 Mbytes).

ROM *GAME PAK*

```
Adreça de començament: 0x08000000
Grandària:  Un màxim de 256 Mbits o 32 MegaBytes.
Alineació de d'informació: 16 bit
Espera d'estat: 0
```

La ROM en el cartutx de joc apareix en aquesta àrea. Si una cartutx és present al començament, la instrucció que es troba la posició 0x08000000 és carregada dins el comptador de programa (PC) i l'execució comença des de aquí. Les transferències, a la ROM i des de la ROM, són d'una amplada de 16 bits.

ROM *GAME PAK* IMATGE 1

```
Començament: 0x0A000000
Grandària: Un màxim de 256 Mbits o 32 MegaBytes.
Amplada de bus:  16 bit
Espera d'estat: 1
```

És una imatge de la ROM *Game Pak*.

ROM *GAME PAK* IMATGE 2:

```
Adreça de començament: 0x0C000000
Grandària: Un màxim de 256 Mbits o 32 MegaBytes.
Amplada de bus: 16 bit
Espera d'estat: 2
```

És una imatge de la ROM *Game Pak*.

CART RAM

```
Adreça de començament: 0x0E000000 (també pot apareixe a l'adreça
0x0F000000)
Grandària:  0-64kb
Grandària de port: 8 bit
```

Pot ser Flash ROM o SRAM. Utilitzat primàriament per salvar informació del jugador durant joc. La versió SRAM pot tenir fins a 64kb, però normalment és de 32kb. La SRAM té una bateria per el guardat, la vida d'aquest guardat és la més llarga (en termes de quantes vegades s'hagi escrit) que tots els mètodes de guardats. A quan la versió Flash ROM, normalment té 64kb. La seva vida ve determinat per el numero de rescrits que és fa per sector (alguns manufacturadors estimen un mínim de 10000 rescrits)

A1.4 Vista dels Gràfics GBA per hardware

La GBA té una pantalla LCD color TFT d'una grandària de 240x160 pixels amb una taxa de refresc d'exactament 280'896 cicles per *frame*, o al voltant de 59.73 Hz. Molts dels programes en la GBA necessiten estructurar-se al voltant de la taxa de refresc. Cada refresc consisteix en un període de dibuixat de 160 línies verticals (Vertical Draw –*VDRAW*-) seguit d'un període de 68 scanlines[5] sense efecte (Vertical Blank –*VBLANK*-). Durant els períodes *HDRAW* i *VBLANK* el hardware gràfic de la GBA processa la informació de fons i dels objectes (*sprites*) i els dibuixa a la pantalla. Fora dels períodes *HBLANK* i *VBLANK*, el codi de programa pot modificar les dades de fons o objecte sense el risc de alterar el sistema de vídeo.

A1.4.1 Modes de vídeo

La GBA pot inicialitzar 6 modes diferents. Els modes 0,1 i 2 estan basat fons de rajoles i els modes 3, 4 i 5 estan basats en fons de mapa de bits.

A1.4.1.2 Modes basats en Rajoles (Modes 0,1 i 2)

En aquests modes, la pantalla es subdivideix en quadrats de 8x8 pixels (d'aquí el terme rajola). El conjunt de rajoles formen el fons. És necessari, col·locar les rajoles a la memòria de vídeo i un mapa, que tan sols es un vector (Figura A1.6) on cada element es la referència a la textura amb uns quants atributs més. Llavors, el processador de vídeo farà una ullada al vector mapa (A1.6) i construirà la pantalla (A1.8) amb les rajoles que li hem especificat (A1.7). La grandària del fons pot ocupar des de 256x256 pixels a 1024x1024 pixels.

Figura A1.6 *Col·lecció de rajoles guardades com mapes de bits de 8x8 pixels a VRAM*

```
Map[] = {
64,64,64,64,64,64,64,6 ,6 ,64,64,64,64,64,64,64,
 64,64,64,64,7 ,64,62,06,06,64,64,64,64,64,64,64,
 64,64,64,62,06,06,06,06,06,64,64,64,64,64,64,64,
 64,64,62,06,06,06,06,06,06,64,64,64,64,64,64,64,
 64,62,06,06,06,06,06,06,06,63,64,64,64,64,64,64,
 06,06,06,06,06,06,06,06,06,06,06,06,06,06,06,06,
 64,67,06,06,06,06,06,06,06,06,06,06,06,06,64,64,
 64,64,06,06,06,06,06,06,06,06,06,06,06,06,64,64,
 64,64,06,06,06,06,06,06,06,06,06,06,06,06,64,64,
 64,64,66,66,66,66,66,66,66,66,66,66,66,66,64,64,
 64,64,64,64,64,64,64,64,64,64,64,64,64,64,64};
```

Figura A1.7 *El vector de mapa en C. Cada posició del vector fa referència a una textura.*

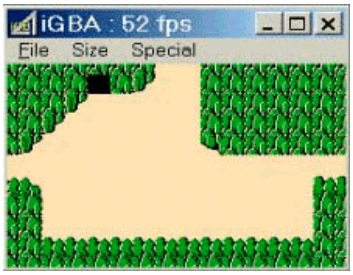


Figura A1.8 *Resultat gràfic a un emulador GBA, amb les textures de la A1.6 i el vector de mapa de la A1.7.*

[5] scanline : Línia horitzontal de pixels igual a l'amplada de pantalla

La GBA pot mostrar fins a quatre fons diferents quant opera en mode rajola. Cada fons es pot controlar independentment utilitzant diferents mapes i textures. Els seus atributs de desplaçament (scroll) també es poden controlar independentment.

Pel que fa als dibuixats de fons, el color 0 es normalment transparent permetent que els fons amb major prioritat es vegin a través dels que tenen menys. Per defecte, el dibuixat dels fons segueix l'ordre 3, 2, 1 i 0; amb el que el fons 0 apareixerà damunt del fons 3, però es pot canviar. Alguns fons també tenen la possibilitat de ser rotats y escalats (és un bon efecte per qualsevol mapa).
Pel que fa les textures, es poden guardar com a mapes de bits de 256 o de 16 colors. Ambdues utilitzen la mateixa paleta però els fons són diferents. Poden utilitzar modes de color diferents. Les paletes de 16 colors funcionen dividint la paleta de 256 colors en 16 paletes de 16 colors cada una. A més, en el mode de 16 colors pots tenir els 256 colors en la pantalla, mentre cada rajola només utilitzi una de les 16 paletes.

Les següent resumeixen les habilitats de cadascun dels modes rajola i de cada fons.

	MODE 0	MODE 1	MODE 2
Fons Disponible	**Tots**	**0,1,2**	**2,3**
Rotació/Escalat?	**NO**	**Fons 2**	**Fons 2,3**

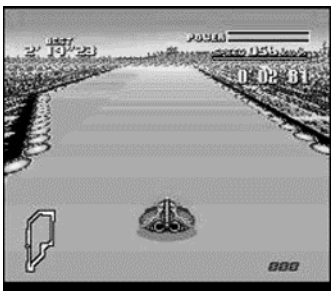
TaulaA1.1 *Modes de Pantalla*

	Numero Màxim de Rajoles	Grandària del mapa (en pixels)	Número de Colors per Rajola	Característiques Especials
Rotació/Escalat	**256**	**128x128, 256x256 512x512, 1024x1024**	**256**	**Rotació/Escalat**
Fons de text pla	**1024**	**256x256, 512x256 256x512, 512x512**	**256 o 16**	**Intercanvi horitzontal/vertical de rajoles**

Taula A1.2 *Característiques de fons*

Els modes que utilitzen fons transformables (rotables i/o escalables) permeten la simulació d'un mode extra a la GBA anomenat "**mode 7".**

El nom **"mode 7"** prové de la consola SNES (Super Nintendo o Super Famicom, a Japó) ja que inicialitzava el **mode 7** per hardware. Aquest mode, consisteix en aprofitar tècniques de rotats i escalats per simular un renderitzat de pla 3D. Un exemple directe on s'utilitza aquest mode, és a la pista de carreres en el joc F-Zero o el joc Axelay a les etapes 1,3 i 5 de la SNES.



A l'esquerra joc Axelay i a la dreta joc F-Zero.

La GBA no incorpora aquest mode, però es pot configurar manualment inicialitzant els registres especials de rotat i/o escalat de la GBA durant el període *hblank*. (veure annex 4 per més informació)

A1.4.1.2 Modes basats en bitmap (Modes 3,4 i 5)

Els modes basats en mapa de bits (*bitmaps*) depenen d'un bùfer abans de que la imatge sigui dibuixada per pantalla. El mode 4 utilitza una paleta de 8 bits, i els restants (mode 3 i 5) escriuen el color codificat en 16 bits a la posició de cada pixel.

En aquest modes els fons estan basats en *bitmaps* i que són tractats com un fons 2 (BG2) per propòsits de ser rotat/escalat o fer-lo transparent. En mode *bitmap*, les dades del *frame buffer* s'estén a la regió de les dades d'objecte i de rajola, limitant-t'ho al rang d'adreces 00x6014000 - 0x6018000 (índex d'sprites).

Aquests tipus de modes a la GBA són:

Mode 3:

Inicialitza l'estándar mode gràfic 240x160 basat en *bitmap* de 16 bits i sense paleta. El mapa comença a la posició 0x06000000 i la longitud del *frame* es de 240x160x2bytes/posició = 76800 bytes.

El *frame buffer* ocupa molt d'espai la memòria de vídeo i per la qual cosa no permet aplicar una tècnica del tipus bolcat de pàgina (*page flipping*). La única opció que queda és copiar la informació en un bùfer a memòria externa i bolcar-lo més tard memòria de vídeo.

Mode 4:

Mode mapa de bits de 8 bit amb paleta i amb una resolució de 240x160.

Com que la memòria de vídeo utilitzada és la meitat respecte el mode 3, la memòria de vídeo queda partida en dos *frame buffers*. Un *frame buffer* s'utilitza per veure el *frame* actual mentre que a l'altre *frame buffer* es copia la informació del proper *frame* a visualitzar-se, tècnica coneguda com *page flipping*. Per això, aquest mode permet escriure la informació de pixel directament a memòria de vídeo.
El bit 4 del registre REG_DISPCNT activa el *frame buffer* a la posició 0x06000000 quan el bit 4 és 0 i a la posició 0x0600A000 en cas contrari.
A quan la paleta, s'ubica la posició 0x50000000, i conté 256 colors on cada color és de 16 bits.

Encara que es té avantatge a l'hora de copiar la informació del pixel directament a VRAM, en el mode 4 només permet ser accedida de 16 en 16 bits. Això vol dir que, per escriure un pixel en aquest mode, s'ha de llegir un dels dos pixels adjacents de la posició on és vol escriure, emmascarar el pixel que vols canviar i escriure els dos pixels, de nou, a memòria. Degut això, la majoria de motors gràfics que utilitzen el mode 4, escriuen 2 pixels de cop a la memòria de vídeo. Això vol dir que l'ampla efectiva de pantalla, en aquests casos, és de 120.

Mode 5:

Com el mode 3, esta basat en un *bitmap* de 16 bits i sense paleta, però a diferència, té una resolució de 160x128. L'avantatge que presenta aquest mode respecte el mode 3, es que tenim suficient espai per tenir dos *frame buffer* per aplicar la tècnica del *page flipping*. Igual que passava al mode 4, amb el bit 4 del registre REG_DISPCNT activa el *frame buffer* a la posició 0x06000000 quan el bit 4 és 0 i a la posició 0x0600A000 en cas contrari.

A1.4.2 Format de color

Tots els colors (amb dos mapejats o paletitzats) representen valors de 16 bits, utilitzant 5 bits per el vermell, verd i blau, i ignorant el bit 15 (figura *A1.9*). En el cas de memòria paletitzada, els pixels d'una imatge són representats com índexs de 8 bits o 4 bits dins la RAM de la paleta. La paleta comença a la posició 0x50000000 pel fons i 0x5000200 per els *sprites*. Cada paleta representa una taula de 256 colors de 16 bits de longitud. En el cas dels fons basats en *bitmaps* i sense paleta (modes 3 i 5), la informació de pixel representa directament una informació de color de 16 bits.

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
X	B	B	B	B	B	G	G	G	G	G	R	R	R	R	R

R = Vermell
G = Verd
B = Blau

Figura A1.9 *Format del color*

A1.5 Accés Directe a Memòria (DMA)

La GBA té 4 canals de DMA diferents a la GBA, cadascun amb la seva pròpia prioritat i no podem fer-los servir alhora. Cada canal té diferent utilitat i diferent capacitat, tal com es descriuen a continuació:

DMA 0 (0x40000B8) : Aquest és el canal amb la màxim prioritat i no pot ser interromput pels altres canals. S'utilitza quan el temps és crític, tal com la còpia d'informació OAM durant el període *HBLANK*. Només podem utilitzar aquest canal a la memòria interna (iwram) de la GBA. Això vol dir que no podem transferir cap dada des de memòria externa.

DMA 1 (0x40000C4) i DMA 2 (0x40000D0): Aquests canals són utilitzats principalment pel só directe (Direct Sound) i tenen la prioritat més alta després del DMA 0. Aquests canals poden transferir des de qualsevol regió de memòria i són els únics canals que és poden fer servir juntament amb el bùfer de so FIFO.

DMA 3 (0x40000DC): Aquest és el canal DMA de propòsit general i utilitzat freqüentment per copiar informació des de qualsevol àrea de memòria.

Ara és el moment per discutir el mode de transferència DMA. Bàsicament, el mode estableix quan es vol que comenci la transferència DMA, i quanta informació es vol transferir per escriptura (16 o 32 bits a la vegada). També es controla la repetició de transferència quan succeeixi alguna interrupció. Si s'activa la repetició de transferència es pot alterar l'adreça font o destí automàticament abans de la pròxima transferència.

REG_DMAxCNT

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Enable	IRQ	Període		-	16/32	Repetir	Font	Dest	-	-	-	-	-	-	-

Bits 0-15: Aquest és el comptador de paraula. Els valor d'aquests bits determinen quantes paraules es vol copiar.

Bit 31: Aquest bit dona llum verda perquè comenci la transferència del canal. Aquest bit hauria de ser l'últim a posar, després d'haver assignat les corresponents adreces i el resta del mode. Si aquest bit és zero el canal romandrà parat.

Bit 30: Habilitació de interrupció. Aquest bit permet que, quan la transferència DMA hagi acabat, aquest generi una interrupció.

Bit 28-29: Període - aquests bits controlen quan es vol que comenci la transferència DMA.

00- Començar immediatament. Tant bon punt es posi el bit 31 a '1'la transferència començarà.
01- Començar al període Vblank. Tan bon punt l'última línia de pantalla sigui dibuixada, començarà la transferència.
10- Començar al període Hblank. La transferència començarà al pròxim període Hblank.

11- Invàlid per tots els canals excepte per el DMA 3. Quan s'utilitzi aquest mode per el DMA 3 començarà la transferència a l'inici del període de renderització, o sigui, després que la primera línia horitzontal hagi estat dibuixada a pantalla. A partir d'aquí, s'efectuaran transferències cada cop que s'hagi dibuixat una

línia, durant el període hblank, permeten la còpia d'una línia d'informació horitzontal des de el frame-bùfer. Principalment, l'ús d'aquest mode DMA està destinat per el sistema gràfic mode 3 el qual no disposa backbuffer. Mentre que la GBA dibuixa una línia de la pantalla, el DMA copia la informació per reemplaçar la línia que ja s'ha dibuixat. Aquest fet assegura que sempre s'actualitzarà el bùfer de vídeo per la línia que ja s'havia dibuixat assegurant absència de parpelleig. Aquest mode facilita l'afegiment de dispositius tal com cameres que tenen cadenes d'informació.

Bit 26: Grandària de paraula- Si és '0'el DMA copiarà paraules serà de 16 bits. Si és '1'copiarà paraules de 32 bits.

Bit 25: Repetició DMA- Aquests bit implica que el DMA repetir i la transferència a cada període Hblank o Vblank, especificat en els bits de Període.

Bits 23-24: Aquests bits controlen el pas d'increment a la d'adreça font afectat per cada transferència.

00- Increment- L'adreça font s'incrementarà a cada transferència.
01 -Decrement- L'adreça font decrementarà a cada transferència.
10-fixat- Bo per omplir una regió de memòria amb la mateixa informació i a cada frame.
11-Valor invàlid

Bits 21-22: Aquests bits controlen el pas d'increment a la d'adreça destí afectat per cada transferència.

00- Increment- L'adreça destí s'incrementarà a cada transferència.
01 -Decrement- L'adreça destí es decrementarà a cada transferència.
10-fixat-accés directe al bùfer FIFO.
11-Si és posa aquest valor, l'adreça destí incrementarà. Al final de cada transferència DMA, l'adreça destí tornarà a tenir la adreça inicial.

Annex 3 Eines del desenvolupament

A2.1 kit HAM

El kit construït per *Emanuel Schleussinger* està disponible a la plataforma *Windows* i *Linux*, descarregable a l'adreça http://www.ngine.de/. Aquest annex, inclou tota la informació sobre aquest kit necessària en el desenvolupament d'aquest projecte.

En aquest kit inclou dos eines fonamentals pel desenvolupament d'aquest projecte: El compilador per generar ROMS GBA i un emulador per executar-les. A més a més, també s'inclouen multitud d'aplicacions basats en el hardware de la GBA que poden facilitar, en major part, les tasques del nostre projecte.

Els projectes HAM s'organitzen amb un fitxer anomenat "**makefile**". Els scripts HAM realitzen les crides necessàries basat en el "makefile" per crear la ROM GBA d'una manera directa i simple. L'extensió de les ROMS GBA creades és ".gba" la qual, és reconeguda per qualsevol emulador de GBA.

A2.1.1 El "makefile"

Principalment, el "makefile" fa la funció de projecte i, com a tal, inclou els fitxers de codi font que es volen compilar i s'especifica el nom de la ROM GBA que es vol crear.
Per exemple, si volguéssim compilar un projecte on tenim un únic fitxer "main.c" i que el nom de la ROM fos "el_meu_programa" el contingut d'aquest fitxer seria,

```
#################################
# HAM Makefile
#################################

include $(HAMDIR)/system/master.mak

#
# Set the name of your desired GBA image name here
#
PROGNAME = el_meu_programa

#
# Set a list of files you want to compile
#
OFILES += main.o

#####################################
# Standard Makefile targets start here
#####################################

all : $(PROGNAME).$(EXT) clean

#
# Most Makefile targets are predefined for you, suchas
# vba, clean ... in the following file
#
include $(HAMDIR)/system/standard-targets.mak

#####################################
# custom  Makefile targets start here
#####################################

#gfx: makefile
#        gfx2gba -t8 -m -fsrc gfx\\*.bmp
#        gfx2gba -t8 -m -rs -fsrc gfx\\facerot.bmp
```

Del contingut de l'exemple cal observar la part en negreta a on, a la primera part (PROGNAME=...) es dona el nom del fitxer binari ROM GBA. En l'exemple, si no es donés cap error de compilació i de enllaçament, és crearia la ROM "el_meu_programa.gba" al directori actual. També, es pot especificar la ruta on es vol que és creí la ROM. Per exemple,

```
PROG_NAME = c:/temp/el_meu_programa
```

Crearia la ROM "el_meu_programa.gba" al directori `C:\temp`.

La segona part en negreta (FILES += ...) representa els fitxers C, C++ o ensamblador implicats en la creació de la ROM GBA. El compilador tractarà tots els fitxers inclosos en aquesta secció. Si volguéssim incloure més fitxers, seria a continuació del "main.o" seguit d'espai. Per exemple,

```
FILES+= main.o fitxer1.o fitxer2.o fitxer3.o ...
```

Per últim observar que, "makefile" tracta comentaris amb el caràcter #.

La resta d'informacióés exclusiva per HAM. Són fitxers script externs que tenen la finalitat de configurar el compilador per tal de crear la ROM GBA de forma automàtica i optimitzada. Per cada nova ROM GBA, es recomana fer servir un "makefile" original de plantilla i modificar només els apartats de nom de programa i inclusió de fitxers codi font.

A2.1.2 Creació d'una ROM GBA

Per crear una ROM GBA es pot fer des de la consola MS-DOS o des de un entorn de programació anomenat *Visual HAM*, inclòs al kit HAM.

Creació d'una ROM GBA desde la consola MS-DOS

Si volem crear una ROM des de la consola MS-DOS, s'ha d'executar el procés per lots "startham.bat" al directori arrel on s'ha instal·lat el kit HAM. Llavors, anar al directori on tenim els fitxers de codi font i el "makefile" correctament editat i, per últim, introduïm la comanda *make* .
Make és un script que crida al compilador i, conjuntament amb el fitxer "makefile", creen la ROM GBA de forma automàtica al directori especificat al fitxer "makefile".
Al script *make*, podem afegir un paràmetre addicional per executar, seguidament d'haver creat la nostra ROM, altres aplicacions d'utilitat basades en ROMS GBA. Uns dels paràmetres més importants que farem servir són:

- *make vba*: Desprès d'haver creat la ROM GBA, s'obre l'aplicació emulador GBA, integrat al kit HAM, anomenat VBA (VisualboyAdvance) que obrirà i executarà la ROM.
- *make gdb*: S'entra al debug ARM-THUMB del nostre projecte.

Creació d'una ROM GBA desde la consola *Visual HAM*

Una altre manera de crear les ROMS GBA és a través d'un entorn de programació *Visual HAM* (figura A2.1) inclòs al kit HAM. *Visual HAM* és similar a *Visual C* i com aquest, crea nous projectes i inclou els fitxers codi font al "makefile". *Visual HAM*, inclou icones que criden l'script *make* amb els paràmetres esmentats anteriorment.

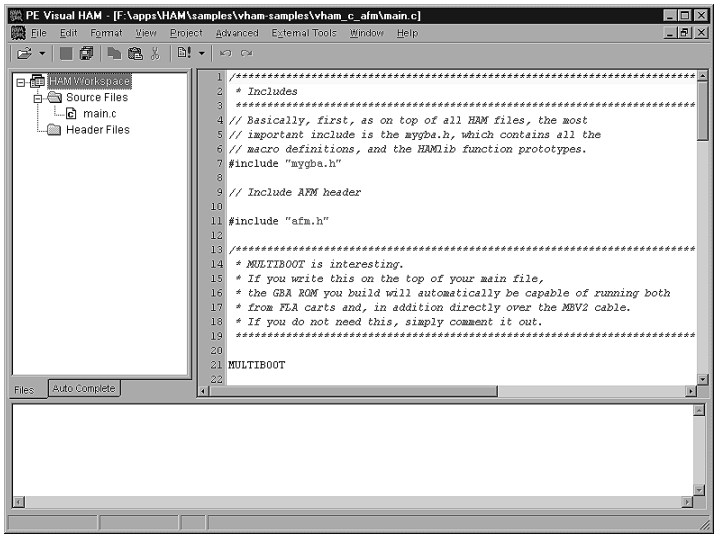


Figura A2.1 *Aspecte que ofereix l'entorn de programació Visual HAM.*

A2.1.3 Referència de funcions HAMLib

• Funcions d'inicialització HAMLib

Les funcions de inicialització HAMLib utilitzades en aquest projecte són les següents:

void ham_Init()
inicialitza el sistema HAMLib.

`void ham_Init()`

Inicialitza el sistema HAMLib. Es necessita cridar ***ham_Init()*** a l'inici de cada programa creat amb HAMLib. Aquesta funció inicialitza tots els sistemes importants de HAM, i es recomana fer la respectiva crida al principi de la funció ***main()***, per assegurar una inicialització sense conflictes.

• Funcions de control de fons

Les funcions de control de fons HAMLib utilitzades en aquest projecte són les següents:

`void ham_InitBg(u16 bgno, u16 active prio, u16 mosaic)`
Inicialitza un fons basat en rajoles.

`void ham_SetBgMode(u8 bgmode)`
Inicialitza el hardware de la GBA per operar en un nou mode de fons.

`void ham_SetBgXY(u16 bgno, u16 x, u16 y)`
Assigna l'offset XY a capa de fons.

```
void ham_InitBg(u16 bgno,
                u16 active,
                u16 prio,
                u16 mosaic)
```

Inicialitza un fons basat en rajoles per utilitzar-se en HAM.

Paràmetres:

Bgno: El numero de fons que es vol inicialitzar (0-3)
Active: Activa/Desactiva el fons (0 = Desactivat ,1 = Activat)
Prio: El numero de prioritat del fons, a l'hora de mostrar-se a pantalla (0= Mínima prioritat, 3 = Màxima prioritat).
Mosaic: Habilita/Desabilita l'utilització de mosaic per aquest fons (0=Deshabilita, 1=Habilita)

`ham_SetBGMode(u8 bgmode)`

Inicialitza un nou mode gràfic a la GBA. Tota la renderització després d'inicialitzar el mode serà portat en aquest nou mode.

Paràmetres:

`Bgmode:`: El mode gràfic a inicialitzar (0-5)

- 0 = Mode Rajola, 4 fons disponibles (0-3) sense rotació ni translació.
- 1= Mode Rajola, 3 fons disponibles (0-2). Els fons 0-1 no poden ser rotats ni escalats, mentre que el fons 2 si.
- 2 = Mode Rajola, 2 fons disponibles (2-3), ambdós dos rotables i escalables.
- 3 = Mode Bitmap, un *frame-buffer* a VRAM de 16bits/pixel.
- 4 = Mode Bitmap, dos *frame-buffers* a VRAM, de 8 bits/pixel.
- 5 = Mode Bitmap, dos *frame-buffers* a VRAM de 16 bits/pixel, però permet una resolució de 160*120 pixels.

```
void ham_SetBgXY( u16 bgno,
                  u16 x,
                  u16 y)
```

Assigna l'offset XY a una capa de fons.

Paràmetres:

`Bgno:` El número de fons a desplaçar.
`x:` El nou valor offset X.
`y:` El nou valor offset Y.

Nota: Aquesta funció no funciona amb fons rotable i/o escalable.

• Funcions de servei d'interrupció

Les funcions d'interrupció HAMLib utilitzades en aquest projecte són les següents:

`void ham_StartIntHandler( u8  intno, void * fp)`
Defineix l'esdeveniment d'interrupció i la funció.

`void ham_StopIntHandler( u8  intno)`
Defineix l'esdeveniment d'interrupció que es vol aturar.

```
void ham_StartIntHandler( u8  intno,
                          void * fp)
```

Defineix l'esdeveniment d'interrupció i la funció.

Paràmetres:

Intno: Es la interrupció que es vol inicialitzar. HAM disposa de les següents macros que enumeren el tipus d'interrupció:

- `INT_TYPE_VBL` (invocat quan el scanlines sobrepassa la línia 160)
- `INT_TYPE_HBL` (invocat a cada scanline, quan x passa el valor 240)
- `INT_TYPE_VCNT` (quan l'actual scanline és igual al valor VCNT (Vertical Count)
- `INT_TYPE_TIM0` (Quan acaba de compta el temporitzador(timer) 0)
- `INT_TYPE_TIM1` (Quan acaba de compta el temporitzador(timer) 1)
- `INT_TYPE_TIM2` (Quan acaba de compta el temporitzador(timer) 2)
- `INT_TYPE_TIM3` (Quan acaba de compta el temporitzador(timer) 3)
- `INT_TYPE_SIO` (Quan es troba una E/S serie)
- `INT_TYPE_DMA0` (Quan el DMA 0 ha acabat de transferir)
- `INT_TYPE_DMA1` (Quan el DMA 1 ha acabat de transferir)
- `INT_TYPE_DMA3` (Quan el DMA 2 ha acabat de transferir)
- `INT_TYPE_DMA3` (Quan el DMA 3 ha acabat de transferir)
- `INT_TYPE_KEY` (Quan el s'ha premut una tecla)
- `INT_TYPE_CART` (Quan s'ha inserit o s'ha tret un cartutx a la GBA)

fp: un punter a la funció que es vol invocar quan esdevé la interrupció.

```
void ham_StartIntHandler( u8  intno,
                          void * fp)
```

Defineix l'esdeveniment d'interrupció que es vol aturar.

Paràmetres:

Intno: Quin tipus d'interrupció que es vol aturar. HAM disposa de les següents macros que enumeren el motiu d'interrupció:

- `INT_TYPE_VBL` (invocat quan el scanlines sobrepassa la línia 160)
- `INT_TYPE_HBL` (invocat a cada scanline, quan x passa el valor 240)
- `INT_TYPE_VCNT` (quan l'actual scanline és igual al valor VCNT (Vertical Count)
- `INT_TYPE_TIM0` (Quan acaba de compta el temporitzador(timer) 0)
- `INT_TYPE_TIM1` (Quan acaba de compta el temporitzador(timer) 1)
- `INT_TYPE_TIM2` (Quan acaba de compta el temporitzador(timer) 2)
- `INT_TYPE_TIM3` (Quan acaba de compta el temporitzador(timer) 3)
- `INT_TYPE_SIO` (Quan es troba una E/S serie)
- `INT_TYPE_DMA0` (Quan el DMA 0 ha acabat de transferir)
- `INT_TYPE_DMA1` (Quan el DMA 1 ha acabat de transferir)
- `INT_TYPE_DMA3` (Quan el DMA 2 ha acabat de transferir)
- `INT_TYPE_DMA3` (Quan el DMA 3 ha acabat de transferir)
- `INT_TYPE_KEY` (Quan el s'ha premut una tecla)
- `INT_TYPE_CART` (Quan un cartutx s'ha inserit o s'ha tret a la GBA)

• Funcions de manipulació de paleta

Les funcions de paleta HAMLib utilitzades en aquest projecte són les següents:

`void ham_SetBgPalColRGB(u8 color_number, u8 r, u8 g, u8 b)`
Inicialitza un color RGB a una posició de paleta indicada.

`void ham_LoadBGPal(void *src, u16 size_u16s)`
Carrega la paleta de fons amb 256 nous colors.

---

void ham_LoadBGPal(void *src, u16 size_u16s)

Carrega la paleta de fons amb 256 nous colors.

Paràmetres:

Src: Punter dels colors de la paleta (els colors necessiten estar definits en 15 bits en format BGR - 5 bits blau, 5 bits verd i 5 bits vermell -). Veure secció 2.4.3.

size_u16s: El numero de colors a ser carregats des de el color 0. (per carregar una paleta sencera, posar 256).

```
void ham_SetBgPalColRGB  (  u8  color_number,
                            u8    r,
                            u8    g,
                            u8    b )
```

Inicialitza un color RGB a una posició de paleta especificada.

Paràmetres:

color_number: El numero de color que es vol inicialitzar.

R: La intensitat de vermell desitjada (0-255).

G:: La intensitat de verd desitjada (0-255).

B: La intensitat de blau desitjada (0-255).

• Funcions mode mapa de bits

Les funcions de mapa de bits HAMLib utilitzades en aquest projecte són les següents:

```
void ham_SetBgMode(u8  bgmode  )
```
El mode gràfic a inicialitzar (0-5).

```
void ham_ClearBackBuffer(u32 col_value)
```
Neteja el búfer de treball amb el color especificat.

```
void ham_PutPixel (u32 x, u32 y, u32 col_value)
```
Escriu un pixel en modes de mapa de bits.

```
void ham_FlipBGBuffer)
```
Intercanvia el búfer de frame per el búfer de treball.

```
u8   ham_GetBGBuffer()
```
Entrega l'actual búfer de treball. (només en modes 4 i 5).

---

void ham_SetBgMode(u8 bgmode)

Inicialitza un nou mode gràfic a la GBA. Tota la renderització després d'inicialitzar el mode serà portat en aquest nou mode.

Paràmetres:

Bgmode: El mode gràfic a inicialitzar (0-5)

0 = Mode Rajola, 4 fons disponibles (0-3) sense rotació ni translació.
1 = Mode Rajola, 3 fons disponibles (0-2). Els fons 0-1 no permeten ser rotats ni escalats, mentre que el fons 2 si.
2 = Mode Rajola, 2 fons disponibles (2-3), ambdós transformables.
3 = Mode Bitmap, un *buffer-frame* a VRAM de 16bits/pixel.
4 = Mode Bitmap, dos *buffer-frame* a VRAM, de 8 bits/pixel
5 = Mode Bitmap, dos *buffer-frame* a VRAM de 16 bits/pixel, però permet una resolució de 160*120 pixels.

---

void ham_ClearBackBuffer(u32 col_value)

Neteja el búfer de treball amb el color especificat.

Paràmetres:

col_value: El valor de color amb el que es vol netejar el búfer de treball .

void ham_PutPixel (u32 x, u32 y, u32 col_value)

Pinta un pixel en modes BitMap en el búfer de treball quan es treballa en modes 4 o 5, i directament a la pantalla, en mode 3.

Paràmetres:

X: Posició X del pixel a ser pintat.
Y: Posició Y del pixel a ser pintat.
col_value: Valor del color del pixel (Pot ser una referència a paleta o un valor RGB)

---

void ham_FlipBGBuffer()

Intercanvia el búfer de treball per el búfer que s'està visualitzant, de manera immediata. Aquesta funció només és apta per modes BitMap (Mapa de Bits) que permeten tenir 2 búfers a VRAM, o sigui, només pels modes 4 i 5.

---

u8 ham_GetBGBuffer()

Retorna el búfer de treball actiu (només en modes 4 i 5).

Aquesta funció retorna:

- **0** si el búfer actiu és el 0. Significa que el búfer actiu a la VRAM comença a la posició 0x06000000 i totes les funcions que treballen en mode BitMap operen sobre el búfer de treball a la posició 0x0600A000.

- **1** si el búfer actiu és el 1. Significa que el búfer actiu a la VRAM comença a la posició 0x0600A000 i totes les funcions que treballen en mode BitMap operen sobre el búfer de treball a la posició 0x06000000.

• Funcions mode rajola

Les funcions de mode rajola HAMLib utilitzades en aquest projecte són les següents:

```
tile_info_ptr ham_InitTileSet(void *src, u16 size_u16, u8 col_mode, u8 cbb_only_mode)
```

Inicialitza un conjunt de rajoles i carrega les dades gràfiques a VRAM.

```
map_fragment_info_ptr ham_InitMapFragment(void *src,
                                          u16 map_total_x, u16 map_total_y,
                                          u16 map_ofs_x,   u16 map_ofs_x,
                                           u8 map_tiles_x,  u8 map_tiles_y,
                                           u8 map_rot)
```

Inicialitza un fragment de mapa (un rectangle amb un mapa)

```
void ham_InsertMapFragment(map_fragment_info_ptr mfi, u8 bgno, u8 x, u8 y)
```

Insereix un fragment de mapa (un rectangle amb un mapa) a la regió de mapa.

```
Map_info_ptr ham_InitMapEmptySet(u8 map_size, u8 map_rot)
```

Inicialitza un mapa buit a la VRAM.

Tile_info_ptr ham_InitTileSet(void *src, u16 size_u16, u8 col_mode, u8 cbb_only_mode)

Inicialitza un conjunt de rajoles i carrega les dades gràfiques a VRAM. La funció carregarà un conjunt de rajoles (normalment des del cartutx de memòria ROM) des de un punter i es copiarà dins un punt de VRAM apropiat. El punt es decidit pel gestor de memòria, i es retornarà en una estructura.

Paràmetres:

Src: Punter font de la informació gràfica de les rajoles.
Size_u16: Grandària de les rajoles a copiades a la memòria RAM de rajoles, en un numero de paquets de 16 bits.
col_mode: El mode de color del conjunt de rajoles.

0 = 16 colors/16 paletes
1 = 256 colors / 1 paleta

cbb_only_mode: El valor d'aquest paràmetre requereix entendre una mica el hardware de la GBA. Bàsicament, si es col·loca la informació gràfica de les rajoles a memòria externa (EWRAM), s'hauràn de col·locar al començament del Bloc Base de Caràcter (CBB) en l'ordre de permetre l'accés a la rajola nº 0 amb una entrada de mapa 0. Si el bloc reservat no és directament el començament del CBB, s'haurà de tenir l'offset de les entrades del mapa. En cas de dubte, i si es té memòria VRAM disponible, es recomana posar el valor 1.

0 = reserva al punt més pròxim, no és segur si no es coneix les entrades de mapa.
1 = reserva només al començament de un CBB per assegurar que la rajola 0 comença a l'índex numero 0 (valor recomanat per HAM en qualsevol cas)

Retorna:

Un punter a la nova estructura de tipus *tile_info*, on aquesta contindrà tota la informació necessària per aconseguir la inicialització de fons.

```
map_fragment_info_ptr ham_InitMapFragment(    void *src,
            u16 map_total_x,
        u16 map_total_y,
                                                u16 map_ofs_x,
                                                u16 map_ofs_x,
                                                u8 map_tiles_x,
                                                u8 map_tiles_y,
                                                u8 map_rot
                                            )
```

Inicialitza un fragment de mapa (un rectangle amb un Mapa). Aquesta funció reservarà una nova estructura que podrà ser utilitzada per fer una referència a la informació de mapa (normalment dins de ROM) per copiar en un rectangle el mapa de pantalla actual.

Paràmetres:

Src: Un punter a la informació font del mapa que volem per crear el fragment de mapa des de, normalment, memòria ROM.

map_total_x: L'amplada del mapa (mesurat en rajoles).

map_total_y: L'altura del mapa (mesurat en rajoles).

map_ofs_x: L'offset horitzontal inicial del mapa que es col començar a copiar (mesurat en rajoles)

map_ofs_y: L'offset vertical inicial del mapa que es col començar a copiar (mesurat en rajoles)

map_tiles_x: El numero de rajoles (des de l'offset horitzontal) que es vol especificar horitzontalment.

map_tiles_y: El numero de rajoles (des de l'offset vertical) que es vol especificar verticalment.

map_rot: S'especifica si el mapa és transformable (0 = no transformable 1 = Transformable)

Retorna:

Un punter a una nova estructura de tipus *map_fragment_info* que conté tota la informació necessària per operacions posteriors.

```
void ham_InsertMapFragment(map_fragment_info_ptr mfi,
                                    u8 bgno, u8 x, u8 y)
```

Insereix un fragment de mapa (un rectangle amb el mapa) al mapa de pantalla.

Paràmetres:

mfi: Al informació de fragment de mapa prèviament creat per la funció ***ham_InitMapFragment()***.

Bgno: El numero de fons on es vol inserir el fragment de mapa (0-3).

X: Pla posició X (mesurat en rajoles) del fons de mapa on es vol col·locar el fragment de mapa.

Y: Pla posició Y (mesurat en rajoles) del fons de mapa on es vol col·locar el fragment de mapa.

```
Map_info_ptr ham_InitMapEmptySet(u8 map_size,
                                 u8 map_rot)
```

Inicialitza un mapa buit a la VRAM de la GBA.

Paràmetres:

Map_size: La grandària del fons de mapa (0-3). Aquest depèn del paràmetre *map_rot*.

- Si *map_rot* és 0:
  - ◊ 0 = 32*32 rajoles (mapa de 256*256 pixels)
  - ◊ 1 = 64*32 rajoles (mapa de 512*256 pixels)
  - ◊ 2 = 32*64 rajoles (mapa de 256*512 pixels)
  - ◊ 3 = 64*64 rajoles (mapa de 512*512 pixels)
- Si *mapa_rot* és 1:
  - ◊ 0 = 16*16 rajoles (mapa de 128*128 pixels)
  - ◊ 1 = 32*32 rajoles (mapa de 256*256 pixels)
  - ◊ 2 = 64*64 rajoles (mapa de 512*512 pixels)
  - ◊ 3 = 128*128 rajoles (mapa de 1024*1024 pixels)

map_rot: S'especifica si el mapa és rotable o no (0 = Mapa no rotable, 1 = Mapa rotable)

Retorna:

Un punter a una nova estructura de tipus *map_info* que conté tota la informació necessària per aconseguir la correcta inicialització de mapa.

• Funcions de finestra

Les funcions de finestra HAMLib utilitzades en aquest projecte són les següents:

void ham_CreateWin(u32 wino, u32 x1, u32 y1, u32 x2, u32 y2, u32 dins_de, u32 fora_de, u32 fx_on)

Crea una nova finestra GBA.

```
void ham_CreateWin(u8 winno, u32 x1, u32 x2,
                        u32 x2, u32 y2, u32 dins_de,
                        u32 fora_de, u32 fx_on)
```

Crea una nova finestra GBA. Aquesta funció es prou ràpida per ser actualitzada durant la interrupció HBL, permetent així, redimensionar la forma de finestra mentre la pantalla s'està construint.

Paràmetres:

Winno: El numero de finestra a inicialitzar i visualitzar. 0 i 1 són finestres estàndard que poden utilitzar-se a qualsevol secció de la pantalla amb les coordenades x i y, winno 2 és la finestra d'objecte.

x1: Cantonada x esquerre superior de la finestra.
y1: Cantonada y esquerre superior de la finestra.
x2: Cantonada x esquerre inferior de la finestra.
y2: Cantonada y esquerre inferior de la finestra.
x1: Cantonada x esquerre superior de la finestra.
y1: Cantonada y esquerre superior de la finestra.

dins_de: Especifica quins fons o objectes han de ser vistos dins la finestra. Es pot utilitzar qualsevol combinació dels següents valors (concatenat amb el símbol "|" -or lògica-)

- WIN_BG0
- WIN_BG1
- WIN_BG2
- WIN_BG3
- WIN_OBJ

fora_de: Especifica quins fons o objectes han de ser vistos fora de la finestra. Es pot utilitzar qualsevol combinació dels següents valors (concatenat amb el símbol "|" -or lògica-)

- WIN_BG0
- WIN_BG1
- WIN_BG2
- WIN_BG3
- WIN_OBJ

fx_on: Habilita la utilització d'un efecte especial en aquesta finestra.

HAM té les següents definicions per cada finestra: La grandària del fons de mapa (0-3). Aquest depèn del paràmetre *map_rot*.

• Controls de la GBA

Ham disposa d'unes macros C que permeten controlar els esdeveniments d'usuari que efectua pels controls de la GBA. Aquestes macros retornen l'estat actual del control: 1 si s'ha premut el control, 0 altrament.

BUTÓ	NOM MACRO HAM
AMUNT	*F_CTRLINPUT_UP_PRESSED*
AVALL	*F_CTRLINPUT_DOWN_PRESSED*
ESQUERRA	*F_CTRLINPUT_LEFT_PRESSED*
DRETA	*F_CTRLINPUT_RIGHT_PRESSED*
START	*F_CTRLINPUT_START_PRESSED*
SELECT	*F_CTRLINPUT_SELECT_PRESSED*
A	*F_CTRLINPUT_A_PRESSED*
B	*F_CTRLINPUT_B_PRESSED*
L	*F_CTRLINPUT_L_PRESSED*
R	*F_CTRLINPUT_R_PRESSED*

Així, amb una simple comparació es sabrà si s'ha premut la tecla.

Exemple,

```
if(F_CTRLINPUT_UP_PRESSED)
{
        // S'ha apretat la tecla amunt...
}
```

A2.1.3 L'eina GFX2GBA

Aquesta eina converteix fitxers PCX,BMP,TGA,SPR i TIM de 8 bit a informació utilitzable per la GBA (format cru, format rajola, format de mapa, etc.). Quan es passa el fitxer gràfic a convertir per l'aplicació GFX2GBA, aquesta crea dos fitxers, amb algun dels formats anomenats anteriorment. En un fitxer comprendrà la informació de pixels i en l'altre la informació de paleta. Com que treballem en mapa de bits, necessitem que aquesta aplicació converteixi la informació gràfica de "textures.bmp" en format cru (anomenat en anglès *raw data*). Tanmateix, existeix la manera de donar el format cru codificat en un vector C unidimensional, si s'afegeix l'opció `-fscr`. Nosaltres farem servir aquesta opció.

Per exemple, si volem convertir un fitxer gràfic anomenat "textura1.bmp" el qual té una dimensió de 64 x 64 pixels, executaríem aquesta eina amb les opcions següents,

```
gfx2gba -fsrc –pTextura1.pal Textura1.bmp
```

On:

`-pTextura1.pal` : és el nom del fitxer de sortida amb la informació crua de paleta d'aquest fitxer gràfic.

Això ens crearia dos fitxers: "**Textura1.pal.c**" i "**Textura1.raw.c**".

- **"Textura1.pal.c"** inclou la informació de la paleta en un vector C unidimensional de 256 valors constants de 16 bits anomenada **"Textura1_Palette"**.
- **"Textura1.raw.c"** inclou la informació dels pixels del fitxer un vector C unidimensional de 4096 valors constants de 8 bits, anomenada **"Textura1_Bitmap"**.

Aquests dos fitxers creats són afegits a un fitxer C anomenat `"Textura.c"` mitjançant la directiva `#include`. Així doncs, tindrem els dos vectors de forma global. Al fitxer `"Textura.h"` s'hi declararà la referència externa d'aquests vectors, i així, poder accedir-hi a qualsevol punt d'execució.

A la següent pàgina mostra un exemple de conversió.

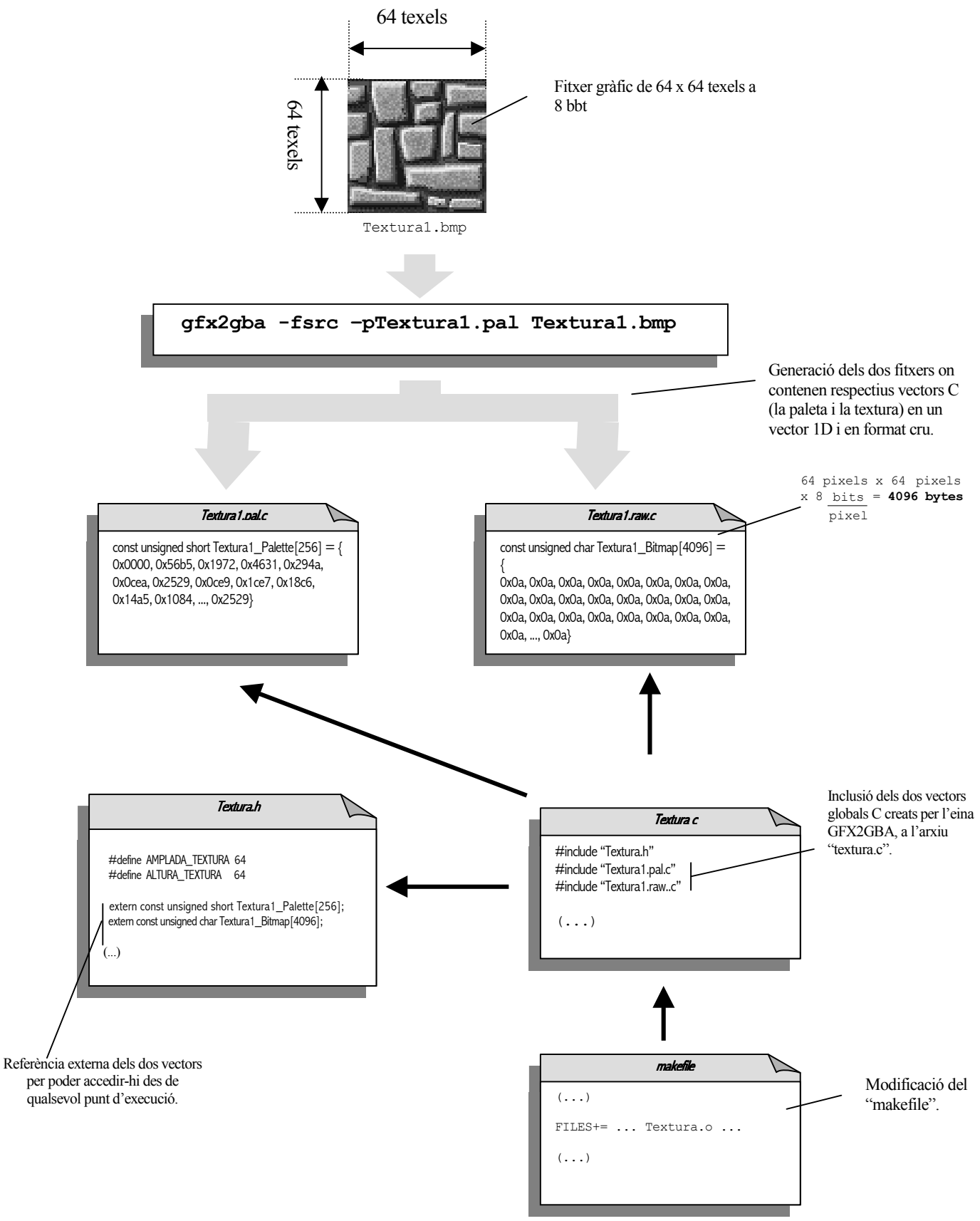


Figura A2.2 *Vista gràfica del procés de conversió del fitxer gràfic "textura1.bmp" en un format manejable a la GBA.*

El llistat A2.1 mostra la substitució de la paleta de sistema pel de textura, retornada per l'eina GFX2GBA. Això s'ha de fer perquè els texels de textura anteriorment convertida corresponguin amb els seus colors originals.

```
#include "myGBA.h"
#include "Textura.h"  // En aquí tenim les referència externa de la paleta...

void main()
{
   ham_init();

   ham_SetBGMode(4);

   // Inicialitzem la paleta segons els valors de la taula Textura1_palette ...

   ham_LoadBGPal((u16 *)Textura1_Palette,SIZEOF_16BIT(Textura1_Palette));

   (...)


}
```

Llistat A2.1

I ja ho tenim tot apunt per tractar les textures dins el codi.

A2.1.3 Eines de temps

HAM incorpora una eina a temps d'execució que pot mesurar el cost de temps d'un tros de codi mitjançant l'ús de tres macros, anomenades: *TOOL_PROFILER_INIT*, *TOOL_PROFILER_START* i *TOOL_PROFILER_END*. Per inicialitzar aquesta eina de temps, cal declarar la macro *TOOL_PROFILER_INIT* abans de la funció *main*. Entre les macro *TOOL_PROFILER_START* i *TOOL_PROFILER_END* es col·loca el codi o funció a avaluar. Com que el nostre objectiu és avaluar el cost que té la nostra renderització, la macros *TOOL_PROFILER_START* i *TOOL_PROFILER_END* abans i després respectivament de la funció ***renderitzar()***.

Llavors es podrà consultar el temps transcorregut amb les variables globals *ham_prof_frames_elapsed* i *ham_prof_subticks*, on:

- *ham_prof_frames_elapsed:* Mesura el numero de frames transcorreguts (1 frame = 16.7680 ms).
- *ham_prof_subticks:* Mesura el numero de tics transcorreguts en el frame actual (1 tic = 0.0038 ms).

El següent tall de codi C, explica com mesurar els FPS en un tros de codi, mitjançant la llibreria HAM.

```
#include "myGBA.h"

(...)

TOOL_PROFILER_INIT

int main(int argc, char *argv[])
{
          (...)

         do{
                        (...)

                        TOOL_PROFILER_START

                        { codi a avaluar }

                        TOOL_PROFILER_END

                        Temps = ham_prof_frames_elapsed*16.768f + ham_prof_subticks*0.0038f

                        (...)

         }while(1);

}
```

A2.2 L'eina bin2C v1.0

Descripció

Petita utilitat, per Christian Padovano, que trasllada un fitxer binari a un vector C de tipus caràcter per la finalitat que sigui inclòs a un fitxer C.

Utilització

Des de l'intèrpret de comandes cal invocar l'eina amb els següents paràmetres:

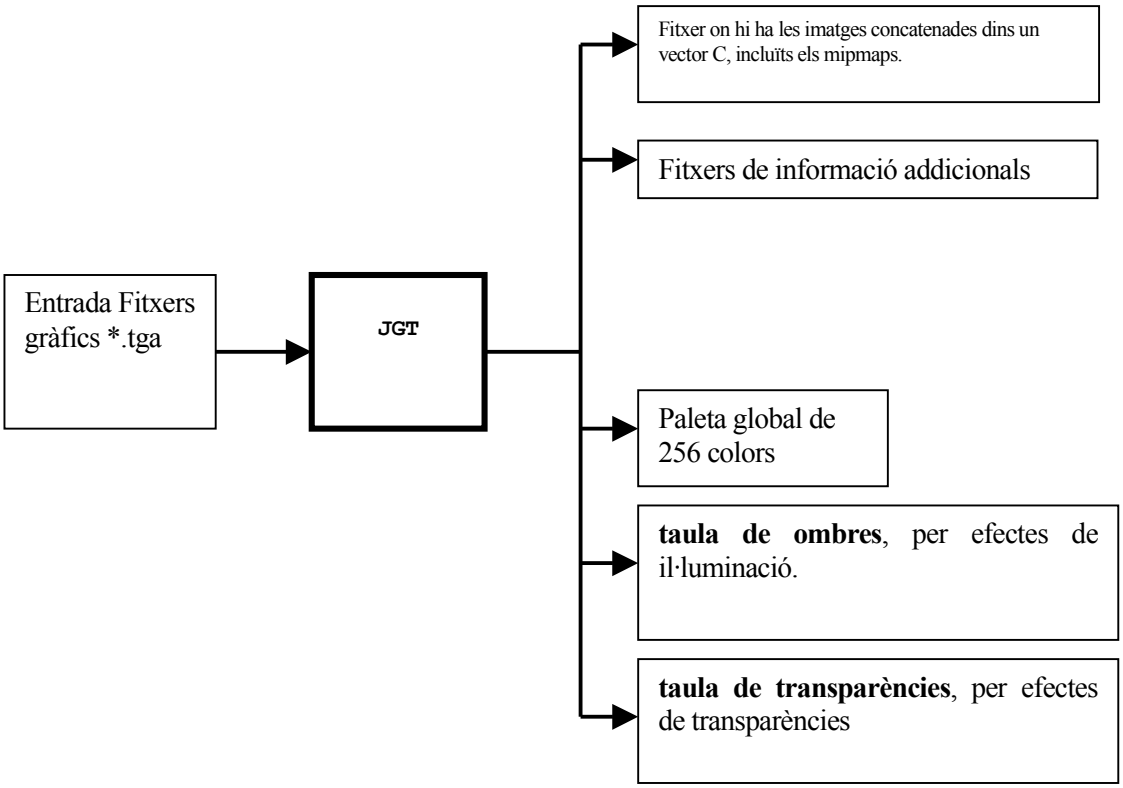
Bin2C <nom_fitxer_binari> <nom_fitxer_desti>

Nota: En el fitxer destí no cal posar l'extensió .c, l'eina ja l'afegeix.

A2.3 L'eina JGT v1.3

A2.3.1 Descripció

Jespa Generate Tool (JGT) versió 1.3, una eina de creació pròpia, prepara les textures de alta profunditat de color per que poguin treballar sota la plataforma GBA per un sistema de video de 8 bpp (mode paleta). Les imatges han de provenir d'un format Targa (tga) a 24 bpp. L'eina té com objectiu principal concatenar la informació gràfica crua en un fitxer dins d'un vector C. JGT també genera, taules de il·luminació i de transparència del 50% de barreja (entre el fons i imatge a pintar), en base la paleta global creada. Aquesta eina també es capaç de generar MIMAPs per cada imatge, molt útil per evitar aliasing en renderitzacions 3D (veure annex 6 per més informació sobre el filtre MIPMAP).



Les característiques tècniques de l'eina JGT són les següents:

- Generació de MIPMAPS al màxim permès per les dimensions originals de la imatge.
- Entrada de fitxers TGA de 24 bpp sense comprimir.
- Quantifica qualsevol numero de fitxers TGA a una paleta única
- Genera la taula de ombres
- Genera la taula de transparències
- Genera fitxers de informació addicional de les imatges processades.
- L'eina JGT esta pensada per fer la lectura de imatges per sistemes que no disposin de gestor de fitxers com la consola de videojocs GBA.
- Detecció de imatges tipus sprite si el color de transparència és un rosa pur (256,0,256).

A2.3.2 Utilització

Per utilitza l'eina JGT, cal posar el següent a l'intèrpret de comandes:

```
JGT
```

L'eina llegirà el fitxer de configuració imprescindible per la seva execució *GFX_Generate.cfg*.

A2.3.3 El fitxer de configuració (*GFX_Generate.cfg*)

Aquí s'entra el nom amb la seva ruta relativa/absoluta dels fitxers gràfics a convertir. A part, s'hi poden entrar quines propietats tindrà aquest gràfic dins el nostre programa. Les propietats poden ser les següents:

- WITH_MIPMAPS: Donar la possibilitat de tenor la imatge amb mipmaps.
- IS_PANORAMA: S'informa que aquesta imatge és de tipus panoràmica.

El format per carregar una imatge dins de *GFX_Genereate.cfg* és:

```
nom_imatge.tga, propietat1, propietat2, ... etc
```

A2.3.4 Fitxers generats per JGT

Els fitxers que genera el JGT bàsicament són mòduls i capçaleres C, preparats per ser inclosos al makefile del compilador. Els fitxers que genera els següents:

- images.raw.c: Imatges en format cru, qualitatitzades a 8 bits. També hi inclou els seus MIPMAPs (si en té), totes en únic vector C unidimensional anomenat images.
- Paleta.raw.c: Conté la informació de la paleta global de 256 colors.
- Info_images.h: Capçalera del mòdul info_images.c, en aquí conté definicions necessàries pel seu mòdul.
- Info_Images.c: Conté una taula de informació de les imatges que s'han entrat, per ordre.
- Shades.raw.c: Conté informació de la taula de ombres.
- Glenz.raw.c: Conté informació de la taula de transparència.

A2.3.5 Carrega i gestió dels fitxers gràfics JGT

Havent explicat la utilització d'aquesta eina, interessa saber la manera de carregarà la informació que entrega l'eina JGT. Per això, en aquest annex es donarà un exemple pràctic pel cas de convertir 3 imatges següents:

Figura A2.5a
imatge "Paret.tga"

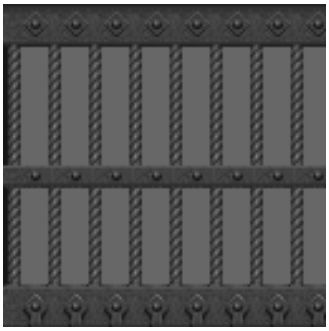


Figura A2.5*b* *imatge de tipus sprite "valla.tga"*

Figura A2.5c *Imatge "p2.tga" de tipus panoramic de 1024x256*

La imatge paret.tga la volem amb mipmaps, valla.tga sense mipmaps i notificarem a l'eina JGT que la imatge de la figura A2.5a es tracta d'un paisatge.

JGT.cfg

```
Paret.tga, WITH_MIPMAPS
Valla.tga, WITHOUT_MIPMAPS
Paisatge.tga, IS_PANORAMA
```

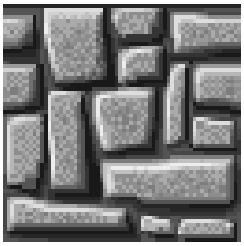
Veiem amb detall contingut dels fitxers resultants i comentem els aspectes particulars de cadascun d'ells.

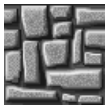
• El fitxer Images.raw.c

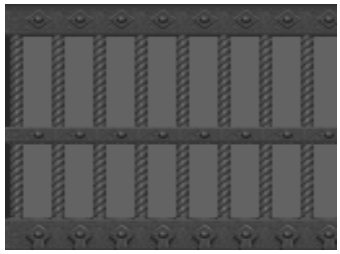
Al fitxer gràfic *images.raw.c* representa la informació crua de la imatges *tga*. A continuació, hi conté la informació dels mipmaps (si escau). Llavors ve la següent imatge amb els seus mipmaps respectiu etc. Tota aquesta informació en un únic vector C anomenat images[].

Images.raw.c

```
unsigned char images[]={

// Imatge original "paret.tga" (64x64)

0x0, 0x0, 0xa, ...
.
.
.
.
                ...     , 0xc, 0xa, 0xb,
// 1er mipmap "paret.tga" (32x32)

0x0, 0xa, 0xb, ...
.
.
.
                ...     , 0xc, 0xa, 0xb,
// 2on mipmap "paret.tga" (16x16)

.
.
.
.
.
.
// 6er mipmap "paret.tga" (1x1)

0xb,

// Imatge "valla.tga" (128x128)

0x0, 0xb, 0xc,...,
.
.
.
.
.
.
                ...     , 0xc, 0xa, 0xb
// Imatge panoramica "p2.tga" (1024x512)

0x0, 0xb, 0xc,...,
.
.
.
                ...     , 0xc, 0xa, 0xb};
```







- ## El fitxer paleta.raw.c

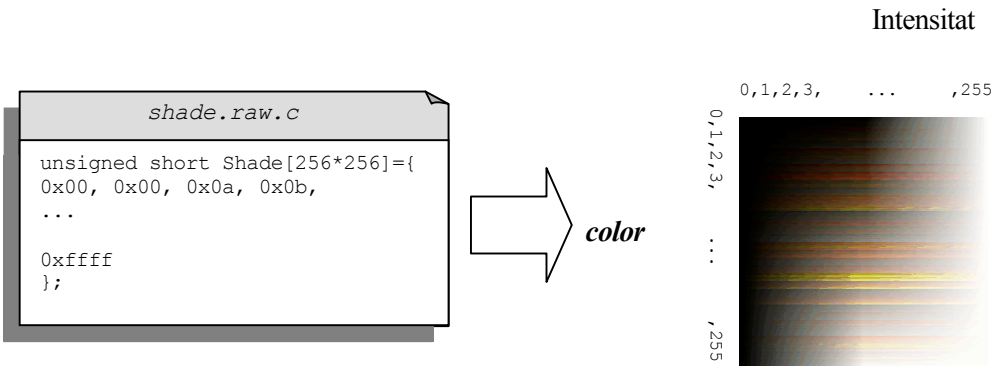
En aquest mòdul inclou la paleta global de 256 colors de 16 bits cada color en format cru dins un vector C, cadascun acord el sistema de vídeo mode 4 de la GBA.

paleta.raw.c

```
unsigned short paleta[256]={
0x0000, 0x0042, 0x00cc, 0x00ff,
...

0xffff
};
```

- ## El fitxer shade.raw.c

En aquest mòdul inclou una taula de ombres en un vector C, el contingut són índexs de paleta referents al color aproximat segons el color i la intensitat seleccionada als valors [0, 256] (veure annex a6 per més informació sobre la construcció d'una taula de ombres).

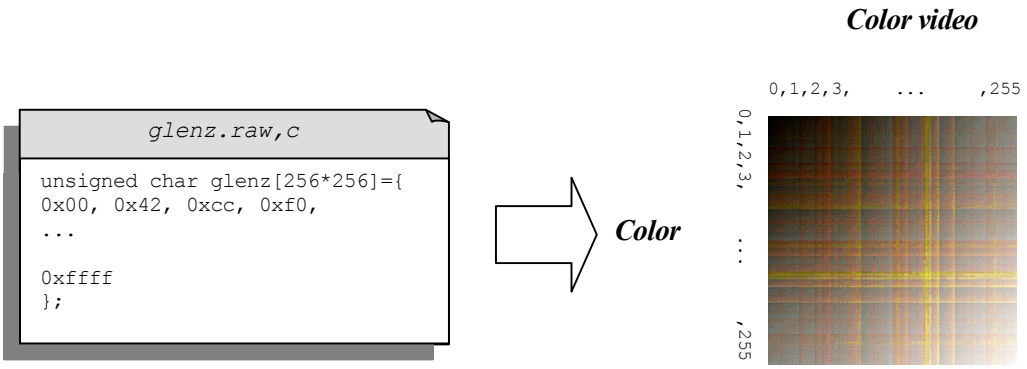


Per seleccionar el índex de paleta segons un valor de color, intensitat cal fer l'offset lineal següent:

```
Index_color_amb_intensitat ← color*256 + intensitat
```

- **El fitxer glenz.raw.c**

En aquest mòdul inclou una taula de transparència del 50 % en un vector C, el contingut són índexs de paleta referents al color transparent aproximat al 50% de barreja entre la imatge a pintar i el color de fons llegit [0, 256] (veure annex 6 per més informació sobre la construcció d'una taula de transparències).



Per escriure el color de paleta transparent al valor de color llegit de fons (color a la memòria de vídeo) cal fer el següent offset lineal:

```
index_color_transparent ← color *256 + color_video
```

- **El fitxer *info_images.c* i *info_images.h***

El fitxers info_images.c i info_images.h són imprescindibles per tal de fer la gestió dels fitxers anteriors. Entra altres coses inclou:

- Referència índex paleta referent al color de màscara per no escriure pixels que tinguin aquest índex de paleta quan pintem imatges tipus sprite.

- Referència índex paleta dels colors blau, verd, groc entre altres colors. Naturalment, seran colors aproximats dels 256 que hi ha a la paleta.

- Estructures, declaració i inicialització de tipus necessaris per la inicialització i mostra de informació de imatges.

- Es genera la mateixa informació continguda en info_imatges.h a format arm (.s) per poder llegir les característiques de les imatges des de ensamblador ARM.

Entre les més importants destaquen el vector de informació de imatges que esta associada a la següent estructura,

```
typedef struct{

  int n_image;
  int width;
  int height;
  int N_MipMaps;
  int Propietries;
  char name[50];

}tInfo_Image;
```

n_image: Numero d'imatge.
width: Amplada de la imatge.
height: Altura de la imatge.
N_MipMaps: Numero de mipmaps que disposa aquesta imatge.
Propietries: Propietats de la imatge, codificades bit a bit, de les quals destaquem el significat:

```
Bit 0   : Indiferent.
Bit 1   : Si activat indica que és de tipus panoràmic.
Bit 2   : Indiferent.
Bit 3   : Si activat indica que és de tipus sprite.
Bit 4   : Si activat indica que la amplada és potència de 2.
Bit 5   : Si activat indica que la altura és potència de 2.
Bit 6-31: Indiferent.
```

name: Nom del fitxer imatge.

que al mòdul *info_images.c* inicialitzarà la informació segons les tres imatges entrades de la següent forma,

```
const tInfo_Image Info_Image[N_IMAGES]={

//  ID_Image     width   height       MipMaps    Propietries        Name
//  --------    ------   -------     --------    ------------     ----------
      0,          0,        0,         0,        0x0,            "invalid_image",
      1,         64,       64,         4,        0x34,           "paret.tga",
      2,        128,      128,         1,        0x38,           "valla.tga",
      3,       1024,      256,         1,        0x32,           "p2.tga",
};
```

Podem observar que el numero de imatge 0 és una referència a una imatge invàlida. Això vol dir que associem qualsevol mapeig amb aquesta imatge no s'haurà de pintar res.

- ## Estructuració de la informació

Amb la informació adquirida del mòdul ho tenim tot preparat per mostrar imatges en qualsevol moment, però es obvi que necessitem una prèvia estructuració de la informació ja que, de no fer-ho, seria enfarragos i costos accedir a la informació de cada pixel de imatge mitjançant el vector *images[]*.

Primer de tot ens definirem una estructura mínima imprescindible que defineix la estructura bàsica d'un bloc de imatge,

```
typedef struct{
        char      *ptr_pixels;
        int        DimX;
        int        DimY;
 }tImatge;
```

I ara ens definim el tipus textura, que tindrà els següents camps,

```
typedef struct {
       tImatge       *GFX_MipMap;
       int            N_MipMaps;
       int            Propietats;
}tTextura;
```

Per últim la inicialització d'aquests camps de forma automàtica mitjançant el bucle següent,

```
tTextura Textura[N_IMAGES];

 ...

 ptr_texels = (char *)images;

 // Per cada imatge...
 for(k = 0; k < N_IMAGES; k++)
 {
        DimX = Info_Image[k].width;
        DimY = Info_Image[k].height;


          // Reservem memoria pel numero de mipmaps que suporta l'actual imatge
          // Es compte també, la imatge original com mipmap de nivell 0.
          Reservar_Memoria(Textura[k].GFX_MipMap, tImatge, Info_Image[k].N_MipMaps+1);

          if(Textura[k].GFX_MipMap == NULL)
          {
                      return FALS;
          }


        // Numero de mipmaps...
        Textura[k].N_MipMaps = Info_Image[k].N_MipMaps;



        // Carreguem propietats...
        Textura[k].Propietats = Info_Image[k].Propietries;


        // Per cada MIPMAP de textura...
        for(j = 0; j < Info_Image[k].N_MipMaps; j++)
        {


                Textura[k].GFX_MipMap[j].ptr_pixels  = ptr_texels;
                Textura[k].GFX_MipMap[j].DimX = DimX >> j;
                Textura[k].GFX_MipMap[j].DimY = DimY >> j;


                tamany_imatge_mipmap = (DimX >> j)*(DimY >> j);
                ptr_texels += tamany_imatge_mipmap;


        }
```

ANNEX 3: “MODE 7” A LA GBA

A3.0 Objectiu

Aquest annex donarà el disseny i la implementació necessària per construir una demostració del “mode 7” a la consola de videojocs *Game Boy Advance* (GBA). El “mode 7” (nom originari de la consola SuperNintendo –SNES-) és una habilitat per hardware que disposa en algunes màquines per simular un pla en perspectiva.

La GBA no incorpora la inicialització del “mode 7” a la seva BIOS, però, basat en els modes rajola transformables de la GBA (modes 1 o 2), és possible inicialitzar l’efecte “mode 7”.

El seguiment de la construcció “mode 7” serà el següent:

- Primerament, s’explicarà l’origen del “mode 7”.

- Llavors, a la secció A3.2 i A3.3, es començarà amb l’estudi dels modes gràfics i registres involucrats en la inicialització/renderització de l’efecte “mode 7” a la GBA.

- Tot seguit (secció A3.4), entrarem en un estudi teòric deduint les formules involucrades en una *renderització* d’un pla 3D per-software.

- A la secció A3.5 i basant-nos amb els dos punts anteriors, s’implementarà en C el “mode 7” per la GBA.

- Per últim, s’explicarà com crear la una demostració “mode 7” a la GBA, mitjançant eines del kit HAM.

Abans de entrar en l’estudi veiem, però, una mica de historia d’aquest mode gràfic.

A3.1 Història

En la dècada dels 90 , Nintendo va treure la continuació la NES, La Super Nintendo (SNES) o en Japó, la Super Famicom (figura A3.0). La SNES tenia una hardware gràfic 2D que permetia rotar i escalar gràfics 2D per-hardware. Els programadors de la SNES van aprofitar aquesta capacitat, configurant el hardware necessari, per tal que a cada *scanline* de pantalla disposes d’un paràmetre de escalat i rotat diferent a l’espai de mapa. Amb aquesta nova configuració va permetre a la SNES, la capacitat de simular un pla amb perspectiva 3D per-hardware; aquest l’anomenaren “**mode 7”**.

Figura a3.0 *Super Nintendo (SNES)*

El “mode 7” podia simular pla amb perspectiva 3D des de qualsevol punt de vista i ser rotada en qualsevol angle. L’efecte “mode 7” disposà a la SNES de fer un passeig 3D pel mapa del món en un joc RPG com, per exemple, el joc *Terra Enigma*. Aquest efecte, també es va estendre bastant en els jocs de carreres com, per exemple, el joc *F-Zero*. *Axelay* també utilitzar el “mode 7”, però sense rotats a l’espai 3D. Per últim, les series *Street Fighter* utilitzava el “mode 7” per crear un efecte dissimulat de desplaçament en el terra.

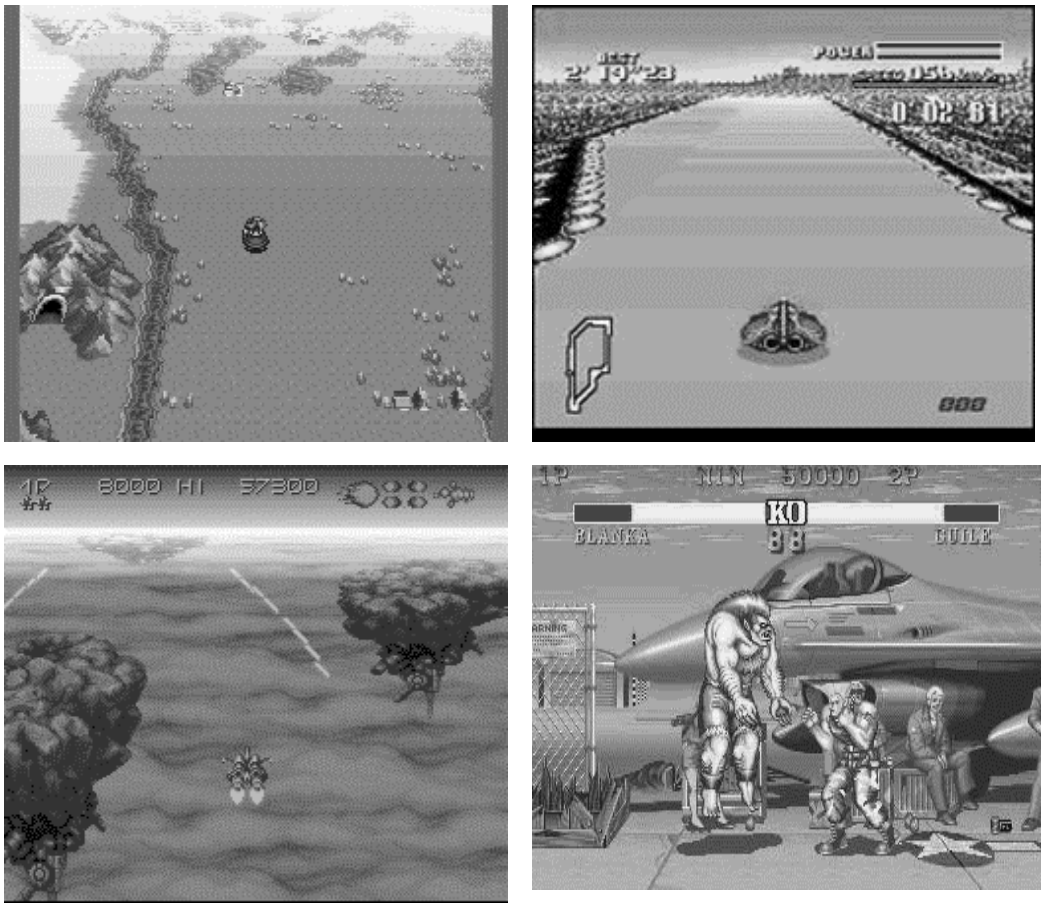


Figura a3.1 *A l’esquerra superior, Terra Enigma, a la dreta superior F-Zero, a la l’esquerra inferior Axelay, per últim, a la dreta inferior Street Figther*

A3.2 Modes rajola a la GBA: mode 0,1 o 2

Quan la GBA inicialitza el mode 0, 1 o 2, la pantalla es subdivideix en quadrats de 8x8 pixels (rajola), on el conjunt d’aquestes rajoles formen el fons. Per la seva inicialització és necessari col·locar les rajoles a la memòria de vídeo i un mapa, que tan sols es un vector (figura A3.3) on cada element es la referència a la textura amb uns quants atributs més. Llavors, el processador de vídeo repassa el vector de mapa (figura A3.3) i construeix la pantalla (figura A3.4) amb les rajoles que li hem especificat (figura A3.2). La grandària del fons pot ocupar des de 256x256 pixels fins a 1024x1024 pixels.

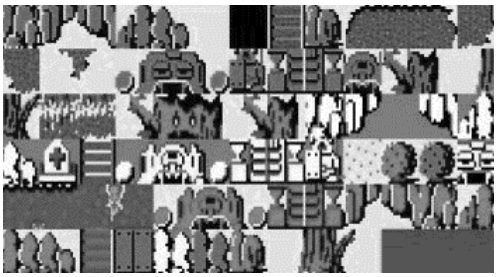


Figura a3.2 *Col·lecció de rajoles guardades com mapes de bits de 8x8 pixels (rajola) a VRAM*

```
Map[] = {
64,64,64,64,64,64,64,6 ,6 ,64,64,64,64,64,64,64,
 64,64,64,64,7 ,64,62,06,06,64,64,64,64,64,64,64,
 64,64,64,62,06,06,06,06,06,64,64,64,64,64,64,64,
 64,64,62,06,06,06,06,06,06,64,64,64,64,64,64,64,
 64,62,06,06,06,06,06,06,06,63,64,64,64,64,64,64,
 06,06,06,06,06,06,06,06,06,06,06,06,06,06,06,06,
 64,67,06,06,06,06,06,06,06,06,06,06,06,06,64,64,
 64,64,06,06,06,06,06,06,06,06,06,06,06,06,64,64,
 64,64,06,06,06,06,06,06,06,06,06,06,06,06,64,64,
 64,64,66,66,66,66,66,66,66,66,66,66,66,66,64,64,
64,64,64,64,64,64,64,64,64,64,64,64,64,64,64,64};
```

Figura a3.3 *El vector de mapa en C. Cada posició del vector fa referència a una textura.*

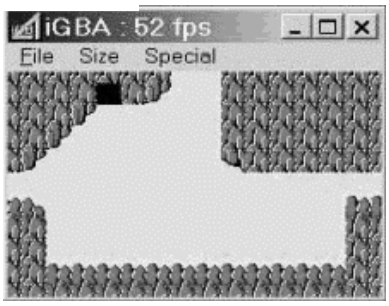


Figura a3.4 *Resultat gràfic, amb les textures de la figura a3.2 i el vector de mapa de la figura a3.4.*

La GBA pot mostrar fins a quatre fons diferents quan opera en mode rajola, on cadascun d’ells es pot controlar independentment utilitzant diferents mapes i textures. Els seus atributs de desplaçament (scroll) també es poden controlar independentment.

Pel que fa als dibuixats de fons, el color 0 es normalment transparent permetent que els fons amb major prioritat es vegin a través dels que tenen menys. Per defecte, el dibuixat dels fons segueix l’ordre 3, 2, 1 i 0; amb el que el fons 0 apareixerà damunt del fons 3, però es pot canviar. Alguns fons també tenen la possibilitat de ser rotats y escalats (és un bon efecte per qualsevol mapa).

Pel que fa a les textures, es poden guardar com mapes de bits de 256 o de 16 colors. Ambdues utilitzen la mateixa paleta però els fons són diferents. Poden utilitzar modes de color diferents. Les paletes de 16 colors funcionen dividint la paleta de 256 colors en 16 paletes de 16 colors cada una. A més, en el mode de 16 colors pots tenir els 256 colors en la pantalla, mentre cada rajola només utilitzi una de les 16 paletes.

Les següents taules resumeixen les característiques de cadascun dels modes rajola.

	MODE 0	MODE 1	MODE 2
Fons Disponible	**Tots**	**0,1,2**	**2,3**
Rotació/Escalat?	**NO**	**Fons 2**	**Fons 2,3**

Taula A3.1 *Modes de Pantalla*

	Numero Màxim de Rajoles	Grandària del mapa (en pixels)	Número de Colors per Rajola	Característiques Especials
Rotació/Escalat	**256**	**128x128, 256x256 512x512, 1024x1024**	**256**	**Rotació/Escalat**
Fons de text pla	**1024**	**256x256, 512x256 256x512, 512x512**	**256 o 16**	**Intercanvi horitzontal/vertical de rajoles**

Taula A3.2 *Característiques de fons*

A3.3 Registres de rotat i/o escalat

Bàsicament, disposem de 4 registres de 16 bits i codificats en una Aritmètica de Punt Fixa (APF) 8.8 que expliquen els paràmetres per la rotació i escalat per un fons rotable i/o escalable. També hi ha 2 registres de 32 bits codificats en una APF 19.8 anomenats punts de referència que especifiquen el centre de rotació/escalat i/o el desplaçament d’un fons rotable i/o escalable.

A3.4.1 REGISTRES DE PUNT DE REFERÈNCIA (REG_BG*x*X i REG_Bg*x*Y)

Per aconseguir un desplaçament d’un mapa rotable i/o escalable, disposem dels registres **REG_BG*x*X** i **REG_BG*x*Y**, on ***x*** és un numero de fons (naturalment rotable i/o esclable). Aquests registres, defineixen la coordenada del mapa que es vol coincidir a la coordenada superior esquerre de pantalla (pixel 0,0) o primer scanline.

Si per exemple, tenim un mapa de 512x512 emmagatzemat a VRAM i inicialitzem ambdós registre REG_BGxX a 272 i el registre REG_BGxY a 352, la pantalla mostraria el següent tros de mapa.

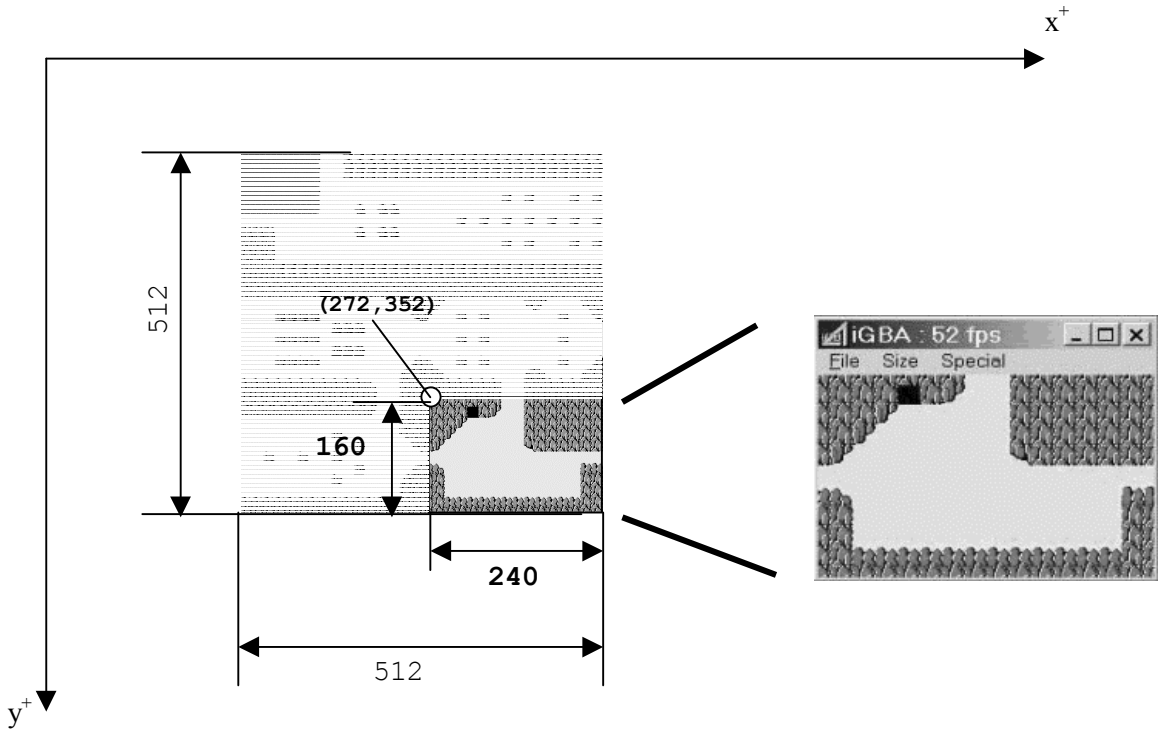


Figura a3.5 *Mapa de 512x512 pixels*

Aquests registres estan codificats a una APF 19.8, o sigui, 19 bits per la part entera i 8 per la part fraccional.

Els registres de referència són copiats als registres interns del processador gràfic durant cada retraç horitzontal (hblank). Els registres interns són llavors incrementats per els registres BGxPB (dmx) i BGxPD (dmy) (que s’explicaran a continuació) després de renderitzar un scanline. Una important cosa a tenir en compte:

Si s’escriu qualsevol registre de punt de referència per software fora del període hblank, s’escriurà un nou valor al corresponent registre intern del processador gràfic. Això vol dir que es podrà escriure un offset XY de mapa diferent per-scanline dins el frame actual.

A3.4.2 REGISTRES DE ROTACIÓ/ESCALAT (BG*x*PA, BG*x*PB, BG*x*PC i BG*x*PD)

El processador gràfic de la GBA, rota/escalar un fons a partir de 4 paràmetres que es passa als següents registres:

BG*x*PA- Paràmetre A de rotació/escalat pel fons número *x* (alies dx)
BG*x*PB- Paràmetre B de rotació/escalat pel fons número *x* (alies dmx)
BG*x*PC- Paràmetre C de rotació/escalat pel fons número *x* (alies dy)
BG*x*PD- Paràmetre D de rotació/escalat pel fons número *x* (alies dmy)

- **Registres BG*x*PA (alies dx) i BG*x*PC (alies dy)**

Quan el processador gràfic de la GBA transforma una línia horitzontal de mapa, dx i dy especifica el pendent i l’augment d’aquesta línia.

Per exemple:

Una línia horitzontal de mapa de longitud 100, amb dx = 1 i dy = 1. El pendent (dy/dx) serà 1, per tant tindrem una resultant línia dibuixada amb una inclinació de 45 º. La longitud de la línia ja no serà 100 (original), sinó que tindrà un augment de `sqrt(100`^{2} `+ 100`^{2}`) = 141.42`, exactament com ho explica el teorema de Pitàgoras ($a^2 + b^2 = c^2$).

- **Registres BG*x*PB (alies dmx) i BG*x*PD (alies dmy)**

De igual forma, quan el processador gràfic de la GBA transforma una línia vertical de mapa, dmx i dmy especifiquen el pendent i l’augment d’aquesta línia.

Si ambdós dmx i dmy configuren el pendent (dx/dy) amb els valors `dmx=-1` i `dmy=1` , amb l’exemple anterior (dx = 1 i dy = 1) significaria rotar 45 º.

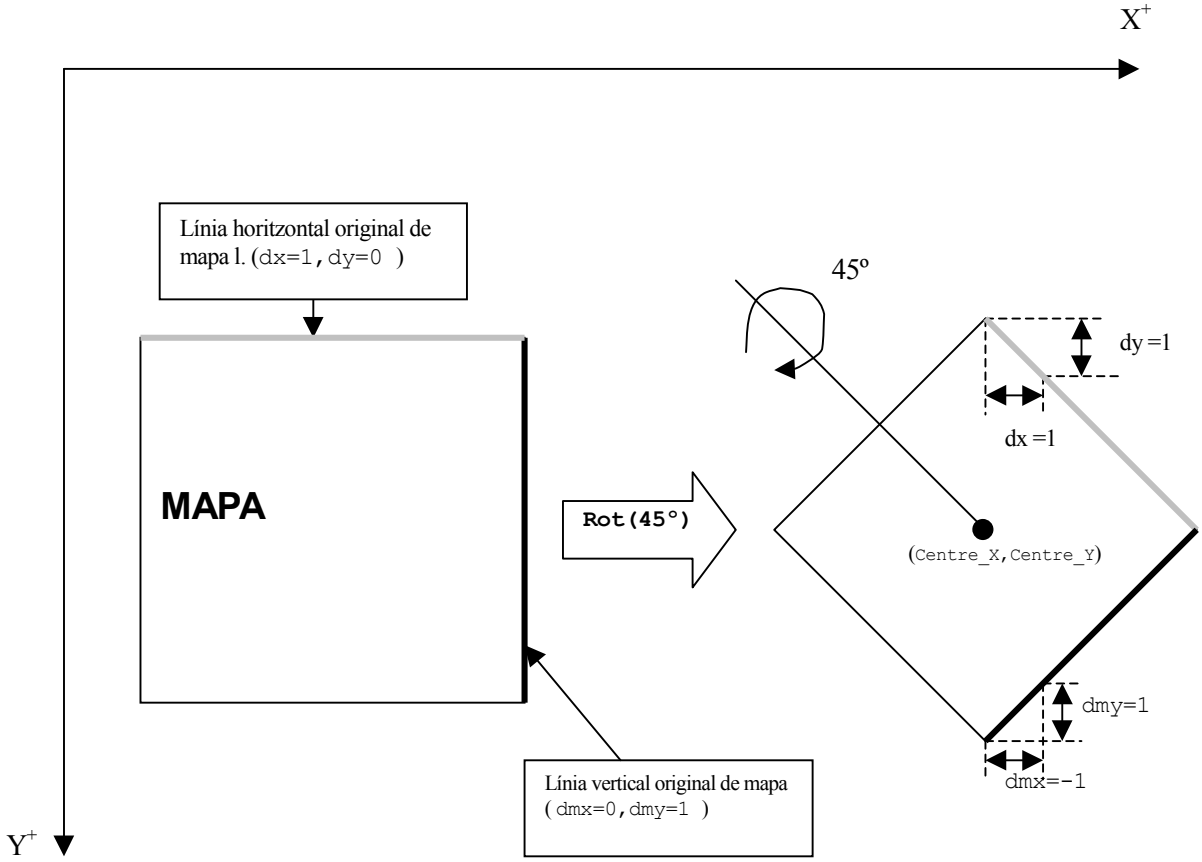


Figura a3.6 *A l’esquerra, mapa original amb el valor de pendent per totes les línies verticals i horitzontals que el representen. A la dreta, mapa rotat 45º, amb la representació del respectius valors de pendent.*

Per rotar un mapa de forma automàtica, podem trobar el respectiu pendent directament per cada registre amb un angle de rotació α amb les següents formules:

```
PA (dx)  =  cos(α)·Escalat_X
PB (dmx) = -sin(α)·Escalat_Y
PC (dy)  =  sin(α)·Escalat_X
PD (dmy) =  cos(α)·Escalat_Y
```

Per que les funcions sinus i cosinus obtinguin el valor adequat, cal que α compleixi el sentit d’orientacions següents:

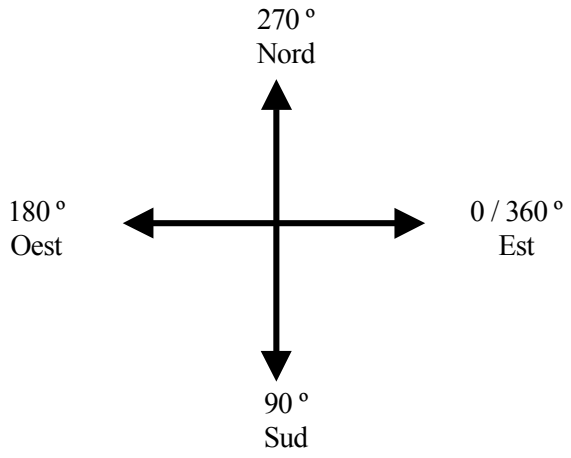


Figura a3.7 *Orientacions de α*

Càlcul del centre de rotació

El centre de rotació es pot calcular aplicant la següent matriu de rotació 2D i el desplaçament del centre rotació:

$$\begin{pmatrix} X_centre_rot \\ Y_centre_rot \end{pmatrix} = \begin{pmatrix} PosX \\ PosY \end{pmatrix} - \begin{pmatrix} X_centre \\ Y_centre \end{pmatrix} \cdot \begin{pmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{pmatrix}$$

$$X_centre_rot = PosX - X_centre \cdot \cos(\alpha) - Y_centre \cdot \sin(\alpha)$$
$$Y_centre_rot = PosY + X_centre \cdot \sin(\alpha) - Y_centre \cdot \cos(\alpha)$$

PosX i *PosY* expliquen la posició X,Y de mapa que es vol coincidir al primer scanline.

X_centre i *Y_centre* defineixen el centre de rotat al mapa respecte el pixel (X_P i Y_P) de pantalla i s’obtenen de la següent forma:

$$\begin{pmatrix} X_centre \\ Y_centre \end{pmatrix} = \begin{pmatrix} X_p & 0 \\ 0 & Y_p \end{pmatrix} \cdot \begin{pmatrix} Escalat_X & 0 \\ 0 & Escalat_Y \end{pmatrix}$$

$$X_centre = X_P \cdot Escalat_X$$
$$Y_centre = Y_P \cdot Escalat_Y$$

Per últim s’assignarà els valors *X_centre_rot* i *Y_centre_rot* als registres de punt de referència. En el cas que treballéssim sobre el fons numero 2, s’assignaria les variables *X_centre_rot* i *Y_centre_rot* als registres ***REG_2X*** i ***BG_2Y***.

A3.4. Estudi de la renderització d’un pla

A3.4.1 La estructura del mapa

La estructura del mapa serà el vector 1D com el vist a la figura A3.2. El mapa és pot veure com una malla 2D (figura A3.8) on a cada cel·la tenim una referència a la rajola que mapeja la part corresponent al mapa.

El mapa representarà el pla XZ, equivalent al mapa XY de la GBA, per on el nostre jugador és mourà (veure secció següent). Com s’ha donat a entendre a la secció anterior, el mode 1 o 2 permet tenir un fons o espai de mapa de fins a 1024x1024 pixels o unitats. Per aquest estudi, cal tenir un mapa de grandària 512x512 unitats.

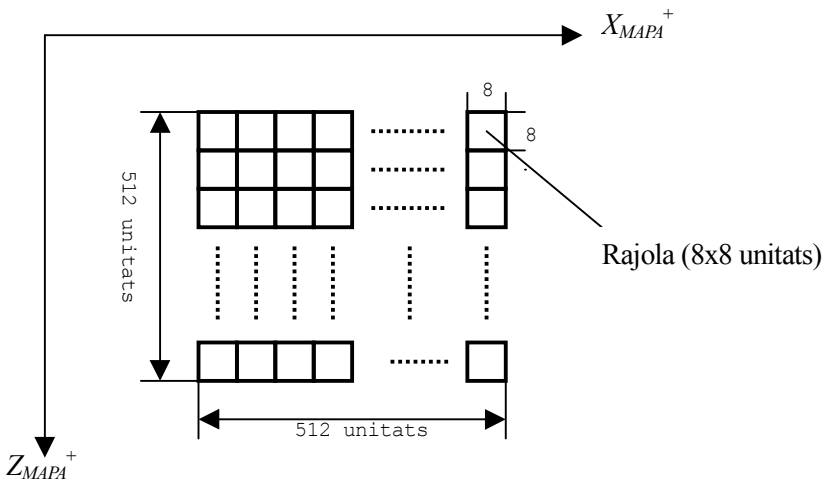
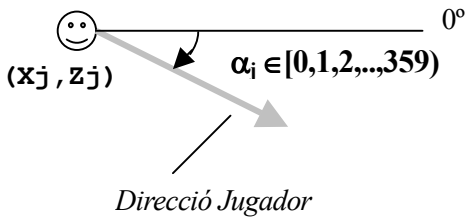


Figura a3.7 *El mapa és representa com una malla 2D. La grandària de cel·la (rajola) és de 8x8 unitats.*

A3.4.2 El jugador

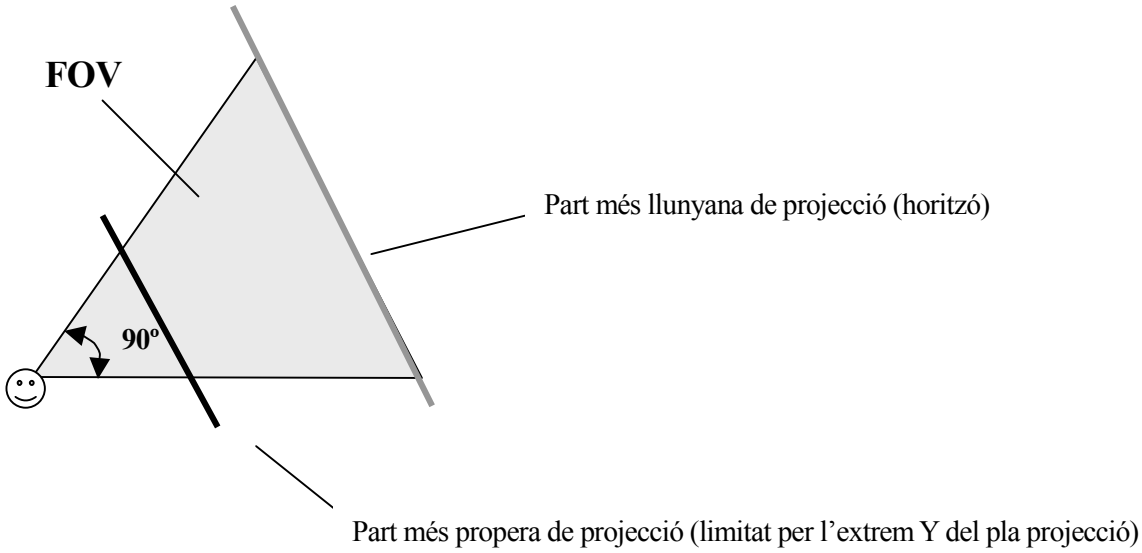
El jugador representa l’element de camera a l’espai 3D, el qual recorrerà en l’espai de mapa explicat a la secció anterior. Definim les coordenades de jugador **Xj** i **Zj** a la posició de mapa actual i α_j com a enter i definit a l’interval 0-359.



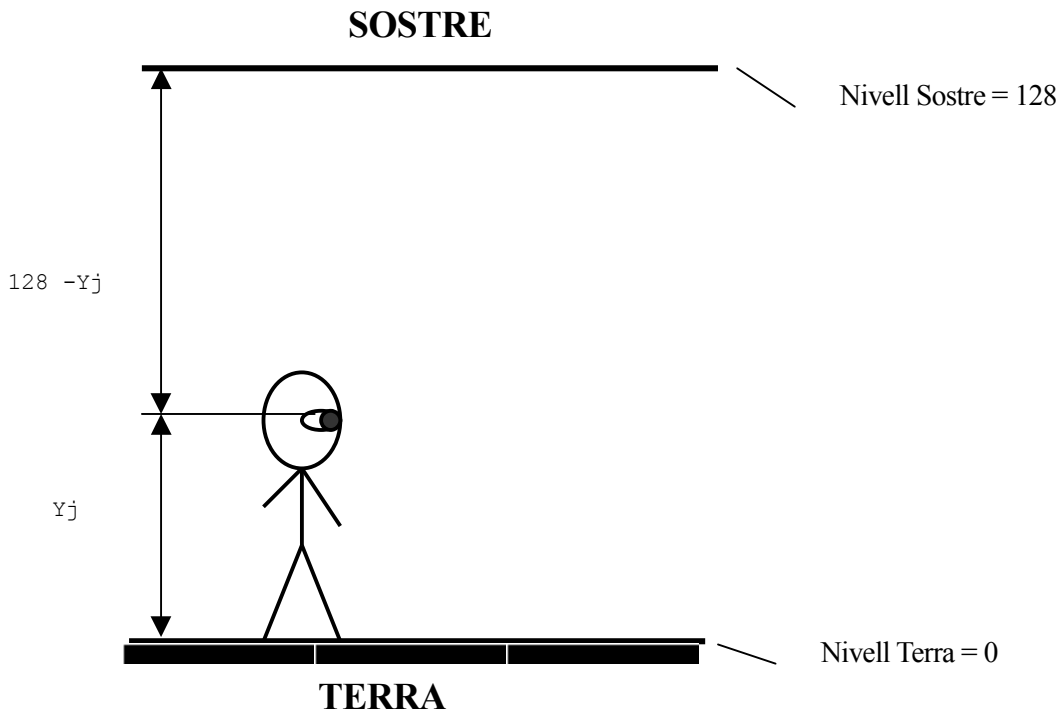
Podrem moure el jugador a l’espai de mapa quan l’usuari premi el control direccional endavant o enrera. La llibreria ham, disposa d’unes macros que entrega l’estat del control direccional.

```
if(F_CTRLINPUT_UP_PRESSED) // S’ha premut tecla amunt → el jugador avança
{
        Xj = Xj + Cosinus(AngleJugador)*Velocitat;
        Zj = Zj +   Sinus(AngleJugador)*Velocitat;
}else{
        if(F_CTRLINPUT_DOWN_PRESSED) // S’ha premut tecla avall → el jugador retrocedeix
        {
                        Xj = Xj - Cosinus(AngleJugador)*Velocitat
                        Zj = Zj -   Sinus(AngleJugador)*Velocitat
        }
}

if(F_CTRLINPUT_LEFT_PRESSED) // S’ha premut tecla esquerra → decrementem angle

        if(--AngleJugador < 0) AngleJugador += 360;

else

    if(F_CTRLINPUT_RIGHT_PRESSED) // S’ha premut tecla dreta → incrementem angle
          if(++AngleJugador > 359) AngleJugador -= 360;
```

Quan el FOV (Camp de visió) de jugador, s’ha convingut de 90º



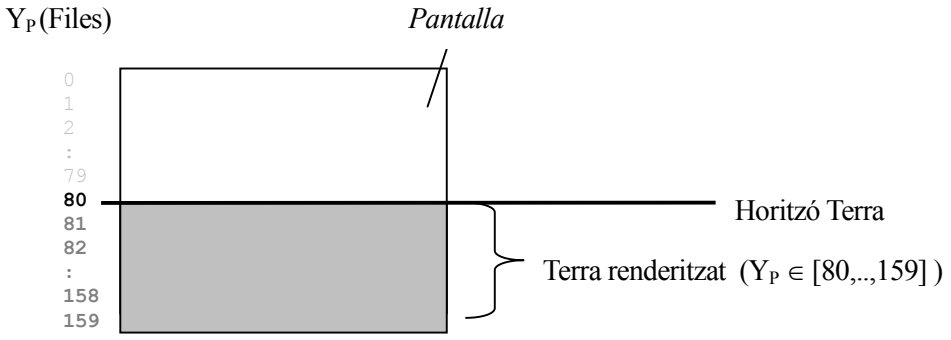
Per últim, l’altura de l’ull de jugador respecte terra que estarà representarà al valor que marqui la variable *Yj*. *Yj* no podrà ser inferior a 0 (Nivell terra) ni superior a 128 (Nivell de sostre) en el món 3D.



A3.4.3 Càlcul de components per la renderització d’un terra

En la present secció es veurà quines són les components necessàries calcular per renderitzar un terra. La següent subsecció explicarà la obtenció de components d’un pla sense rotat de jugador a l’espai 3D, per l’objectiu d’entendre millor l’obtenció de esmentades components. Més tard, permetrem girs a l’espai 3D, però que el seu càlcul no serà res més que una derivada de les anteriors components (sense gir) calculades.

Els scanlines renderitzats d’un terra al pla de projecció, es pintaran des de el scanline que defineix l’horitzó de terra (fila 80); fins a l’extrem superior del pla projecció (fila 159).



Càlcul sense gir a l’espai 3D (α_j = 0º)

Si volem renderitzar un pla sense permetre girar al jugador, fixarem la seva posició al centre amb l’angle de direcció a 0º, o també denominat espai d’ull (veure secció 2.1.1.4, del capítol 2, per més informació sobre l’espai d’ull). Com que tindrem una profunditat Z (Distància) constant, implica la renderització del pla a basat en tires de mapa. Per renderitzar una tira de mapa , només cal fer el càlcul de un valor de escalat en Z i un desplaçament a l’espai de mapa. Partim de la figura A3.8 per estudiar els càlculs que implica la renderització d’una tira de mapa,

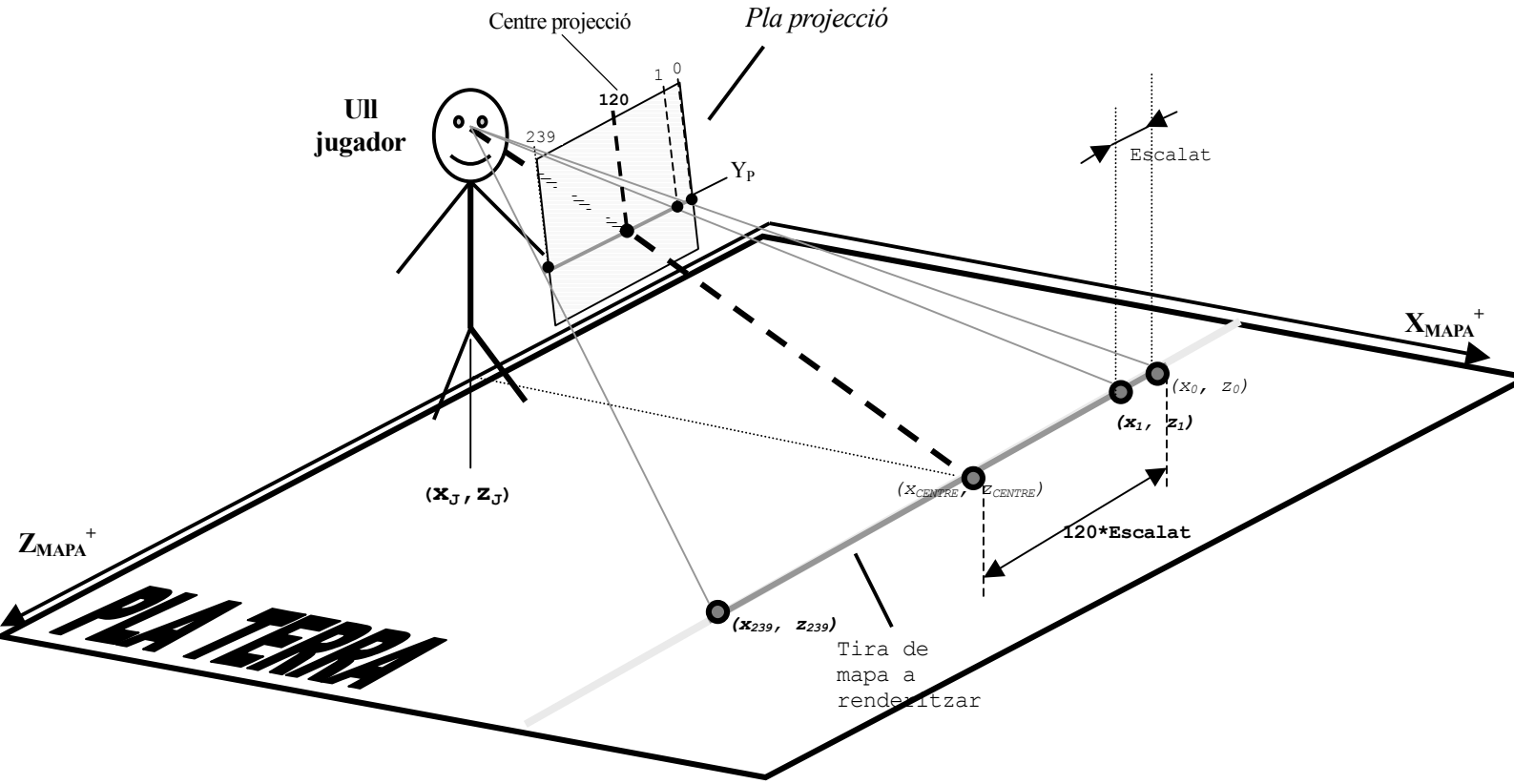


Figura a3.8

El nostre objectiu és trobar la coordenada inicial de mapa (x_0, z_0) projectat al primer pixel del *scanline* (Y_P) i el coeficient de escalat. El procediment per trobar-lo és el següent:

1. Trobem les coordenades de mapa al centre de projecció del scanline (X_{CENTRE}, Z_{CENTRE}) respecte jugador.
2. Calculem el coeficient de variació constant (`Escalat`). L’Escalat representa l’increment a l’espai de mapa equivalent a cada increment unitari en X del pla projecció.
3. Es resta 120 vegades el coeficient de escalat per trobar les coordenades inicials a l’espai de mapa (x_0, z_0).

1. Càlcul de X_{CENTRE} , Z_{CENTRE}

A la figura A3.8 podem observar que Z_{CENTRE} coincideix amb la coordenada de jugador, per tant, respecte el jugador Z_{CENTRE} és Z_j.

$$Z_{CENTRE} = Z_J$$

S’estudiarà la següent figura per trobar X_{CENTRE},

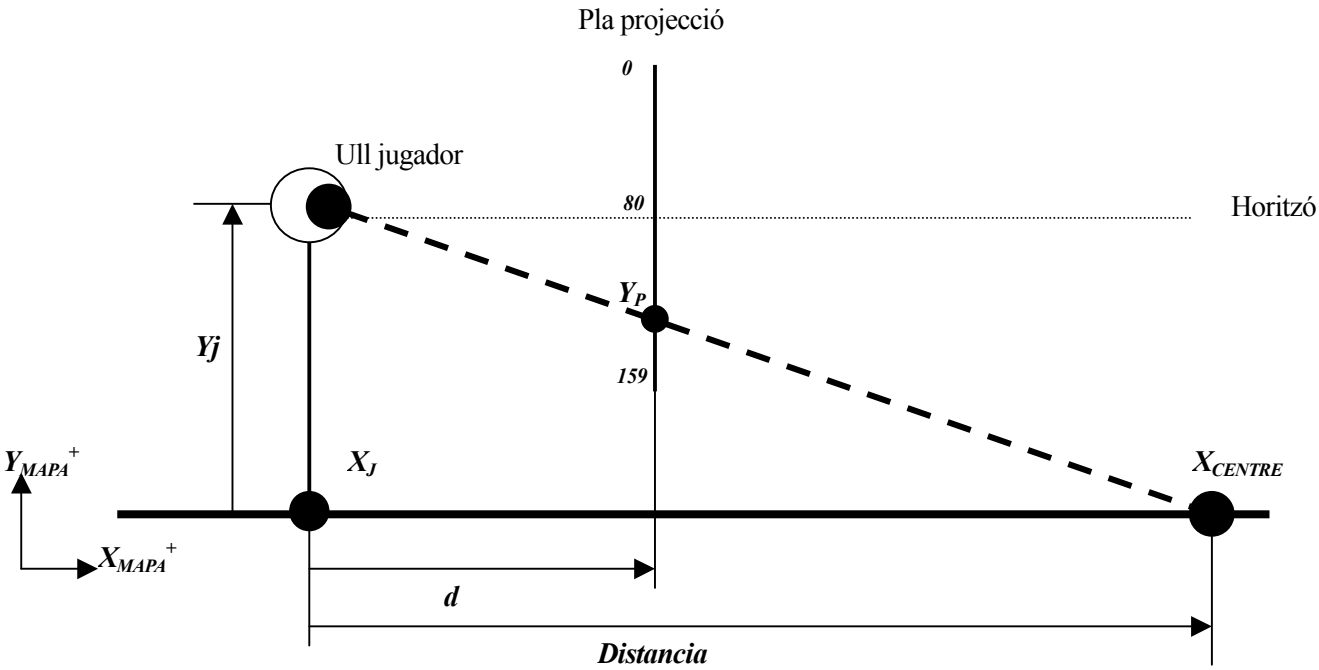


Figura a3.9

Observant la figura A3.9 podem trobar X_{CENTRE}.

$$X_{CENTRE} = X_J + Distancia$$

Per trobar el valor de *Distancia*, amb la figura A3.9 s’empra similitud de triangles,

$$\frac{Y_P}{d} = \frac{Y_J}{Distancia}$$

Aïllant,

$$Distancia = d \cdot \frac{Y_J}{Y_P}$$

Yp està respecte l’horitzó o centre de projecció Y (80), per tant

$$\text{Distancia} = d \cdot \frac{Y_J}{Y_P - 79}$$ **Equació a3.1**

d representa la distància entre jugador i el pla de projecció. Si sabem l’amplada de pantalla (240) i el camp de visió (o en anglès *Fiel Of View* –FOV-) de 90º en que es treballa, trobem *d*.

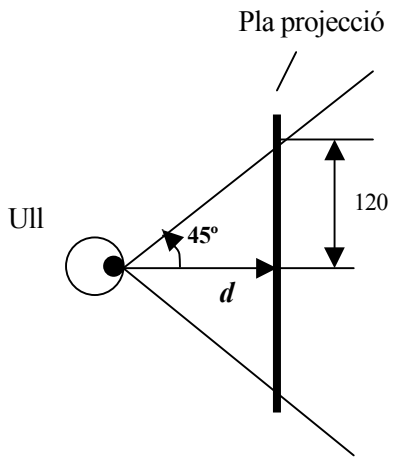


Figura a3.10

Aplicant trigonometria a la figura A3.10, la constant *d* s’aconsegueix com,

$$\tan(45°) = \frac{120}{d} \Rightarrow d = \frac{120}{\tan(45°)} = 120$$

2. Càlcul de l’escalat

L’escalat explica quina variació que es té a l’espai de mapa a cada increment unitari al pla projecció. Podem trobar el valor de escalat estudiant la següent figura.

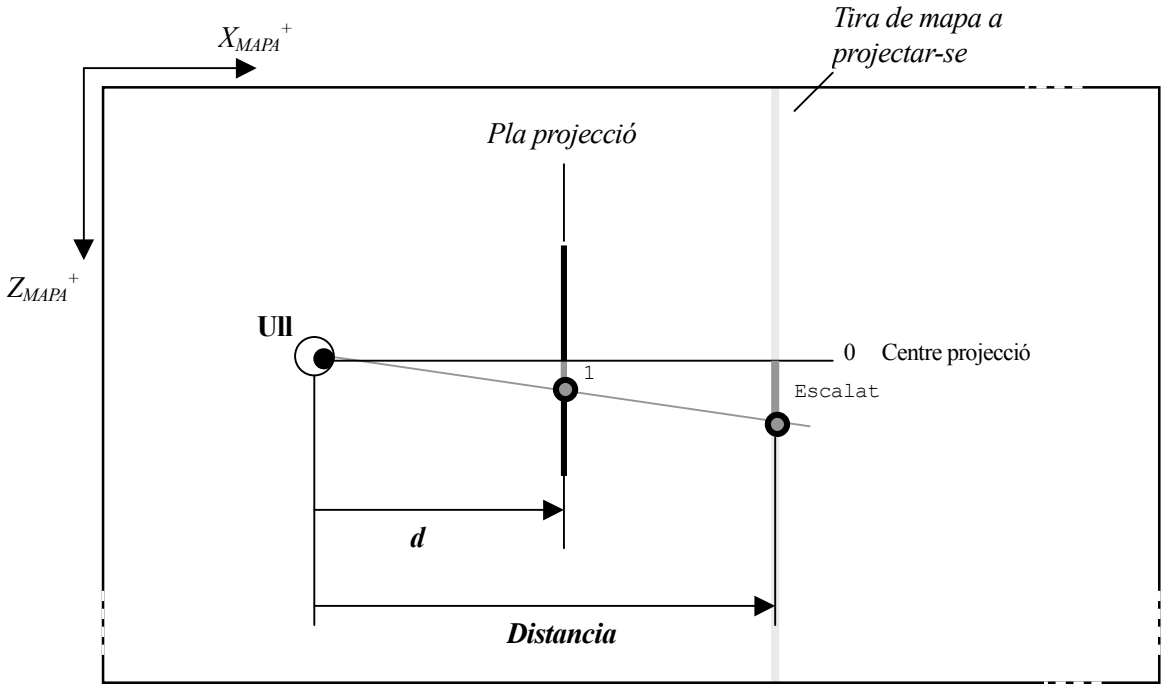


Figura a3.11 *Mapa vista per sobre*

Amb la figura A3.11, s’emprem teoria de similitud de triangles,

$$\frac{1}{d} = \frac{\text{Escalat}}{\text{Distancia}}$$

Aïllem la variable Escalat i el trobem,

$$\text{Escalat} = \frac{\text{Distancia}}{d} \qquad \textit{Equació a3.2}$$

3. Càlcul de la coordenada inicial de mapa (x_0, z_0)

Sabem les coordenades centrals podem trobar l’offset inicial de mapa, restant 120 vegades el Escalat_Z a Z_{CENTRE}. Com que no hi ha variació en X de la tira de mapa a renderitzar, X_0 serà directament X_{CENTRE}.

$$Z_0 = Z_{CENTRE} - 120 \cdot Escalat = Z_J - 120 \cdot Escalat$$

$$X_0 = X_{CENTRE} = X_J + Distancia = X_J + d \cdot \frac{Y_J}{Y_P - 79}$$

Càlcul amb gir α_j a l’espai 3D

Per renderitzar un pla permetent el gir de jugador, però igualment al seu centre, treballarem en forma similar a l’estudiat en la secció anterior. Es veu clar que ara no sempre mapejarem tires de mapa. No obstant, donat que és manté la característica de tenir una profunditat Z constant, mapejarem el mapa amb un dda’s (*Digital Diferential Analizer*) a l’espai de mapa. Per detallar el que s’ha explicat partim de següent figura,

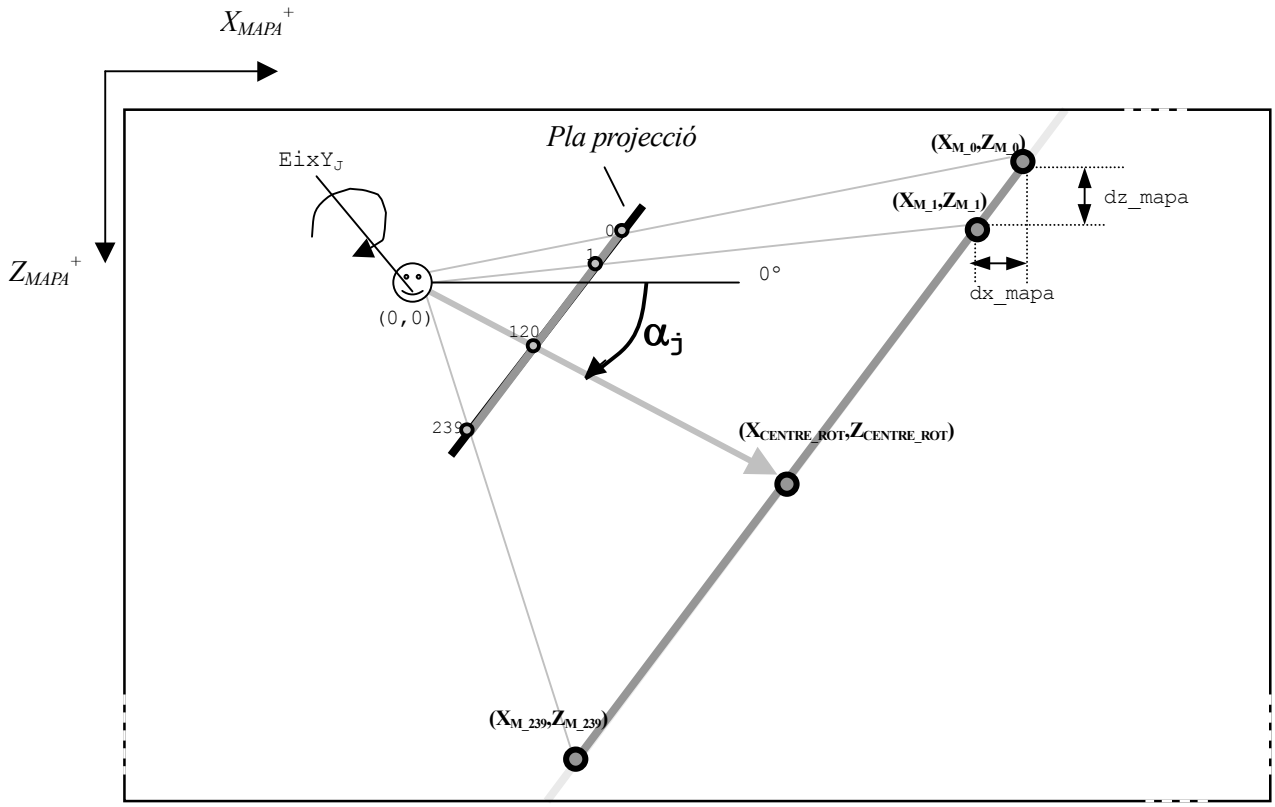


Figura a3.12 *Mapa amb vista per sobre*

Un altre cop, el nostre objectiu està en trobar la coordenada inicial de mapa (x_{M_0}, z_{M_0}) i les deltes constants XZ. El procediment per trobar-lo és el següent:

1. Havent obtingut les coordenades X_{CENTRE}, Z_{CENTRE} a la secció anterior, s’aplicarà una rotació α_j respecte la coordenada Y de jugador i s’obtindran les coordenades X_{CENTRE_ROT}, Z_{CENTRE_ROT}.

2. El següent pas serà trobar els valors de desplaçament delta X i Z (dz_mapa i dx_mapa) constants al espai de mapa.

1. Càlcul de les coordenades (X_{CENTRE_ROT}, Z_{CENTRE_ROT})

Per trobar les coordenades X_{CENTRE_ROT}, Z_{CENTRE_ROT}, cal aplicar una rotació α_j respecte l’eix Y de jugador a les coordenades X_{CENTRE}, Z_{CENTRE} trobades anteriorment.

$$\begin{pmatrix} X_{CENTRE_ROT} \\ 0 \\ Z_{CENTRE_ROT} \end{pmatrix} = \begin{pmatrix} X_{CENTRE} \\ 0 \\ Z_{CENTRE} \end{pmatrix} \cdot Rot(EixY_J, \alpha_J)$$

La matriu de rotació Rot(Y, α_j) és la següent.

$$Rot(EixY_J, \alpha_J) = \begin{pmatrix} \cos(\alpha_J) & 0 & -\sin(\alpha_J) \\ 0 & 1 & 0 \\ \sin(\alpha_J) & 0 & \cos(\alpha_J) \end{pmatrix}$$

Amb la qual cosa es troba els valors X_{CENTRE_ROT}, Z_{CENTRE_ROT}.

$$\begin{pmatrix} X_{CENTRE_ROT} \\ 0 \\ Z_{CENTRE_ROT} \end{pmatrix} = \begin{pmatrix} X_{CENTRE} \\ 0 \\ Z_{CENTRE} \end{pmatrix} \cdot Rot(EixY_J, \alpha_J) = \begin{pmatrix} X_{CENTRE} \\ 0 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} \cos(\alpha_J) & 0 & -\sin(\alpha_J) \\ 0 & 1 & 0 \\ \sin(\alpha_J) & 0 & \cos(\alpha_J) \end{pmatrix} = \begin{pmatrix} \cos(\alpha_J) \cdot X_{CENTRE} \\ 0 \\ \sin(\alpha_J) \cdot X_{CENTRE} \end{pmatrix}$$

$Z_{CENTRE} = 0$

$$X_{CENTRE_ROT} = \cos(\alpha_j) \cdot X_{CENTRE}$$ **Equació A3.3**

$$Z_{CENTRE_ROT} = \sin(\alpha_j) \cdot X_{CENTRE}$$ **Equació A3.4**

2. Càlcul dels increments Delta (`dz_mapa, dx_mapa`)

Les deltes XZ són constants i es desplacen de forma lineal pel mapa. El seu càlcul l’estudiarem amb la següent figura.

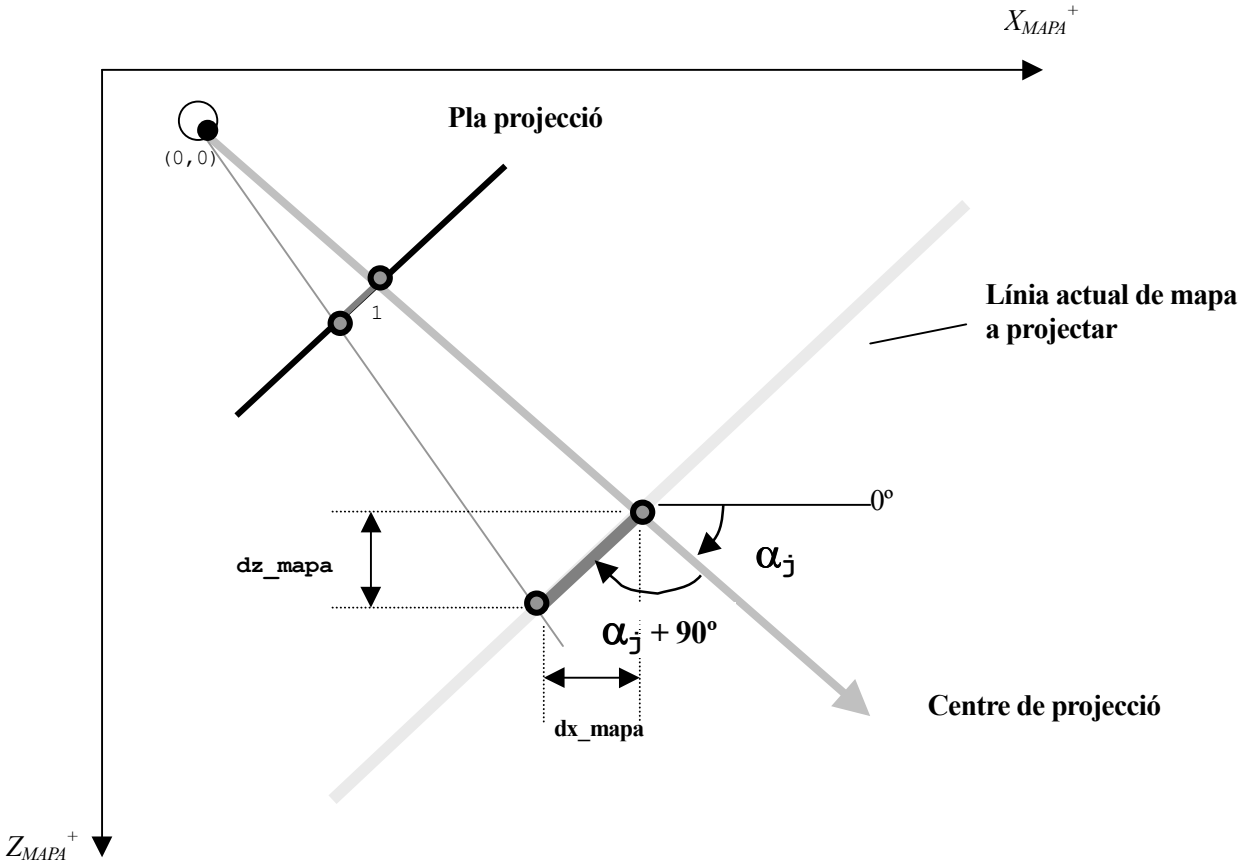


Figura a3.13

Per mapejar una línia de mapa com el de la figura A3.13, les deltes *dx_mapa* i *dz_mapa* són constants i per tant consten d’un pendent amb una velocitat de desplaçament. La velocitat de desplaçament ve marcada per el valor de escalat trobat a la secció anterior. A la figura A3.13, per trobar els pendents de les deltes s’ha de trobar el respectiu angle de direcció.

Es pot comprovar que l’angle direcció dels pendents és perpendicular a l’angle del jugador, per tant trobem els valors dels increments delta com:

$$dx_mapa = cosinus(\alpha_j+90°)\cdot Escalat = \mathbf{-sinus(\alpha_j)\cdot Escalat}$$ **Equació a3.5**

$$dz_mapa = sinus(\alpha_j+90°)\cdot Escalat = \mathbf{cosinus(\alpha_j)\cdot Escalat}$$ **Equació a3.6**

3. Càlcul de la coordenada inicial de mapa (x_0, z_0)

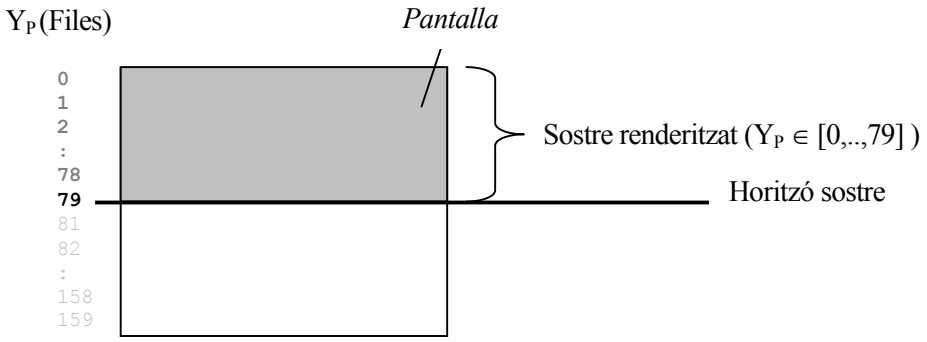
Sabem les coordenades centrals podem trobar l’offset inicial de mapa, restant 120 cops els gradients *dx_mapa* i *dz_mapa* a la coordenades X_{CENTRE_ROT} i Z_{CENTRE_ROT}

$$Z_{_M0} = X_{CENTRE_ROT} - 120 \cdot dx_mapa \qquad \textbf{Equació A3.7}$$

$$X_{_M0} = Z_{CENTRE_ROT} - 120 \cdot dz_mapa \qquad \textbf{Equació A3.8}$$

A3.4.4 Càlcul de components per la renderització d’un sostre

La renderització d’un sostre és semblant a la renderització d’un terra. L’únic a tenir en compte que un sostre renderitzat comprendrà el scanline primer de la pantalla (Fila 0) fins al scanline 79, que defineix el seu horitzó.



Respecte dels càlculs realitzats a la secció anterior per un terra, els càlculs seran els mateixos. L’únic a considerar és el càlcul de distància (equació A3.1), que s’haurà de modificar el denominador perquè Y_p estigui respecte el centre de projecció. Com que el nivell de sostre és 128 caldrà trobar la distància respecte l’altura de jugador en el numerador.

$$Distancia = d \cdot \frac{128 - Y_J}{80 - Y_P}$$

A3.4.5 Algorisme de renderitzat terra i sostre

Havent estudiat la manera d’obtenir càlculs que implica el renderitzat d’un pla, ja sabem l’algorisme necessari per renderitzar el terra i sostre per software.

```
Accio RenderitzaSostreTerra(e Xj,Yj,Zj,Angle:enter)
{ PRE: Xj i Zj ∈ [0,..,1024]; Angle ∈ [0°, 1°, 2°,..,360°] }
var
        X_centre,X_centre_rot, Z_centre_rot, dx_mapa, dz_mapa, Escalat_Z, x0, z0: real

fvar

        per i des de 0 a 159 pas 1 fer

                si i < 80 llavors { Calculem distancia pel terra}

                        Distancia ← d*Yj div (i-79)

                Altrament { Calculem distància pel sostre }

                        Distancia ← d*(128-Yj) div (80-i)
                Fsi


                X_centre_rot ← cos(Angle)*Distancia + Xj
                Z_centre_rot ← sin(Angle)*Distancia + Zj

                Escalat_Z ← Distancia div d
                dx_mapa ← -sin(angle)*Escalat_Z
                dz_mapa ← cos(angle)*Escalat_Z

                x0 ← X_centre_rot  - 120*dx_mapa
                z0 ← Z_centre_rot  - 120*dz_mapa

                RenderitzaScanline(0,239,i,z0,x0,dx_mapa,dz_mapa)
        fper

faccio


accio RenderitzaScanline(e x1,x2, y:enter; x0, z0,dx_mapa,dz_mapa:real)
var
        Npixels, x :enter
        u, v, x_mapa, z_mapa: real
fvar

        Npixels ← x2 - x1 + 1
        x_mapa ← x0
        z_mapa ← z0

        Si Npixels ≥ 1 llavors { Es renderitza scanline }

                Repetir
                        {recordem que la rajola és de 8x8 pixels, per tant s’ha de
                         realitzar mòdul 8 a amplada i alçada per no sortir del sus
                         límits}

                        u ← X_mapa mod 8
                        v ← Z_mapa mod 8

                        PosarPixel(x,y,textura[u][v])

                        x_mapa ← x_mapa + dx_mapa
                        z_mapa ← z_mapa + dz_mapa

                        Npixels ← Npixels - 1

                Mentre Npixels > 0
        Fsi
FAccio
```

Llistat a3.1

A3.5 Adaptació del “mode 7” a la GBA

Havent estudiat la teoria 3D necessària pel renderitzat d’un pla per-software, aquesta secció s’explicarà com **enganyar** la GBA perquè el renderitzi per-hardware, basant-nos amb l’algorisme de la funció *RenderitzaSostreTerra()* anteriorment estudiada. (llistat A3.1)

La renderització tindrà lloc durant el retraçat d’un *scanline*. El retraçat d’un *scanline* consisteix d’un període de 1004 cicles de pintat (*HDraw*) seguit d’un període de 228 cicles que no es fa res (*HBlank*). Durant el període de pintat el hardware gràfic processa les dades de fons i els objectes (sprites) a la pantalla, mentre que **durant el període *HBlank* és deixa lliure perquè codi de programa pugui modificar els atributs de fons i/o d’objectes**.

Es aquí on es tindrà el cor de la renderització, es a dir, **la rutina d’interrupció *Hblank* s’encarregarà de modificar els registres de punt de referència, rotació i/o escalat** (explicats a la secció A3.3) **adequadament per-scanline, aconseguint així, mostrar el pla 3D**.

A3.5.1 Rutina d’interrupció *HBlank*

Implementació de la rutina *HBlank*

Fent referència a l’algorisme de funció *RenderitzarSostreTerra()* del llistat A3.1, explicarem com compatibiltzar-lo a la rutina *HBlank*, així com fer l’assignació de registres dins d’ell.

- **Compatibilització de l’algorisme a la rutina d’interrupció *hblank* registres:**

  Tindrem el mateix flux de codi a la rutina *hblank*, excepte la crida de funció `RenderitzarScanLine()`, la qual es substituirà per assignació de registres hardware. No obstant, s’ha de tenir en compte el següent.

1. Donat que en una rutina d’interrupció no s’hi passen paràmetres en mode usuari, es necessari declarar les variables de jugador (`Xj`, `Yj`, `Zj` i `AngleJugador`) de forma global. Així mateix, no les podrem modificar durant el període *HBlank*, perquè causaria una mala renderització del pla, per tant, caldrà un control on aquest serà a la rutina d’interrupció *VBlank*, la qual s’executa quan el hardware gràfic ha acabat de retraçar tota la pantalla.

2. El comptador de scanlines “i” no tindrà efecte dins l’algorisme, perquè el comptat de scanlines es fa per hardware. La llibreria HAM inclou una macro que entrega el numero de scanline que actualment s’està tractant anomenat *V_CURRENT_SCANLINE* definit a l’interval [0..228] on, d’aquest interval, [0..159] és el període de dibuixat de pantalla (*VDraw*) i [160..228] és el període *VBlank*.

- **Assignació de valors *dx_mapa* i *dz_mapa* a registres:**

A l’estudi de la renderització d’un pla s’ha demostrat que una línia de mapa renderitzada a pantalla representa un DDA a l’espai de mapa. Per tant, només caldrà utilitzar els registres *dx*, *dy* per rotar línies horitzontals de mapa o els registres *dmy*, *dmx* per rotar línies verticals de mapa, segons les deltes *dx_mapa* i *dz_mapa* anteriorment calculades. A cada *scanline*, el hardware gràfic utilitza els registres *dmx*, *dmy* per incrementar el registres de punt de referència per tant **no** els utilitzarem pels nostres propòsits. Per tant, els registres a utilitzar seran: **REG*x*_PA (*dx*) i REG*x*_PC (*dy*)**. Cal dir que no caldrà fer cap adaptació dels valors calculats de *dx_mapa* i *dz_mapa* ja que són exactament els valors estudiats a la secció A3.4.4.2 i que, aquests, assignaran els registres *dx* i *dy*.

La llibreria ham disposa de les definicions `R_BG`***x***`ROTDX` i `R_BG`***x***`ROTDY` les quals fan referència a respectius registres segons el fons ***x*** que s’utilitza.

- **Assignació de valors *x0* i *z0* (offset inici de mapa) a registres:**

La última cosa que s’haurà de fer serà l’assignació del offset inicial de mapa (x0,z0), estudiat a la secció anterior (Càlcul amb gir a l’espai 3D), del pròxim scanline a renderitzar, mitjançant els registres de punt de referència (**REG_xX** i **REG_xY**). La llibreria HAM disposa de les macros `R_BG`***x***`ROTX` i `R_BG`***x***`ROTY` que fan referència a respectius registres segons el fons ***x*** que es vol.

Havent considerat els punts anteriors, la implementació final de la rutina *Hblank* és:

```
Accio hbl_Func()
var
   X_centre,X_centre_rot, Z_centre_rot, dx_mapa, dz_mapa, Escalat_Z, x0, z0: real
fvar

        { Entrega del scanline actual }
        i← V_CURRENT_SCANLINE

        si (i ∈ [0,..,159]) llavors { efectuem els càlculs i assignació de registres}

                si i < 80 llavors                       { Calculem distancia pel terra}
                        Distancia ← d*Yj div (i-79)
                Altrament                               { Calculem distància pel sostre }
                        Distancia ← d*(128-Yj) div (80-i)
                Fsi

                X_centre_rot ← cos(Angle)*Distancia
                Z_centre_rot ← sin(Angle)*Distancia

                Escalat_Z ← Distancia div d
                dx_mapa ← -sin(angle)*Escalat_Z
                dz_mapa ← cos(angle)*Escalat_Z

                x0 ← X_centre_rot  - 120*dx_mapa
                z0 ← Z_centre_rot  - 120*dz_mapa

                { Assignació a registres (suposem que es treballa sobre el fons 2) }

                R_BG2ROTX ← enter(x0*256.0)  { donem el valor x0 a APF 24.8 }
                R_BG2ROTY ← enter(z0*256.0)  { donem el valor z0 a APF 24.8 }

                R_BG2ROTDX ← enter(dx_mapa*256.0) { donem el valor dx_mapa a APF 24.8 }
                R_BG2ROTDY ← enter(dy_mapa*256.0) { donem el valor dz_mapa a APF 24.8 }
        fsi
faccio
```

Per iniciar la rutina de interrupció *hblank* amb el codi descrit, s’ha d’executar la següent funció de la llibreria HAM:

```
ham_StartIntHandler(INT_TYPE_HBL,&hbl_Func)
```

Optimització de la rutina *HBlank*

Donat que es disposa de només 228 cicles ($16{,}48 \cdot 10^{-6}$ segons) per executar el codi pertinent a la rutina *HBlank*, s’ha de optimitzar al màxim l’execució de codi per no sobrepassar els 228 cicles, altrament es podria arribar a tenir el sistema inestable i, possiblement, penjar la màquina.

1a Optimització: Pre-Calcul de operacions trigonomètriques.

Les invocacions a funcions trigonomètriques són molt costoses en temps d’execució. Donat que la variable que representa l’angle de jugador és enter i $\in$ [0,1,2,..,360) podem precalcular els valors trigonomètrics cosinus i sinus en taules globals 360 valors.

2a Optimització: Operacions en Aritmètica de Punt Fixa 16.8 (APF 16.8).

La GBA no disposa de coprocessador d’aritmètica real, per tant totes les operacions en reals són emulades per software, la qual cosa li representa un gran cost a la màquina durant l’execució d’aquestes operacions. Així doncs, la segona optimització que podem donar al codi de l’interrupció és l’ús de la Aritmètica de Punt Fixa (APF) 16.8 (consultar annex 5 per més informació). Definim la variable APF (*FIXED*), la qual substituirà qualsevol variable de tipus real, per el tipus codificat APF en C com,

```
#define    FIXED      int    // int = 32 bits
```

3a Optimització: Eliminació de la divisió.

Durant l’execució tenim una divisió. Les divisions també són operacions costoses a temps d’execució. No obstant, com que el denominador és sempre enter i definit a l’interval [1,..,80] podem precalcular el seus recíprocs en una taula global, per així executar una multiplicació en ves de la divisió. El resultats els tindrem codificats a una APF 16.16 per tenir més precisió a la part fraccional.

```
reciprocal_div : taula [1,...,80] de FIXED { taula de division reciproques (1/y) codificades a
                                             una APF 8.16 (FIXE és un enter de 32 bits}
Accio CalcularDivisioReciproca()
Var
        i:enter
        divisio_reciproca: real
FVar
        Per i desde 1 a 80 fer
                { divisio reciproca }
                divisio_reciproca  =  (1.0/i)

                { divisió reciproca codificada a APF 8.16}
                devisio_reciproca = divisio_reciproca*65536.0

                { emmagatzament de la divisió reciproca }
                reciprocal_div[i] = enter(divisi_reciproca)
        FPer
FAccio
```

4a Aproximacions.

Després de les divisions, les multiplicacions són les també tenen bastant cost d’execució. Al codi tenim algunes divisions i multiplicacions pel valor 120. No obstant, si aproximem el valor a 128 podem executar 7 desplaçaments aritmètics a la dreta per la divisió o a l’esquerra quan es multipliqui.

5a Optimització: Portar el codi HBlank a memòria interna (IWRAM)

Per defecte el compilador HAM, situa el codi de programa a memòria externa o a la ROM el qual són regions de memòria lentes. La cinquena optimització que podem fer, és portar el codi *HBlank* a la regió de memòria interna de treball (IWRAM), aprofitant la més ràpida execució de codi en la màquina. Ho podem fer si es declara la directiva de compilador MEM_FUNC_IN_IWRAM abans de la funció ***hbl_Func()*** . En C es faria el següent,

```
MEM_FUNC_IN_IWRAM void hbl_Func()
{
        ...
}
```

6a Optimització: Eliminació de accessos a memòria

Els resultats d’un sinus i cosinus estaran en taules globals segons un valor d’angle de jugador. Un accés a una taula global, representa un cost de 2 cicles de tipus no continus (2N): un cicle no continu per carrega el valor de punter de la taula SIN o COS i un altre cicle no continu per retorna el resultat segons el valor d’angle. Un cicle no continu és el tipus d’accés més costós que podem trobar durant l’execució d’un programa a la GBA. Donat els accessos a les taules de SIN i COS s’efectuen amb un angle de jugador en tots els càlculs de la rutina HBlank, podem precarregar els valors SIN i COS a l’inici de programa i així evitar fer més accessos a respectives taules.

Donem la implementació final i optimitzada en C corresponent a la rutina d’interrupció *hblank*, per poder apreciar les operacions en APFs i els desplaçaments aritmètics.

```
MEM_FUNC_IN_IWRAM void hbl_Func(void)
{
       FIXED  X_centre,X_centre_rot, Z_centre_rot, dx_mapa, dz_mapa, Escalat_Z, x0, z0,
              TempC, TempS;

       i = V_CURRENT_SCANLINE;  // { Entrega del scanline actual }

       if (0 <= i && i<=159) // { efectuem els càlculs i assignació de registres}
       {
           //{Pre carrega de valors sinus i cosinus per evitar més accessos a les taules }

              TempS = SIN[AngleJugador];
              TempC = COS[AngleJugador];

              if (i < 80) { Calculem distancia pel terra}
              // Abans:
              //
              //   Distancia = Yj*120/(i-79)
              //
              // Ara:
              //
              //   Distancia = Yj*128/(i-79) = Yj*(2^7)/(i-79) =
              //   = Yj*(2^7)*reciprocal_div[i-79]/256 †   =
              //   = Yj*(2^7 + 2^-8)*reciprocal_div[i-79] =
              //   = Yj*reciprocal_div[i-79]*2^-1 → Distancia = Yj*reciprocal_div[i-79] >> 1
             //

                     Distancia = Yj*reciprocal_div[i-79] >> 1;

              else //{ Calculem distància pel sostre }

                     Distancia = (128-Yj)*reciprocal_div[80-i] >> 1;

              X_centre_rot = TempC*Distancia >> 8;
              Z_centre_rot = TempS*Distancia >> 8;

              Escalat_Z = Distancia >> 7; // Distancia/120 ≅ Distancia/128
              dx_mapa = -TempS*Escalat_Z >> 8;
              dz_mapa = TempC*Escalat_Z >> 8;

              x0 = X_centre_rot  - dx_mapa << 8;
              z0 = Z_centre_rot  - dz_mapa << 8;

              // { Assignacio a registres }

              R_BG2ROTX = x0;
              R_BG2ROTY = x0;

              R_BG2ROTDX = dx_mapa;
              R_BG2ROTDY = dz_mapa;
       fsi
faccio
```

† Cal recordar que la taula de divisions recíproques estan codificats a una APF 8.16. Nosaltres treballem a una APF 16.8 en tota la l’execució de la rutina HBlank, per tant cal dividir entre 256 si volem tenir la corresponent APF 16.8.

A3.5.2 Rutina d’interrupció *VBlank*

Com s’ha dit, cal que les actualitzacions de jugador es facin fora del període de retraçat de pantalla, altrament es tindria una renderització del pla 3D totalment anòmala, donat que a cada interrupció *HBlank* tindríem coordenades de jugador diferents als del anterior *scanline* renderitzat.

El control de la actualització de jugador tindrà lloc durant el període vertical que no fa res (*VBlank*). El període Vblank esta disponible quan justament s’ha acabat de pintar tota la pantalla, definit a l’interval de *scanline* [160..228]. Per això, si les coordenades de jugador s’actualitzen al període *Vblank*, podem assegurar que l’usuari no podrà modificar els atributs de jugador durant el pintat de scanlines (interval [0..159]).

Una altre cosa a tenir en compte es que, la interrupció *HBlank* esdevé quan ha acabat el pintat d’un scanline. Des de la funció `hbl_Func()` **mai es compendrà el primer scanline (que fa referència a sostre)**. Per això, en la rutina *Vblank* s’ha de fer la posta a punt dels registres per el primer *scanline*.

```
MEM_FUNC_IN_IWRAM void vblFunc()
{
   FIXED Distancia,x_centre_rot, z_centre_rot, dx_mapa, dz_mapa, x0, z0, TempC, TempS, Escalat_Z;

    // { Actualització de jugador }

    if(F_CTRLINPUT_UP_PRESSED)   {Zj += SIN[AngleJugador] << 3; Xj += COS[AngleJugador] << 3;}
    if(F_CTRLINPUT_DOWN_PRESSED) {Zj -= SIN[AngleJugador] << 3; Xj -= COS[AngleJugador] << 3;}
    if(F_CTRLINPUT_LEFT_PRESSED) if(--Angle < 0)   Angle += 360;
    if(F_CTRLINPUT_RIGHT_PRESSED if(++Angle > 359) Angle -= 360;
    if(F_CTRLINPUT_L_PRESSED if((Yj+1) < 128)  Yj ++;
    if(F_CTRLINPUT_R_PRESSED if((Yj-1) > 0)    Yj --;

    // { Càlculs i assignació de registres corresponents al primer scanline (sostre)}

    TempC = COS[Angle];
    TempS = SIN[Angle];

    Distancia = (128-Yj)*reciprocal_div[80] >> 1;

    x_centre_rot = (TempC*Distancia >> 8) + Xj;
    z_centre_rot = (TempS*Distancia >> 8) + Zj;

    Escalat_Z = Distancia >> 7;

    dx_mapa = -TempS*Escalat_Z >> 8;
    dz_mapa =  TempC*Escalat_Z >> 8;

    x0 = x_centre_rot - (dx_mapa << 7);
    z0 = z_centre_rot - (dz_mapa << 7);

    R_BG2ROTDX = dx_mapa;
    R_BG2ROTDY = dz_mapa;

    R_BG2ROTX = x0;
    R_BG2ROTY = z0;
}
```

Per iniciar la rutina de interrupció *VBlank* amb el codi descrit, s’ha d’executar la següent funció de la llibreria HAM:

```
ham_StartIntHandler(INT_TYPE_VBL,&vbl_Func)
```

A3.6 Preparació de una demostració

Aquesta secció pretén preparar una demostració d’un recorregut interactiu en “mode 7”. El primer pas ha estat la construcció de les següents imatges.

Figura a3.14 *Imatge “nuvols.bmp” de 128x128 pixels i 256 colors*

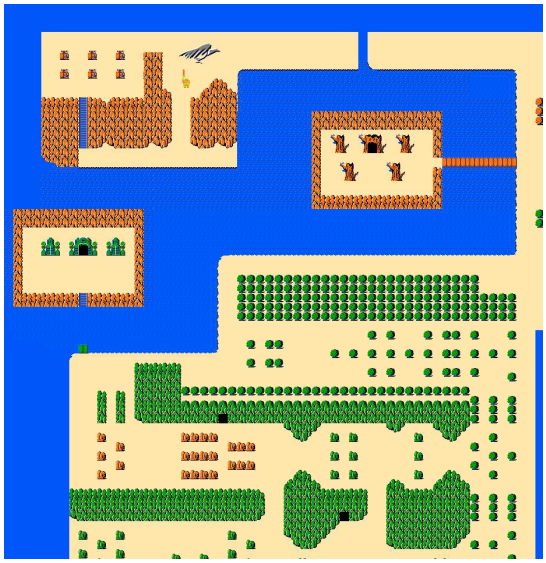


Figura a3.15 *Imatge “terra.bmp” de 1024x1024 pixels i 256 colors.*

De les quals les imatges A3.14 i A3.15 mapejaran el sostre i terra als fons 2 i 3 respectivament, per tant, caldrà inicialitzar un mode gràfic que permeti 2 fons amb transformació (estem parlant de l’inicialització del mode 2).

Les imatges A3.14 i A3.15 han de ser prèviament convertides a un format de tipus rajola per un fons transformable, amb el vector mapa corresponent i llegible a la GBA. Tot això és possible amb l’execució de l’eina GFX2GBA (annex a2.1.3) i amb els següents paràmetres al directori on tenim les imatges “nuvols.bmp” i “terra.bmp”.

```
gfx2gba -fsrc -m -rs -t8 *.bmp
```

L’eina crearia els següents fitxers:

1. ***master.pal.c***: conté un vector 1D en C de la paleta global representativa de les dues imatges.

2. ***nuvols.raw.c***: conté un vector 1D en C de la informació gràfica de les rajoles disponibles i en format cru, corresponent a la imatge “nuvols.bmp”.

3. ***nuvols.map.c***: conté un vector 1D en C de la informació de mapa corresponent a la imatge “nuvols.bmp”.

4. ***terra.raw.c***: conté un vector 1D en C de la informació gràfica de les rajoles disponibles i en format cru, corresponent a la imatge “terra.bmp”.

5. ***terra.map.c***: conté un vector 1D en C de la informació de mapa corresponent a la imatge “terra.bmp”.

Els 5 fitxers els inclourem al fitxer de codi font principal.

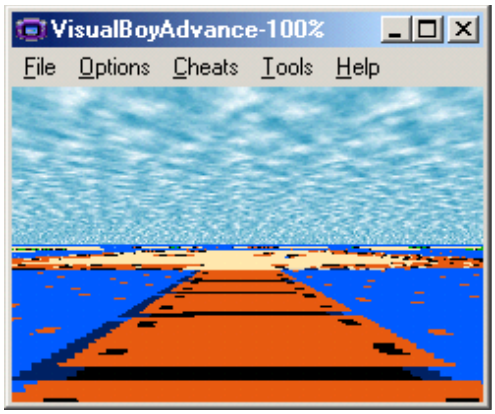
I finalment, donem el codi de la demostració (per més informació sobre funcions de inicialització ham per un fons de tipus rajola, llegir la secció a3.6 i a3.7 de l’annex 3) editat a l’arxiu “main.c”.

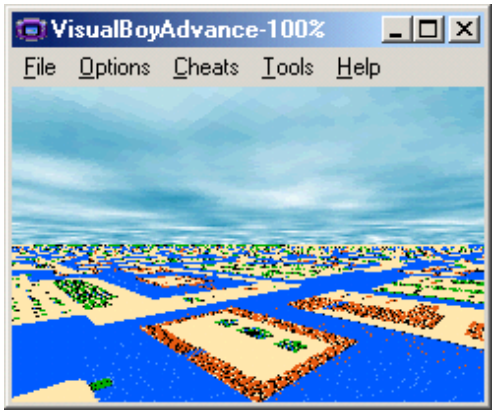
main.c

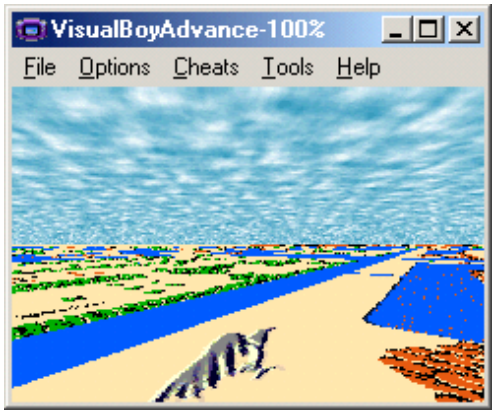
```
#include "mygba.h"

#include "gfx/master.pal.c"  // Paleta global...
#include "gfx/nuvols.raw.c"  // Format cru de les rajoles de la imatge “nuvols.bmp”...
#include "gfx/nuvols.map.c"  // Vector mapa dels “nuvols.bmp”...
#include "gfx/terra.raw.c"   // Format cru de les rajoles de la imatge “terra.bmp”...
#include "gfx/terra.map.c"   // Vector mapa dels “terra.bmp”...
#include "LUTs.c"            // Tenim guardades les taules precalculades SIN,COS i reciprocal_div.

// Variables globals de jugador...
FIXED Xj=0;
FIXED Zj=0;
FIXED Yj = 32;
int AngleJugador = 0;


MEM_FUNC_IN_IWRAM void hblFunc()
{
    FIXED Distancia, x_centre_rot, z_centre_rot, dx_mapa, dz_mapa, x0, z0, TempC, TempS, Escalat_Z;

       int   i = F_VCNT_CURRENT_SCANLINE;

        if((0 <= i && i <= 159))
        {

        TempC = COS[AngleJugador];
        TempS = SIN[AngleJugador];

        if(i >= 80)  Distancia = Yj*reciprocal_div[i-79] >> 1;
        else Distancia = (128-Yj)*reciprocal_div[80-i] >> 1;


         x_centre_rot = (TempC*Distancia >> 8) + (Xj);
         z_centre_rot = (TempS*Distancia >> 8) + (Zj);

         Escalat_Z = Distancia >> 7;

         dx_mapa = -TempS*Escalat_Z >> 8;
         dz_mapa =  TempC*Escalat_Z >> 8;

         x0 = x_centre_rot - (dx_mapa << 7);
         z0 = z_centre_rot - (dz_mapa << 7);

         if(i >= 80){ R_BG3ROTDX = dx_mapa;R_BG3ROTDY = dz_mapa; R_BG3ROTX = x0;R_BG3ROTY = z0;}
         else{ R_BG2ROTDX = dx_mapa;R_BG2ROTDY = dz_mapa;R_BG2ROTX = x0; R_BG2ROTY = z0;}
}


MEM_FUNC_IN_IWRAM void vblFunc()
{
    FIXED Distancia, x_centre_rot, z_centre_rot, dx_mapa, dz_mapa, x0, z0, TempC, TempS, Escalat_Z;

    if(F_CTRLINPUT_UP_PRESSED) {Zj+=SIN[AngleJugador];Xj+=COS[AngleJugador];}
    if(F_CTRLINPUT_DOWN_PRESSED) {Zj-=SIN[AngleJugador];Xj-=COS[AngleJugador];}
    if(F_CTRLINPUT_LEFT_PRESSED)    if(--AngleJugador < 0) AngleJugador += 360;
    if(F_CTRLINPUT_RIGHT_PRESSED)   if(++AngleJugador > 359) AngleJugador -= 360;
    if(F_CTRLINPUT_L_PRESSED && (Yj+1) < 127)   (Yj++);
    if(F_CTRLINPUT_R_PRESSED && (Yj-1) > 0)             (Yj--);

    TempC = COS[AngleJugador];
    TempS = SIN[AngleJugador];
    Distancia = (128-Yj))*reciprocal_div[80] >> 1;

    x_centre_rot = (TempC*Distancia >> 8) + (Xj);
    z_centre_rot = (TempS*Distancia >> 8) + (Zj);

    Escalat_Z = Distancia >> 7;

    dx_mapa = -TempS*Escalat_Z >> 8;
    dz_mapa =  TempC*Escalat_Z >> 8;

    x0 = x_centre_rot - (dx_mapa << 7);
    z0 = z_centre_rot - (dz_mapa << 7);

    R_BG2ROTDX = dx_mapa;R_BG2ROTDY = dz_mapa; R_BG2ROTX = x0;  R_BG2ROTY = z0;

}
```

```
int main(void)
{
        map_fragment_info_ptr mymap2,mymap3;

        // initialize HAMlib
        ham_Init();


        // Inicialitzacio mode rajola amb dos fons rotables i/o escalables
        ham_SetBgMode(2);

        // Configura el fons 2 i 3, perquè es mapegin a pantalla de forma infinita...
        M_BG2CNT_WRAP_SET_ON;
        M_BG3CNT_WRAP_SET_ON;

        // inicialització de la paleta global dels dos fons...
        ham_LoadBGPal(&master_Palette,sizeof(master_Palette));

        // càrrega de les rajoles dels dos fons respectius...
        ham_bg[2].ti = ham_InitTileSet(&nuvols_Tiles,SIZEOF_16BIT(nuvols_Tiles),1,1);
        ham_bg[3].ti = ham_InitTileSet(&terra_Tiles,SIZEOF_16BIT(terra_Tiles),1,1);

        // inicialització de dos mapes buits...
        ham_bg[2].mi = ham_InitMapEmptySet(0,1);
        ham_bg[3].mi = ham_InitMapEmptySet(3,1);

        // Es fa una referència a les dades dels dos mapes en la ROM
        mymap2 = ham_InitMapFragment(&nuvols_Map,16,16,0,0,16,16,1);
        mymap3 = ham_InitMapFragment(&terra_Map,128,128,0,0,128,128,1);

        // Còpia (en aquest cas sencera) els mapes al fons 2 i fons 3 i a les coordenades x=0 y=0
        ham_InsertMapFragment(mymap2,2,0,0);
        ham_InsertMapFragment(mymap3,3,0,0);

        // l'habilitem els dos fons perquè es vegin a pantalla
        ham_InitBg(2,1,1,0);
        ham_InitBg(3,1,1,0);

        // Aquesta funció crea una finestra de 80x240, la qual s’indica que volem veure al seu
        // interior el fons 3 (terra) i al seu exterior el fons 2 (sostre).

        ham_CreateWin(0,0,80,240,160,WIN_BG3,WIN_BG2,0);

        // inicialització de la rutina d'interrupció VBLANK i HBLANK
        ham_StartIntHandler(INT_TYPE_VBL,&vblFunc);
        ham_StartIntHandler(INT_TYPE_HBL,&hblFunc);

    while(1)
    {

    }
}
```

A3.7 Captures de pantalla de la demostració







A3.8 Conclusions

Aquest annex ha donat l’estudi i la corresponent implementació d’una simulació de “mode 7” per la GBA i del qual s’ha podem destacar 2 observacions:

1. Per renderitzar un pla 3D per hardware és necessari utilitzar algun dels modes gràfics basats fons transformables (modes 1 o 2).

2. Durant la renderització de cada *scanline* es fa sempre entre les cotes x1=0 a x2=239 (tota la pantalla).

En futurs projectes, es podria aprofitar aquest estudi per un motor de jocs amb perspectiva “semi-3D” (Només plans 3D), com passava amb alguns dels jocs a la SNES (comentats a la secció A3.1).

A3.9 Treballs futurs

- ## Plans curvats

En futurs treballs, portar l’efecte de plans curvats com l’observat a la figura A3.16. [TH01] explica la possibilitat de generar aquest efecte.

Figura a3.16 *Captura del joc Terra Enigma on es pot observar que el terra i sostre té un efecte curvat i amb perspectiva.*

- ## Direcció lliure a l’espai YZ

Tenir una direcció 2D lliure en YZ per les renderitzacions “mode 7”, permetrà al jugador pujar abaixar el cap i moure lliurament per esmentat espai. En [TONC04B], s’explica en tota detall la manera d’aconseguir-ho.

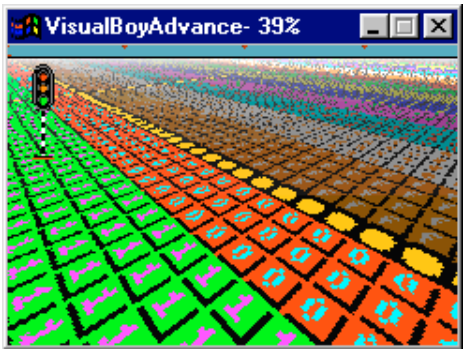


Figura a3.17

Annex 4.1 Aritmètica entera

A4.1.1 Aritmètica de Punt Fixa (APF)

L'Aritmètica de Punt Fixa (APF) és una tècnica molt utilitzada en aplicacions a temps real sota un sistema que no disposi de coprocessador per reals. Qualsevol sistema que no tingui coprocessador per operacions aritmètiques reals, requereix que siguin emulades per software, la qual cosa, i si el sistema és lent, li representa un pes important a la execució de les aplicacions a temps real.

Però, encara que el sistema disposi de coprocessador, quan en parts de programa hi calen conversions de enter a real o viceversa, això pot fer més lent el procés, no obstant, les conversions de enter a APF no li representa cap pes ja que, com veurem més endavant, només cal un desplaçament aritmètic per convertir-lo de enter-APF o viceversa.

A4.1.1 Definició de la APF

Una variable de tipus APF està definida bàsicament per un enter on s'estableix en els ***x*** bits de més pes, menys **1 bit** que es reserva pel signe. La resta de bits ***y*** estableixen la precisió fraccional del numero.

Bit signe	**part_entera**	**part_fraccional**

x = | ***part_entera*** |
y = | part_fraccional |
Total bits = x + y + ***bit signe***

Així doncs, el rang enter estarà definit a l'interval de permeti l'enter.

$$-2^{x-1} \leq \text{rang_enter} \leq 2^{x-1} - 1.$$

Pel que fa a la resolució fraccional (el mínim numero que podrem representar) estarà marcat pel numero de bits establerts a la part fraccional i serà de:

$$\textit{resolució_fraccional} = \frac{1}{2^y}$$

Doncs, ja podem declarar les primeres definicions APF en C:

```
#define  FIXED                 int          // definició de APF és el tamany de
                                             // dades que suporta el processador, es
                                              // a dir, un enter.

#define  BITS_PART_ENTERA      x            // Numero de bits per la part entera.
#define  BITS_PART_FRACCIONAL  y            // Numero de bits per la part fraccional
```

A4.1.2 Conversions APF

- ENTERS

Per fer conversions de enter a APF, s'ha de multiplicar per 2^y, o millor, desplaçar aritmèticament ***y*** cops a l'esquerra,

```
#define INT_TO_FIXED(x) ((x) << BITS_PART_FRACCIONAL)
```

Per fer conversions de APF a enter, s'ha de dividir per 2^y, o millor, desplaçar aritmèticament ***y*** cops a la dreta,

```
#define FIXED_TO_INT(x) ((x) >> BITS_PART_FRACCIONAL)
```

- REALS

Per fer conversions de reals a APF, no podem realitzar un ràpid desplaçament aritmètic donat que els reals estan en format IEEE. Per tant, forçosament s'ha de multiplicar per 2^y,

```
#define DOUBLE_TO_FIXED(x) ((x)*(1 << PART_FRACCIONAL))
```

Per fer conversions de APF a real, només podem dividir per 2^y,

```
#define FIXED_TO_DOUBLE(x) ((x)/(1 << BITS_PART_FRACCIONAL))
```

A4.1.3 Operacions aritmètiques en APF

Les operacions aritmètiques suma i resta funcionen exactament igual que en operacions reals. D'altra banda, per les operacions multiplicació i divisió, cal un petit estudi.

• Operació aritmètica multiplicació

Quan s'efectua una multiplicació en APF, s'ha de fer una correcció després de la multiplicació. Si volem fer la multiplicació 100*100 en base 10, en APF es realitza la següent multiplicació $(100*2^y)*(100*2^y)$. Un numero enter en APF és el mateix multiplicar per 2^y. Així doncs, l'únic que fer és dividir per 2^y per tornar el numero APF a la precisió correcte. La multiplicació APF s'executaria de la següent manera:

```
#define MulFx(num1,num2) (((num1)*(num2)) >> BIT_PART_FRACCIONAL)
```

• Operació aritmètica divisió

De manera similar, quan s'efectua una divisió en APF, s'ha de fer una correcció en el numerador abans de executar la divisió. Si volem fer la divisió 100/100 en base 10, en APF es realitza la següent divisió $(100*2^y)$ / $(100*2^y)$, cosa que provoca el resultat 1, resultat incorrecte. El problema es que es simplifica els factors 2^y. La solució és premultiplicar el numerador per 2^y abans d'executar la divisió.

```
#define DivFx(num1,num2) (((numerador)<< BITS_PART_FRACCIONAL)/(denominador)
```

A4.1.4 Enter de doble capacitat

Tal com s'ha explicat a la secció A4.1.3, a l'hora de aplicar la operació divisió o multiplicació APF, calen ***y*** bits més degut a la amplificació de les operacions, per el que el rang enter quedaria en un rang efectiu de 2^{x-y}. No obstant, molts compiladors poden combinar varis registres per simular tipus de variables de més capacitat que la paraula del processador. Una exemple és el tipus *long long int*, que simula un registre de 64 bits, en cas que el processador sigui de 32 bits. Per forçar la operació divisió/multiplicació en aquest format enter de 64 bits es visualitza al les següents macros C,

```
#define long long int s64

#define Long_Mul(x,y)   (s64)(((s64)x)*((s64)y))
#define Long_Div(x,y)   (s64)(((s64)x)/((s64)y))
```

A4.1.5 Consells a l'hora d'utilitzar una APF

Si en alguna formula de codi està formada per una combinació de multiplicacions i divisions en APF, en recomanable ordenar l'ordre d'execució per tal d'evitar un possible overflow pel rang enter i/o un valor no representable pel rang fraccional. Per tant, caldrà segui un ordre tal que no s'alteri el valor resultant. O sigui, si primer es multiplica, llavors es divideix, llavors és torna a multiplicar, etc.

A4.2 Algorisme de la divisió entera amb signe

En el cas que el sistema no disposi de divisió, el següent llistat mostra l'algorisme de la divisió entera amb signe per què pugui ser codificada.

```
            funcio divisio(entrada Numerador,Denominador:enter) retorna enter

   var
     Shifts: enter
     i: enter
     Quocient: enter
     signe: boolea
   fvar

   signe ← (Numerador SHR 31) XOR (Denominador SHR 31)

   Numerador = abs(Numerador)
   Denominador = abs(Denominador)

   Si (Denominador >  Numerador)  llavors

     Quocient ← 0

   Altrament

        Si Numerador = 0 llavors

            Quocient ← 0

        Altrament

             Si Denominador = 0 llavors {Error: Divisió per zero!}

              Altramanent

                        Shifts ← lmbd(Denominador) - lmbd(Numerador)
                        Denominador ← Denominador SHL Shifts

                        Per i desde 0 fins a Shifts fer

                           subc(Numerador,Denominador)
                        Fper

                        Si Signe llavors

                          Quocient ← - Numerador

                        Altrament

                          Quocient ← Numerador

                        FSi
               FSi
       Fsi
   Fsi

   divisio ← Quocient
Ffunció
```

Llistat A4.1

• La funció abs()

La funció *abs()* retorna el valor absolut del numero enter entrat per paràmetre.

• La funció lmbd()

La funció *lmbd()* retorna el numero de desplaçaments a l'esquerra que cal, per trobar el primer `1` al bit més significatiu (MSB).

Exemple 1

Si tenim el següent numero inicialitzat

Numero = 39

I, si el tamany de paraula és de 8 bits la representació en base 2 de 39 serà de:

$$39 = 2^5 + 2^2 + 2^1 + 2^0 =$$

MSB							LSB
0	0	1	0	0	1	1	1

Si executem la funció *lmbd(Numero)* el resultat, com be hem dit, serà de 2.

```
Funció Lmbd(entrada numero) retorna enter

{ PRE: El tamany de paraula és de 32 bits.}

          Var
     Resultat:enter
     BitActual:enter
          FVar

  Resultat ← 0
  BitActual ← 0x80000000

  Mentre (Numero AND BitActual ≠ 1)  i (Resultat < 32) fer
       Resultat ← Resultat + 1
       BitActual ← BitActual*2
  FMentre

Lmbd ← Resultat

{POST: 0 ≤ Resultat ≤ 32 }

FFuncio
```

Llistat A4.2

La funció subc()

A la funció subc() s'hi entra els valors enters denominador i numerador. La operació que s'hi dins d'aquesta és una resta condicional com mostra al llistat A4.3.

```
Funcio subc(entrada Numerador,Denominador:enter) retorna enter
Var
       Resultat:enter
FVar

     Si Numerador ≥Denominador llavors

        Resultat ← (Numerador - Denominador) SHL 1 + 1

    Altrament

        Resultat ← Numerador SHL 1
    FSi

     Subc ← Resultat

FFuncio
```

Llistat A4.3

• Operacions aritmètiques SHR, SHL, XOR i AND

El les operacions aritmètiques i significat que hi trobem als l'algorismes anteriors són:

SHR: **Desplaçament a la dreta. (Equival a dividir per 2)**
SHL: **Deslaçament a l'esquerra. (Equival a multiplicar per 2)**
XOR: **OR exclusiva lògica.**
AND: **AND lògica.**

ANNEX 5: Filtre MIP-MAPPING

Els programadors de gràfics estudien constantment altres camins per perfeccionar els realisme de gràfics en els jocs. Una de les tècniques simples més empleades que ho fa possible és l'anomenada tècnica "texture mapping". La tècnica "texture mapping" afegeix realisme de gran consideració, a l'escena, però també afegeix un numero de nous problemes. Un dels problemes més freqüents és l'aparició de aliasing durant el mapeig de textures en polígons, a una distància de l'observador.

Aquestes parts del polígon amb aliasing causen efectes luminiscents o animats sobre la superfície de la textura, si l'observador es mou ràpidament pel món virtual. En canvi, si l'observador està fix, aquestes parts amb aliasing apareixen com mostres no trobades en la textura després d'haver-se mapejat en el polígon. El que s'ha explicat, es clarament visible a la figura A5.1, on podem observar que una textura d'escacs mapejada més distorsionada a mesura que es va allunyant de l'observador.

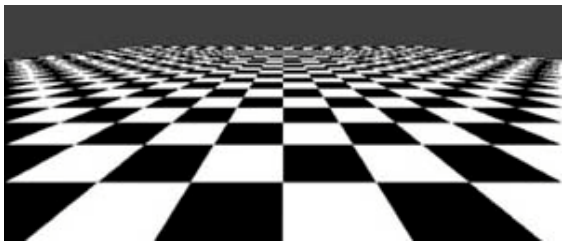


Figura A5.1 *Tauler d'escacs sense MIP-MAPPING*

La tècnica MIP-MAPPING ajudar a alleugerar aquest problema. L'acrònim MIP prové de llatí *Multum in Parvo*, que significa: "moltes coses en un lloc petit". Els recercadors, de l'institut de tecnologia de Nova York van adoptar aquest terme en 1979 per descriure una tècnica pel mig del qual molts pixels poguessin ser filtrats i comprimits en un lloc reduït, conegut com el MIP-map. Per veure com els MIP-maps perfeccionen la claredat Visual, veure figura A5.2, en el qual s'ha utilitzat un MIP-mapping amb filtrat bilinear per suavitzar la textura.

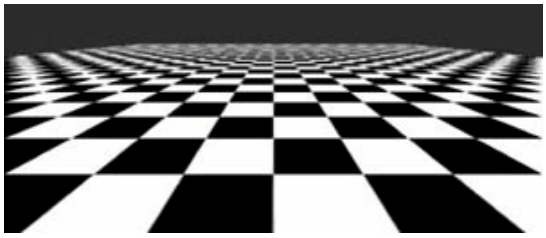


Figura A5.2 *Tauler d'escacs amb MIP-MAPPING i filtre bilinear*

Per entendre millor quin és el factor que causa els problemes de la figura A5.1, s'ha de observar com treballa el render del "texture mapping" i entendre com el procés de mostreig del mapeig de textura afecta al que es veu a pantalla. Si observem la figura A5.3A, una ona sinus s'ha mostrejat a una freqüència més alta que la seva freqüència. Com podem veure, hi ha una representació molt bona de la ona la qual pot ser reconstruïda. No obstant, si la freqüència de mostreig deixa d'estar per sota del doble de la freqüència de la ona sinus, llavors és possible que els punts mostrejats coincideixin amb els punts de transició de la ona (valor 0), tal com podem apreciar a la figura A5.3B, i per tant la impossibilitat de recuperar la seva informació. Les freqüències menors que el doble de la ona sinus mostrejada, vist a la figura A5.3A, causa que la informació de les mostres apareixin com una ona sinus de més baixa freqüència que l'original (veure figura A5.3C). D'aquestes figures, podem deduir

que per tenir la informació completa del senyal a mostrejar, la freqüència de mostreig ha de ser com a mínim el doble d'esmentat senyal.

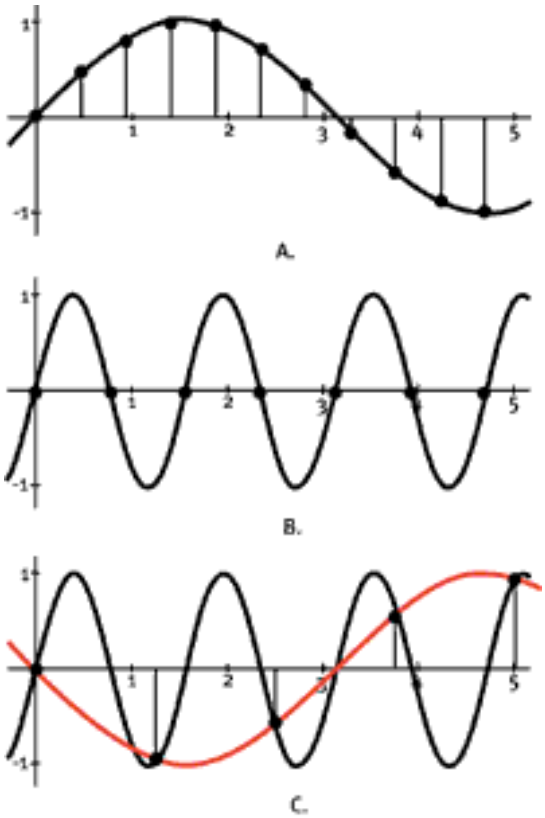


Figura A5.3 *Mostreig d'una ona sinus amb variant l'interval de mostreig*

Aquest fet és conegut com el límit de Nyquist. Però, com es demostra que la freqüència de mostreig ha de ser el doble de la freqüència del senyal a mostrejar?

Per donar resposta a aquesta pregunta, hem de desviar-nos del tema Nyquist i anar al domini de Fourier.

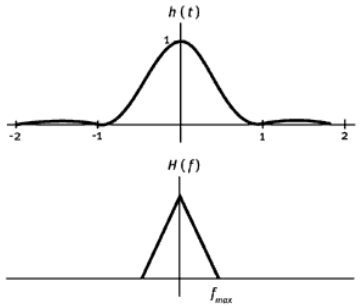


Figura A5.4 *Anàlisis del mostreig de Fourier*

A la figura A5.4, hi tenim el dibuix de la funció h(t) = sinc^2x i el dibuix de la transformada de Fourier, H(t), on H(t) representa el domini de freqüència i h(t) el domini de temps. Per convertir el domini de temps al domini de freqüència, s'ha d'aplicar la següent fórmula de transformació en h(t).

$$H(t) = \int_{-\infty}^{\infty} h(t) \cdot e^{-i2\Pi ft} \cdot dt$$

Eq. 1

Aquesta forma de la transformació de Fourier, *f* defineix les freqüències de la ona sinus que caracteritza el senyal i $\left(i = \sqrt{-1}\right)$ explica que l'exponent és complexe. L'operador É és sovint utilitzat per denotar "té la transformada de Fourier", per això podem escriure h(t)ÉH(t). En la figura A5.5B, podem observar el tren d'impulsos utilitzat per mostrejar i també podem observar els impulsos de la transformada de Fourier.

Un impuls, denotat com d(t), es una senyal teòric infinitament curt i potent i té una integral de 1. Una propietat interessant del tren d'impulsos amb un període Ts és que, la transformada de Fourier és un tren d'impulsos amb un període 1/Ts.

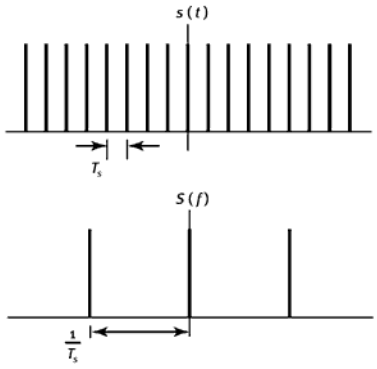


Figura A5.5B

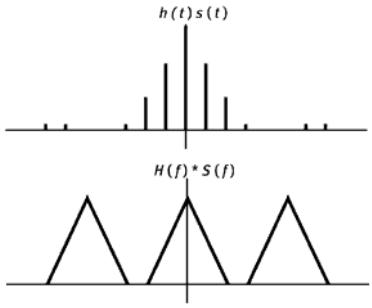


Figura A5.5C

L'efecte de mostreig h(t) amb la funció de mostreig s(t) es pot observar a la figura A5.5C. En el domini de temps, el mostreig pot ser pensat com multiplicar h(t) per s(t), i en el domini de freqüència, pot ser pensat com la convolució de H(t) i S(t).

$$h(t)s(t) \supset H(f) * S(f)$$

Eq. 3

La convolució de qualsevol funció f(x) i g(x) està donada per

$$f(x) * g(x) = \int_{-\infty}^{\infty} f(u)g(x - u)du$$

Eq. 4

La convolució d'un sol impuls a $t=t_0$, amb h(t) és justament el valor h(t) desplaçat a un lloc.

$$h(t) * \delta(t - t_0) = \int_{-\infty}^{\infty} h(u)\delta(t - t_0 - u)du = h(t - t_0)$$

Eq. 5

Llavors podem aplicar el resultat de l'equació 5 per trobar la convolució a H(f) i S(f).

$$h(t)s(t) \supset \frac{1}{T_S} \sum H\left(\frac{f - n}{T_S}\right)$$

Eq. 6

La equació 6 simplement significa que el resultat de la convolució H(f) i S(f) és tal que H(f) és duplicat en intervals de 1/Ts, com podem veure en la figura A5.5C. La funció sinc^2x és de banda limitada (perquè l'amplada de banda és limitat) a fmax,, on fs = 1/Ts. Així doncs, es aquí d'on ve el denominat límit de Nyquist.

A la figura A5.5D podem observar que si la freqüència de mostreig esta per sota de 2fmax, llavors l'espectre adjacent es solapa a altres freqüències, i aquestes freqüències es perden al senyal resultant. No obstant, en ves de desaparèixer les senyals de freqüència alta al complet apareixen a baixes freqüències com l'aliasing i d'aquí és d'on s'origina el terme "aliasing". Per eliminar l'aliasing (vingui d'on vingui) cal que cada senyal al mostrejar-se sigui de banda limitada i a menys de 2fs o que la freqüència de mostreig sigui més gran a 2fmax

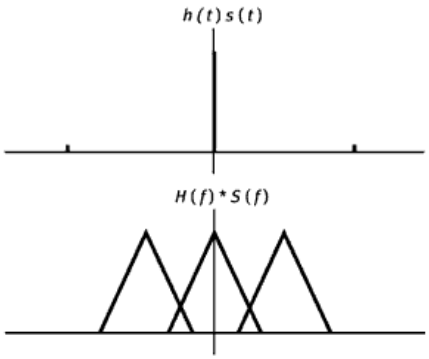


Figura A5.5D

Tot seguit es veurà com el MIP-Mapping ajuda a reduir aquestes zones d'aliasing a la nostra textura. Recordem que la tècnica "texture-mapping" esta dissenyada per incrementar el realisme i el detall en escenes. D'altra banda, els detalls sútils a la textura són components d'alta freqüència que poden provocar problemes d'aliasing. Realment, no podem tractar les freqüències de mostreig a la regió de renderització durant el mapeig de textura (du i dv), hem de filtrar les textures per eliminar els detalls d'alta freqüència durant el desenvolupament.

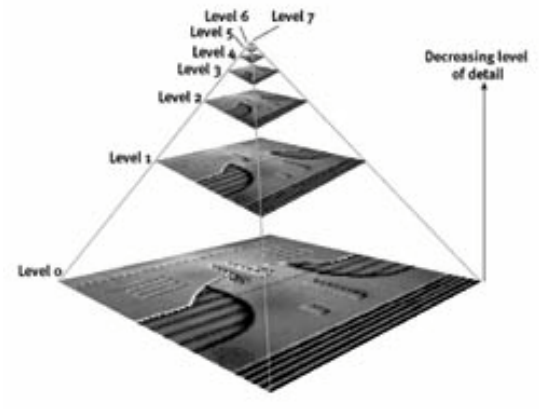


Figura A5.6

En algunes targetes gràfiques 3D, com la Nvidia Riva 128, generen automàticament els MIP-Maps quan les textures es carreguen a memòria. A la figura A5.6 s'il·lustra la piràmide formada per MIP-maps d'una textura de 64x64 pixels. Com podem observar, el Nivell De Detall (o NDD) decrementa quan s'incrementa el nivell MIP-map. Un cop filtrades les textures, tot el que s'ha de fer a temps de execució és seleccionar el MIP-Map correcte (NDD) per la textura desitjada i donar-ho al render. Però abans s'han de saber els mètodes per generar els MIP-maps.

Hi ha moltes maneres de generar els MIP-maps per una textura. La primera opció és simplement pregenerar-les utilitzant alguna eina de procés gràfic tal com *PhotoShop*®. Una altra és generar els MIP-Maps a temps d'execució. La pregeneració de MIP-Maps requereix un 30 % de la grandària de textura, però dona un control de qualitat sobre la seva generació i permet afegir afectes i detalls a cada nivell MIP-Map. Indiferent de quin mètode es triï mencionats anteriorment, els MIP-Maps requereixen un 30% de capacitat d'espai a temps d'execució respecte la grandària de la textura original. Per aquest fet, es pot considerar d'importància en el requeriments de memòria del joc o aplicació.

• Generació de MIP-MAPS de 8 bits

Generar MIP-Maps per textures de 8 bits és un procès molt simple i hi ha moltes tècniques de filtrat que poden aplicar-se davant de la generació de MIP-Maps. Normalment, es suficient utilitzar una caixa de filtrat estàndard. El primer nivell MIP-Map es pot generar agafant la informació crua dels pixels (raw input) i copiar-la directament dins l'estructura d'informació MIP-Map.

La creació de la resta de MIP-Maps segons la textura a la que es fa referència, és un procés iteratiu. Cada nivell MIP-map es genera amb la informació del seu nivell precedecent. Quan el nivell MIP-Map és creat, es guarda consecutivament a memòria i també es guarda adequadament el punter del nivell MIP-Map perquè a temps d'execució s'hi pogui accedir de forma immediata.

El primer pas per generar un nou valor de pixel és calcular la mitjana de color dels 4 pixels corresponents al nivell MIP-Map precedent.

Tots els gràfics de 8 bits tenen associats una paleta, per això un cop s'ha calculat el nou valor de pixel, s'haurà de buscar l'entrada de paleta amb el valor de color més pròxim a esmentat nou valor (veure tècnica *nearest color*, annex a6.1).

La cerca del color es simple, però consumeix molt temps. Afortunadament, aquest pas es requereix només durant la inicialització del MIP-Map, previa a l'execució del joc. No obstant, si es vol realitzar altres efectes durant la renderització (tal com un filtratge biliniar o triliniear), el procés de cerca esdevindrà molt lent. En aquest cas, necessitem utilitzar textures de 16 o 24 bits, però donat que moltes targetes gràfiques del mercat suporten profunditat de color de 16, es farà referència a la generació de MIP-Maps per textures de 16 bits.

• Generació de MIP-MAPS de 16 bits

El procés de construcció de MIP-Maps per textures de 16 bits és molt similar al utilitzat per els textures de 8 bits, tal com podem veure al llistat 4. Com que les textures de 16 bits no requereixen paleta, amb els 4 pixels del nivell precedent s'obté la mitjana del nou valor de pixel. Un problema que pot sorgir es que el MIP-map pot esdevenir fosc a cada NDD successor. Aquest efecte es pot compensar si s'afegeix una quantitat petita a cada component del color (RGB), però aquesta compensació és normalment innecessària si la pèrdua de color durant el procés és petita.

A la figura A5.7 (sota) podem observar alguns problemes que podem trobar a l'hora de seleccionar quin NDD aplicar a temps d'execució. A la figura, a l'esquerra una textura rectangular mapeja un triangle a l'espai de textura i, a la dreta, el triangle mapejat es transformat en un quadrilàter a l'espai de pantalla. Podem observar que, la projecció de perspectiva en textura causa que els texels individuals esdevinguin quadrilàters diferents en grandària. En el cas de tenir un polígon transformat de forma planera, com el de la figura, es crucial determinar el millor NDD per bons resultats visuals. Si s'escull un NDD massa alt, implica la utilització de textures grans i per conseqüent apareixerà aliasing durant el mapeig. Per altra banda, si NDD implica la utilització de textures petites i per conseqüent la imatge apareixerà borrosa. Per exemple, el NDD de la textura de la figura A5.8 escollit es massa baix, observant els texels llargs visibles a la part de la imatge engrandida.

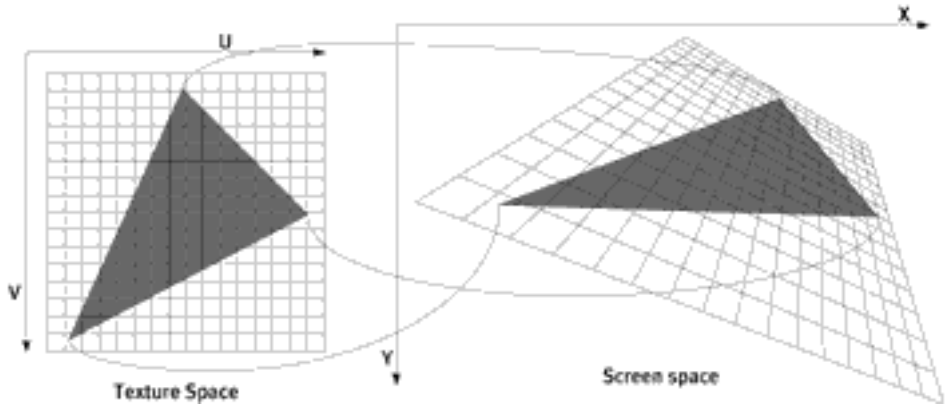


Figura A5.7

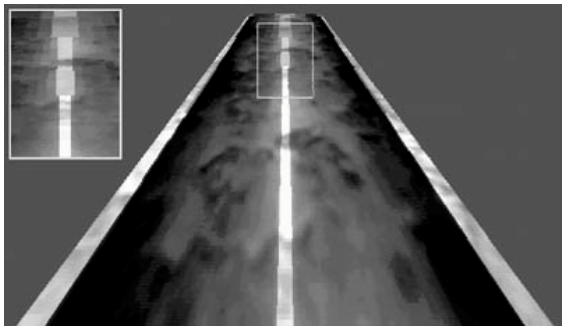


Figura A5.8

Actualment existeixen multitud de mètodes per seleccionar el NDD, tots ells amb avantatges i desavantatges. En aquest annex, s'explicarà dos mètodes generalment coneguts: El NDD basat en l'àrea de textura a l'espai d'imatge i la selecció del NDD utilitzant les coordenades u i v.

Cal considerar que, és possible mapejar cada pixel de la pantalla amb diferents texels de textura. Com a resultat, la selecció NND correcte requereix el càlcul de NDD per cada pixel. El càlcul del NDD pot ser una mica lent; consecutivament molts renders per software (i també algunes targetes acceleradores velles) calculen el NDD en base de polígons o triangles. Un avantatge afegit en la selecció del MIP-map per polígons, especialment en renders basats per software, és que es pot utilitzar versions petites de textures per polígons distants, ajudant a reduir la quantitat de cache del processador que es requereix durant les operacions de mapeig. D'altra banda, la selecció NDD per pixel permet fer altres cases amb MIP-mapping, incluent el mostreig de punts, filtre bilinear amb un únic NDD, o filtre trilinear entre dos NDD molt iguals.

La selecció NDD per polígon és un mètode molt costos perquè només es pot fer la selecció d'un MIP-map un cop per polígon. No obstant, hi ha un parell de inconvenients per aquesta selecció. El primer problema es que, per polígons adjacents que comparteixen la mateixa textura podrien ser dibuixats utilitzant diferents NDDs; amb això apareixeria a un discontinuïtat a la textura quan es mostrés per pantalla; aquest fet se'l anomena MIP-banding. La figura A5.9 s'observa una petita quantitat de MIP-banding que passa a causa de la utilització de MIP-mapping per triangle.

Un altre problema és que el mapeig a pantalla pot visalitzar talls quan el NDD canvia a causa del moviment del observador (o polígon).

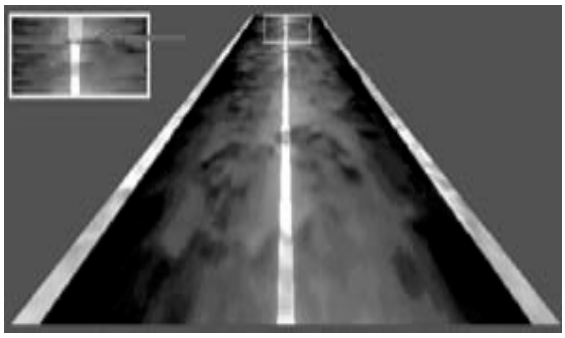


Figura A5.9

• Selecció NDD basat per l'àrea

La selecció de NND basat per l'àrea complementa les tècniques MIP-mapping per polígon. En aquest mètode, es selecciona el NDD per la quantitat de textura comprimida que serà aplicada a la textura dins l'espai de pantalla. Per determinar la compressió de textura apropiada, s'ha de calcular l'àrea del polígon a l'àrea del polígon a l'espai de pantalla i l'àrea dins l'espai de textura, de textura que serà mapejada al polígon. Com podem observar en el llistat 5, podem determinar la proporció de texels als pixels a llavors determinar quin NDD utilitzar. Les dimensions u i v de cada NDD successiu tenen una quarta part de l'àrea del nivell precedent. Durant la selecció del NDD, hem de pujar un nivell a la piràmide de MIP-maps (figura A5.6) per cada múltiple de quatre quan l'àrea de texel que sigui més gran que l'àrea de pixel. Per exemple, si la proporció texel-a-pixel és 7:1, haurem de seleccionar el primer nivell. Un cop s'hagi seleccionat el NDD, podrem passar el punter del NDD correcte, amb les dimensions del MIP-map, per les nostres rutines de mapeig de textura. Un problema amb qualsevol mètode que utilitzi l'àrea del polígon projectat i l'àrea de textura coma base de la selecció del NDD és que l'aliasing tendirà a aparèixer tant bon punt el polígon projectat sigui molt petit, degut a la natura anisotròpica de la textura comprimida (que és, la textura més comprimida en una dimensió que una altre).

MIP-Mapping-per-pixel ofereix un millor control de la selecció del NDD que el MIP-Mapping-per-polígon, i addicionalment també permet filtrat de textures – però amb un petit cost addicional. Tots els mètodes per-pixel requereixen guardar el MIP-map sencer de textura a la memòria, i obtenir la selecció del NDD pot significar una reducció a la velocitat d'execució (*performance*).

Afortunadament, moltes tarjes 3D d'avui en día suporten MIP-mapping per-pixel amb filtrat bilinear (poques més de les últimes suporten filtrat trilinear), tècniques que més tard veurem.
Encara que podem utilitzar la selecció NDD basat en l'àrea (necessitem calcular l'àrea de textura per sota cada pixel en ves de per cada polígon), veurem un mètode més precís.

• Selecció NDD basat en la compressió dels extrems

A 1983, Paul Heckbert va estudiar moltes tècniques de càlcul NDD abans de decidir que tals tècniques basades en la compressió que la textura, es veuen amb més detall quan es troba al voltant dels extrems del pixel. La figura A5.10, es mostra un pixel a l'espai de pantalla i el corresponent paral·lelogram a l'espai de textura. Per prevenir aliasing del que pogui passar, volem seleccionar el NDD basat en la màxima compressió dels límits a l'espai de textura. Això correspon a la màxima llargada d'un costat a l'espai de textura, el qual és donat per l'equació 7.

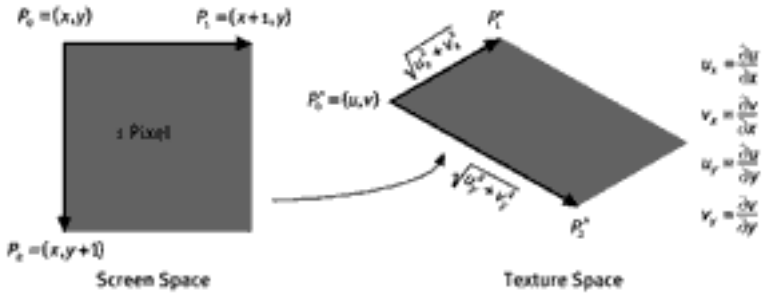


Figura A5.10. Un pixel projectat a l'espai de textura

$$\max\left(\sqrt{u_x^2 + v_x^2}, \sqrt{u_y^2 + v_y^2}\right)$$

Eq. 7

Els valors de u_x, u_v, v_x, i v_y són donades per quatre derivades parcials. El fet de conèixer com calcular els valors u i v per qualsevol pixel a la pantalla, ens permetutilitzar aquest coneixement per determinar les derivatives parcials. Podem saber, els pendents u/z, v/z i 1/z en x i en y i els valors de començament u/z, v/z i 1/z a l'origen de pantalla i els valors u i v per la textura en qualsevol pixel, amb les equacions 8 i 9. La notació de equacions 8 i 9 s'han extret de les sèries "perspective texture mapping" per Chris Hecker's, el qual pot trobar-se a la web. ([HCK97A])

$$u = \frac{dUOverZdX * x + dUOverZdY * y + UOverZ_0}{dOneOverZdX * x + dOneOverZdY * y + OneOverZ_0} = \frac{UOverZ}{Z}$$

Eq. 8

$$v = \frac{dVOverZdX * x + dVOverZdY * y + VOverZ_0}{dOneOverZdX * x + dOneOverZdY * y + OneOverZ_0} = \frac{VOverZ}{Z}$$

Eq. 9

Podem utilitzar aquests resultats per trobar les derivades parcials, com podem veure des de l'equació 10 fins a la 13.

$$u_x = \frac{Z * dUOverZdX - UOverZ * dOneOverZdX}{Z^2} = \frac{c + ay}{Z^2}$$

Eq. 10

$$v_x = \frac{Z * dVOverZdX - VOverZ * dOneOverZdX}{Z^2} = \frac{d + by}{Z^2}$$

Eq. 11

$$u_y = \frac{Z.dUOverZdY - UOverZ.dOneOverZdY}{Z^2} = \frac{e + ay}{Z^2}$$

Eq. 12

$$v_x = \frac{Z.dVOverZdY - VOverZ.dOneOverZdY}{Z^2} = \frac{f + by}{Z^2}$$

Eq. 13

On a, b, c, d, e, i f són obtinguts per les des de equació 14 fins a la l'equació 19.

a = dUOverZdX*dOneOverZdY-dOneOverZdX*dUOverZdY

Eq. 14

b = dVOverZdX*dOneOverZdY-dOneOverZdX*dVOverZdY

Eq. 15

c = dUOverZdX*OneOverZ_0-dOneOverZdX*UOverZ_0

Eq. 16

d = dVOverZdX*OneOverZ_0-dOneOverZdX*VOverZ_0

Eq. 17

e = dUOverZdY*OneOverZ_0-dOneOverZdY*UOverZ_0

Eq. 18

$$f = dVOverZdY*OneOverZ_0 - dOneOverZdY*VOverZ_0$$

Eq. 19

Un important punt a mencionar és que els numeradors de les derivades parcials u_x i v_x són funcions de només d'y, i els numeradors de les derivades parcials u_v i v_v són funcions de només d'x. Els valors de a, b, c, d, e, i f són calculats un cop per polígon, juntament amb els valors de pendent a utilitzar, vist al llistat 6. Finalment, la fórmula per trobar la compressió màxima dels límits es donar a l'equació 20.

$$\text{Compresió} = \frac{1}{Z^2} \max(x_\ell, y_\ell)$$

On:

$$x_\ell = \left(\sqrt{(c + ay)^2 + (d + by)^2}\right)$$
$$y_\ell = \left(\sqrt{(e - ax)^2 - (f - bx)^2}\right)$$

Eq. 20

A primer cop d'ull, veiem veure que necessitem fer el càlcul d'una arrel quadrada per cada pixel. No obstant, si s'observa amb més detall, veiem que només necessitem computar y_ℓ un cop per el rang de valors d'x del polígon. Per això, només es necessita computar x un cop per scan line. En el llistat 7 es pot veure com precomputem els valors y per un rang de valors x duran la posta a punt normal per "texture mapping", i també només necessitem calcular x un cop per scan line. No ens hem de preucopar per la divisió que es requeix en el denominador, perquè ja s'havia fet servir a la taula precalculada de textura. Així doncs, la sobrecarrega per el càlcul de compressió amb "texture mapping" en el bucle principal tindrà justament dos multiplicacions i una comparació. Ara que coneixem com calcular la compressió del límit, podem aplicar la selecció NDD per-pixel la nostre rutina "texture-mapping" utilitzant, mostreig de punt, filtrat bilinear, i filtrat trilinear.

• Mostra-punt per-pixel MIP-MAPPING

El mostreig de punt es la forma més simple del MIP-mapping per pixel, i tant com es pot veure en el llistat 8, no hi ha molta diferencia entre el nostre bucle "texture mapping" normal i el que utilitza mostreig de punt.

Un cop trobem la quantitat de compressió del límit de l'actual pixel, necessitem determinar el correcte NDD. La compressió del rang de valors crua (raw) va des de 0 a 1, però necessitem engrandir-ho per la dimensió de textura per aconseguir una altura significativa en la nostra piràmide MIP-map. Un cop tinguem l'alçada, determinarem el NDD correcte pujant un nivell en la piràmide per cada potència de dos quan l'altura sigui més gran que l'actual. Llavors, utilitzarem la nostra taula precalculada de NDD per aconseguir el punter de la nostra textura i l'accés del texel correcte com l'usual. A la figura A5.11 es veu el mateix objecte que hem utilitzat per generar la figura A5.9, però en aquest cop apliquem mostreig-punt MIP-mapping. Tal com es pot veure a la figura, el principal problema amb el MIP-mapping de punt-mostrejat es que el MIP-banding es clarament visible als punts on succeixien diferents transicions de NDD Això passa perquè els pixels adjacents poden tenir diferents NDD, per això apareix una discontinuïtat quan combinem diferents NDD.

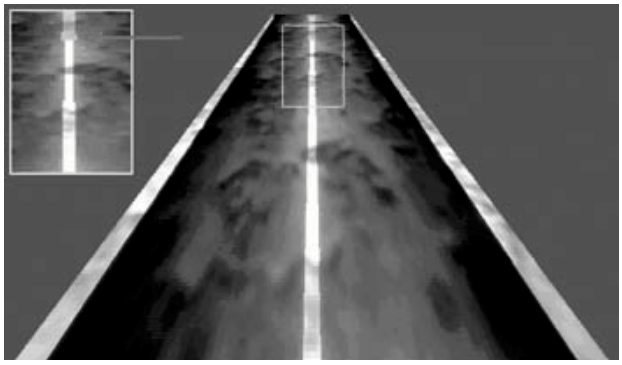


Figura A5.11

• MIP-Mapping per-filtre basat en filtrat-Bilinear

El filtrat bilinear intenta reduir més qualsevol error de aliasing present en una escena per la mitjana de valors dels 4 pixels que són més pròxims als valors u i v de textura reals per cada pixel. Tant com es pots veure a la figura A5.12 i el llistat 9, la interpolació bilinear pot ser implementada utilitzant arbres de interpolacions linears. Es calcula el NDD correcte i es recupera el punter de la nostra textura en el mateix camí que hem fet a l'hora de fer-ho amb mostreig de punt. No obstant, llavors haurem de recuperar quatre valors de textura i aplicar la interpolació bilinear a cada component del color per generar el nou valor de pixel. A la figura A5.13 s'observa la nostra carretera dels exemples, després d'aplicar un MIP-mapping amb filtrat bilinear. Encara que a la figura A5.13 hi ha una millora sobre la figura A5.11, encara s'ha produït MIP-banding. No s'ha pogut eliminar les dicontinuitats causades quan hi ha la combinació entre diferents NDDs.

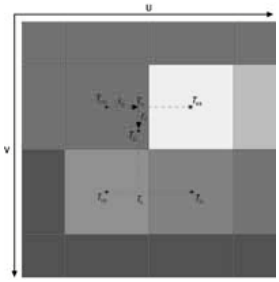


Figura A5.12 *càlcul del filtrat bilinear*

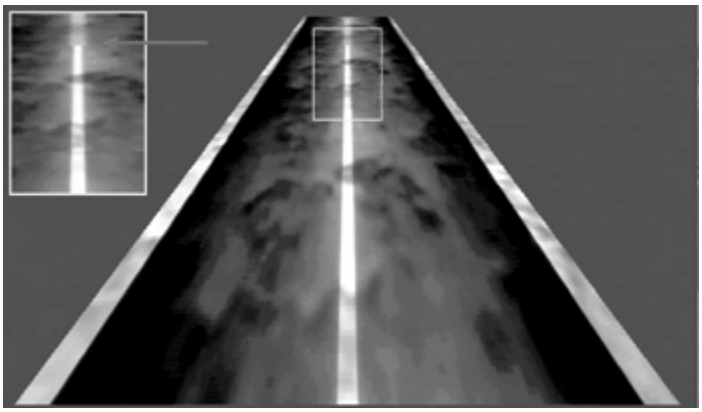


Figura A5.13 *Carretera renderitzada amb MIP-mapping per-pixel i filtrat bilinear*

• MIP-mapping per-pixel amb filtrat trilinear

L'actual state-of-art de MIP-mapping per hardware-accelerat 3D es el filtrat trilinear. El filtrat trilinear procura de eliminar els problemes associats amb MIP-banding fent una suau transparència entre els diferents NDDs. Com podem veure al llistat 10, un cop tenim calculat el correcte NDD procedint de la mateixa forma com s'havia fet en mostrejat de punt, es retorna els punters al NDD calculat i llavors el pròxim NDD per sota (el pròxim nivell en la piràmide). La interpolació trilinear es implementada utilitzant vuit interpolacions linears. Es comença per portar les interpolacions bilinears per separat per cada NDDs seleccionat, finalment s'acaba per la interpolació lineal entre els dos NDDs. Com podem veure a la figura A5.14, la interpolació trilinear dona un resultat suau en la transició entre els NDDs. Desafortunadament, aquesta característica vé amb un considerable cost: La implementació íntegra del MIP-mapping amb filtrat trilinear present aquí requereix 8 accessos de textura per cada pixel i una considerable quantitat de taules precalculades de textura per el salvat del valor de texel entre les iteracions del bucle, així doncs portar a cap una taxa de quadres per segon acceptable amb un filtrat trilinear per software és molt difícil.

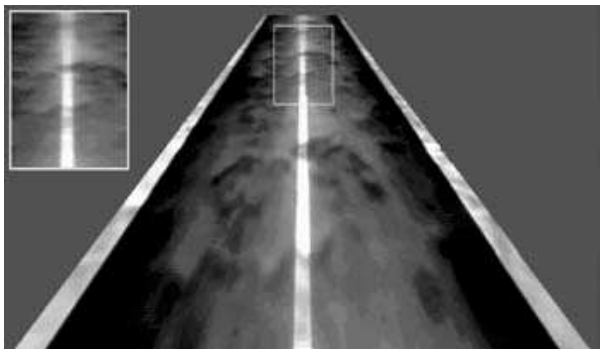


Figura A5.14 *Carretera renderitzada amb MIP-mapping per-pixel i filtrat trilinear*

Malauradament, encara que la sortida del nostre render utilitzant MIP-mapping amb filtrat trilinear sigui significantment millor que el l'antic "texture mapping", encara és imperfecte. El principal gran defecte que encara hi ha és que, com s'ha mencionat al principi, ignorem la realitat que la compressió de textura és anisotropic. Quan seleccionem NDDs utilitzant la màxima compressió amb un sól límit, però que passa si hi ha una gran diferencia en la quantitat de compressió entre cada extrem ? En aquest cas, el NDD seleccionat serà massa baix per l'ultim extrem comprimit, i la nostra escena apareixerà borrosa. Es pot veure clarament aquest efecte a la figura A5.15, la quan es una captura de pantalla de la demo CHAOSVR i que va ser renderitzada utilitzant una targeta basada amb la pastilla 3Dfx's Voodoo2. EL problema presentat apareixerà a qualsevol acceleradora 3D que utilitzi mètodes similars als que mateixos que nosaltres hem desenvolupat aquí per el càlcul del NDD. Clarament, el pròxim pas per intentar la renderització precisa serà adoptar alguna forma de filtrat anisotropica.

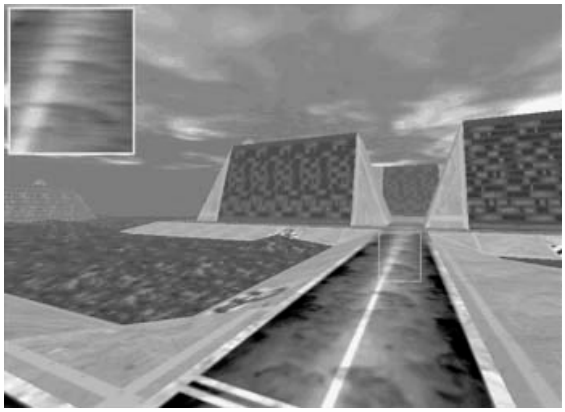


Figura A5.15 *Captura de pantalla de CHAOSVR renderitzada utilitzant un filtra trilinear amb el chipset Voodoo2*

ANNEX 6: Tècniques de paleta

Actualment, tots els sistemes de vídeo en les aplicacions operen amb una densitat de color alta (des de 16 o 32 bits), aquests modes anomenats modes de color empaquetats o RGB. Però, a pesar d'això, encara existeixen moltes aplicacions (jocs en la seva major part) que treballen amb modes de vídeo paletitzats (8 bits) donat que són econòmics en espai. Podem pensar que la qualitat de color no serà gaire bona ja que en els modes paleta només disposa de 256 colors únics.

En aquest annex explicarem tècniques de paleta per aconseguir reduir una imatge amb profunditat de 32 bits a 256 colors, obtenint una bona qualitat de imatge i altres tècniques d'ombrejat i transparència. La clau per aconseguir-ho és la tècnica del tècnica *nearest color* (color més pròxim).

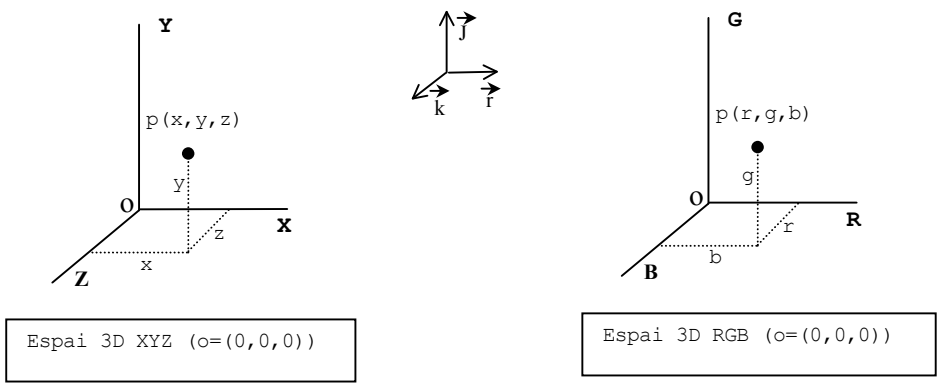
A6.1 Nearest color

El problema del *nearest color* o *best match*, consisteix en trobar el color més semblant a un color RGB dins una paleta de N colors. Suposem que disposem d'una paleta com la següent (3colors, del 0 al 2):

Paleta = (0,0,0), (0,0,1), (53, 23, 23)

Si es busca el més semblant al (53, 20, 20), òbviament el color més semblant serà el nº 2 (color (53, 23, 23)), encara que no es exactament igual al cercat, sí que es pràcticament molt

ESPAI RGB



Les components RGB d'un color poden tractar-se com vectors dins d'un sistema de coordenades cartesianes en 3D, utilitzant com coeficients dels vectors unitaris els valors R,G,B del color:

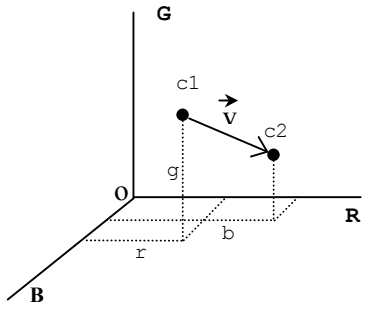
$$vector = r \cdot \vec{r} + g \cdot \vec{J} + b \cdot \vec{k}$$

Aquest model de representació matemàtica ve molt bé per realitzar algunes operacions amb els colors, com veurem en el cas de la cerca del color més proper a un donat (tècnica *nearest color*).

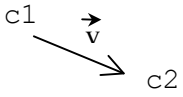
Figura a6.1 *Espai RGB*

igual en tonalitat i lluminositat en pantalla, i és molt més semblant que la resta de colors dins la paleta. La pregunta és: ¿ Com cercar el color més semblant ? Bé, concretant una mica en el llenguatge, s'ha de dir que per trobar el color més semblant a un altre significa trobar el que tingui menys diferència amb aquest. Si tractem els colors com punts en un sistema de coordenades 3D (figura a6.1), ens adonarem de que quant més semblants són els colors, més propers es troben entre ells, i viceversa. En altres paraules, quant menor és la distància entre els colors, més semblants són.

DISTANCIA RGB



c1 = (r1, g1, b1)
c2 = (r2, g2, b2)



Dist(c1, c2) = $|\vec{v}|$

Al tractar als colors com punts en un espai RGB, la distància entre ells correspon al mòdul del vector que els uneix:

$$Dist = \sqrt{(c2 - c1)^2 + (b2 - b1)^2 + (g2 - g1)^2}$$

Figura a6.2 *distància RGB*

La distància entre 2 punts és, com es pot veure a la figura a6.2, el mòdul del vector que els uneix. Per trobar el color més semblant a un altre, es recorre la paleta des de el color 0 fins el 255 (en cas de disposar d'un paleta de 256 colors), calculant les distàncies del nostre color RGB amb el de la paleta. Aquell que ens doni la menor distància de tota la paleta serà el més semblant a l'original (encara que no té perquè ser una bona aproximació).

El que realment es busca amb la tècnica *nearest color* és una relació entre les distàncies (no ens interessa que sigui exacte, tan sols quina és la menor), per el que podem prescindir de l'arrel quadrada per accelerar el procés, utilitzant tan sols els quadrats de les components del vector.

$$Dist^2 = (c2-c1)^2-(b2-b1)^2-(g2-g1)^2$$

D'aquesta manera, una rutina/algorisme de cerca de color més proper a una altre quedaria com el que veiem al següent llistat de codi C.

```
//-----------------------------------------------------
// Algunas macros útils per NearestColor:
//-----------------------------------------------------
#define RED(pal,col)   ((pal)[(col)*3])
#define GREEN(pal,col) ((pal)[(col)*3+1])
#define BLUE(pal,col)  ((pal)[(col)*3+2])
#define byte unsigned char


//-----------------------------------------------------
// int NearestColor( r, g, b, paleta );
// Busca el color més semblant a rgb en paleta.
//
// Torna -1 en cas de error (improbable).
//-----------------------------------------------------
long NearestColor( byte r, byte g, byte b, byte *paleta )
{
  long result = -1;       // color mínim fins ara
  long minim = 200000;   // distància mínima "" ""
  long rd, gd, bd;        // components vector c2-c1
  long dist, i;           // distància quadrada i index bucle

  // repetimos para los 256 colores:
  for( i=0 ; i<256 ; i++ )
  {
    // components del vector: r2-r1, etc.
    rd = r-RED(paleta, i);
    gd = r-GREEN(paleta, i);
    bd = r-BLUE(paleta, i);

    // calculem la distància quadrada:
    dist = rd*rd + gd*gd + bd*bd;

    // comprobem si es el menor fins ara:
    if( dist <minim )
    {
      // es la menor: ho anotem i seguim
      minim = dist ;
     result = i;
    }
  }

 return( result );
}
```

Llistat a6.1

A6.2 Reducció del color a 8 bits

Una de les aplicacions que té la tècnica *nearest color* és la reducció d'imatge de profunditat 15/16/24/32 bits (RGB) a 256 colors (paletitzat), amb una pèrdua de qualitat d'imatge mínima.

El problema tal i com s'ha presentat (convertir una imatge RGB a 256 colors) resulta molt difícil abordar, però en la pràctica són poques les imatges a les quals utilitzen tal numero de colors diferents. Una imatge de 320x200 RGB pot comptar, per exemple, amb 10.000 colors diferents encara que tingui 64.000 pixels, ja que molts colors poden esta repetits (per exemple, una foto d'un cel contindrà molts blaus repetits). A part, molts d'aquests colors poden ser colors molt propers entre ells (per exemple, 2 píxels (120,0,0) i (122,0,0) són difícilment distingibles i els veurem com 2 colors iguals).

Encara així, la rutina de conversió que de implementar s'haurà de convertir una imatge d'un numero elevat de colors en tan sols 256, el qual es veu, a primera vista, complicat.

Per realitzar la conversió recorrem a les tècniques de quantificació (*color quantitzation*), de les quals explicarem la més senzilla de totes. Es necessari disposar de 2 búfers (una que disposi de l'imatge original i l'altre que disposi de l'imatge destí i un vector on s'emmagatzemi la paleta que s'ha de crear per la imatge (la més òptima per el mateix).

La tècnica de quantificació que explicarem és la més senzilla de totes, doncs el mètode consisteix en un **histograma de freqüències**. Aquest mètode es basa en els 2 fenòmens comentats al l'exemple de conversió anterior. La creació d'un histograma de freqüències el mostrem al següent llistat de codi C:

```
Char frequencia[256][256][256]


for( ... altura_img ... )
      for( ... amplada_img ... )
            r,g,b = GetPixel();
            frecuencia[r][g][b]++;
```

Llistat a6.2

El vector *frequencia[...][...][...]* s'utilitza com un indicador de freqüència d'aparició de cada color: cada element del vector indica el numero de pixels de la imatge per cada color RGB, es a dir, si en una imatge hi ha 1000 colors negres (0,0,0), llavors *frequencia[0][0][0] = 1000.*

Havent aconseguit omplir l'histograma de freqüències, l'últim pas que queda es seleccionar la quantitat dels 256 colors més freqüents, per guardar-los a la paleta final.
L'algorisme *ObtenirPaleta256* vindria a ser el següent:

```
const NColors = 256

Algorisme ObtenirPaleta256

      Per r desde 0 a 256 fer
            Per g desde 0 a 256 fer
                  Per b desde 0 a 256 fer

                        Frec ← frequencia[r][g][b]

                        { Si trobem un índex de color menys freqüent
                          que Frec, l'inserim a la paleta }
                        (i,trobat) ← (0,FALS)
                        mentre i < NColors i no trobat fer

                              Si (FrecPaleta[i] < Frec) llavors

                                    PaletaDesti[i]← (r,g,b)
                                    FrecColor[i]← Frec
                                    Trobat ← CERT

                              fsi
                              i← i+1

                        Fmentre
                  FPer
            Fper
      FPer
FAlgorisme
```

Llistat a6.3

En el llistat a6.3, podem observar la creació de la paleta destí. Bàsicament, disposem de 2 bùfers: *PaletaDesti[...]* i *FrecColor[...]* de 256 entrades cadascuna. En un s'hi guardarà els valors RGB de la paleta de 256 colors final i en l'altre la freqüència actual de cada color i, inicialment, tots els valors del vector estaran inicialitzats a 0.

Per cada iteració del triple bucle, es captura la freqüència del color RGB. Es busca a la taula de freqüències, el color menys freqüent que *Frec*. Si en trobem un de menys freqüent a la taula *FrecColor[...]*, inserim el color RGB en *PaletaDesti[...]* i actualitzarem *FrecPaleta[...]*.

Selecció de colors semblants

A l'hora fer la quantificació d'una imatge ocupat majoritàriament d'un cel amb molts gradients de blaus i, en menor cantitat, el paisatge d'una selva (amb molta diversitat de verds), no aconseguiríem una quantificació del tot perfecte. El problema es que al tenir a la major part el cel que la selva, podríem tenir colors més freqüents en blaus que en verds i havent acabada la quantificació, podríem observar una selva de caràcter blavós.

El problema és a que guardem directament els colors que mes freqüentment es repeteixen. Una manera de millorar aquests casos es, establir **una distància mínima** per que els colors amb més freqüència s'adhereixin en d'altres que siguin semblants.

Per exemple, el cas que volguéssim aconseguir adherir colors en una distancia mínima de 20 unitats dins el tribucle del codi a6.3 caldria fer:

```
Per r ...
 Per g ...
    Per b ...

     Frec ← frequencia[r][g][b]
     (i,trobat) ← (0,FALS)

      mentre i < NColors i no trobat fer

        (diff_r,diff_g,diff_b) ← abs(r-PaletaDesti[i].r),abs(g-PaletaDesti[i].g), abs(b-PaletaDesti[i].b)

        {distancia}
        dist ← sqrt(diff_r + diff_g + diff_b)

        { Si la frqüència de color és menys freqüent
        que Frec i la distancia entre aquest es
        més gran que 20, l'inserim a la paleta }
        Si (dist > 20) llavors

           Si (FrecPaleta[i] < Frec)

                 PaletaDesti[i] ← (r,g,b)
                 FrecColor[i]← Frec
                 Trobat ← CERT

           Altrament {Com que es un color semblant actualitzem la taula frequencies}

                 (r_med, g_med, b_med) ← ( (r + PaletaDesti[i].r)/2 , (g + PaletaDesti[i].g)/2 , (b + PaletaDesti[i].b)/2 )

                 PaletaDesti[i] ← (r_med,g_med,b_med)
                 FrecColor[i] ← FrecColor[i] + Frec
                                                        FSi
        FSi

        i ← i+1

      Fmentre
    Fper
  Fper
FPer
```

Llistat a6.4

Podem observar al llistat a6.4 que, si troba un color semblant (dist < 20), es considerar com un color igual al trobat i, per això, s'actualitza la taula de freqüència del color (*FrecColor[...]*) sumant la freqüència *Frec*. També podem observar que es fa la mitjana dels colors RGB de la paleta amb l'actual. Això es fa perquè la distribució dels colors semblants en proporció. En canvi, si el color és molt diferent (dist > 20), i la elevada freqüència del color ho permet, es permetrà la inserció del nou color a la paleta.

A6.2 Creació d'una taula de intensitats

Si en la nostra aplicació o joc, desitgem crear efectes de llum com, per exemple, un *flare*, hem de crear una rutina que sigui capaç de pintar un pixel amb intensitat.

Figura a6.5 *Escala de grisos representat pel seu nivell de intensitat des del negre (nivell 0) al blanc (nivell 255).*

Suposem que volem saber els 256 nivells de intensitat que compren una escala de grisos de la figura a6.5. Ho podem fer mitjançant un factor intensitat que atenua o amplifica un color en concret,

$$factor_intensitat = \frac{Intensitat}{256} \cdot 2 \qquad \textit{Equació a6.1}$$

$$factor_intensitat \in [0.0, 2.0)$$

Aquest factor intensitat representa la proporció de cada component RGB del color original que cal sumar per tal d'obtenir el resultat atenuat quan *factor_intensitat* $\in$ [0.0,1.0) o amplificat quan $\in$ [1.0, 2.0).

```
Funcio COLOR_AfegirIntensitat(e intensitat e/s Color)

        Factor_intensitat ← (intensitat/256)*2.0

        Si (Color_Amb_Intensitat.R + factor_intensitat* Color_Amb_Intensitat.R) ≤ 255) llavors
                Color_Amb_Intensitat.R ← Color_Amb_Intensitat.R + factor_intensitat* Color_Amb_Intensitat.R
        Altrament
                Color_Amb_Intensitat.R ← 255
        FSi

        Si (Color_Amb_Intensitat.G + factor_intensitat* Color_Amb_Intensitat.G) ≤ 255) llavors
                Color_Amb_Intensitat.G ← Color_Amb_Intensitat.G + factor_intensitat* Color_Amb_Intensitat.G
        Altrament
                Color_Amb_Intensitat.G ← 255
        FSi

        Si (Color_Amb_Intensitat.B + factor_intensitat* Color_Amb_Intensitat.B) ≤ 255) llavors
                Color_Amb_Intensitat.B ← Color_Amb_Intensitat.B + factor_intensitat* Color_Amb_Intensitat.B
        Altrament
                Color_Amb_Intensitat.B ← 255
        FSi

        Retorna Color_Amb_Intensitat
FAccio
```

Llistat a6.5

Doncs, amb aquesta rutina del llistat a6.5, ja podem pintar qualsevol dels pixels amb un nivell de intensitat. Però aquesta rutina només funciona quan el nostre sistema de vídeo és RGB.

Per aconseguir efectes de llum en un sistema de vídeo basat en paletes (8 bpp), no podem aplicar la funció del llistat a6.5, tots els valors de la imatge de vídeo apunten al color indexat de la paleta. L'únic camí per aconseguir un efecte de llum en 8 bpp, es creant una taula bidimensional de 256x256 de valors índex de paleta. Per una part indexarem el color que volem i per l'altra la intensitat. El resultat apuntarà l'índex de paleta, on el color representant contindrà el color amb la intensitat seleccionada aproximada (figura a6.6).

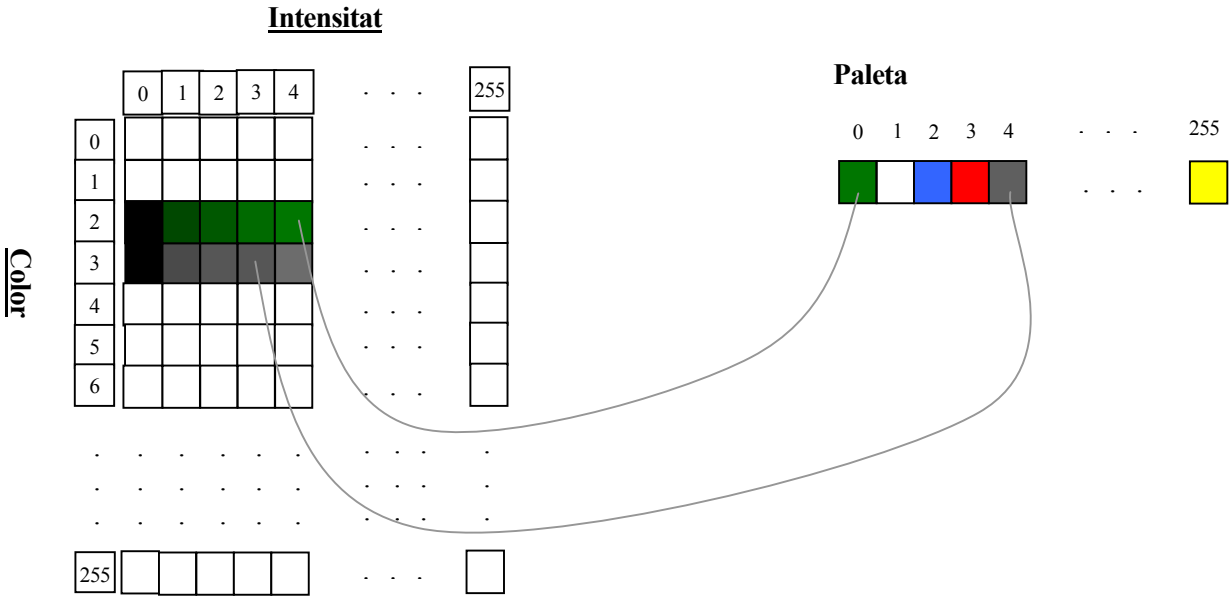


Figura a6.6 *Exemple d'una taula de ombres, cada valor d'aquesta taula és un l'índex al color de paleta adequat.*

I com aconseguirem crear aquesta taula ? Molt senzill. Bàsicament, el que cal fer **per cada color de paleta** i **per cada nivell intensitat** és, amb la rutina de il·luminació (llistat a6.5) calcular el valor RGB real segons el nivell de intensitat. Llavors, es feria la cerca del color de paleta més pròxim a aquest valor, amb la rutina *nearest color* (llistat a6.1).

```
Accio CrearTaulaOmbres(e Paleta e/s TaulaOmbres)

        Per i desde  0 a 256 fer { Per cada color de paleta... }
                Per j desde 0 a 256 fer { Per cada nivell intensitat... }

                        Color ← COLOR_AfegirIntensitat(Paleta[i],j)
                        TaulaOmbres[i][j]← NearestColor(Color)
                Fper
        FPer

FAccio
```

Llistat a6.6

A6.3 Creació d'una taula de transparència

En molts jocs a vegades haurem vist objectes amb transparència, tal que podem observar el fons a través d'ells (figura a6.6).

Figura a6.6 *Efecte de transparencia*

Per aconseguir tal efecte, cal fer una lectura a memòria de vídeo i amb la imatge del objecte, fer la barreja de colors abans de ser pintat a pantalla.

La barreja s'aconsegueix de la següent manera: Primer de tot, establim un interval de 256 nivells de transparència i, així, s'obtindrà el pes de barreja per l'objecte (*to*) i, per conseqüent, el pes de barreja del fons (*tf*) que serà el seu complementari.

$$to = \frac{transparencia}{256} \qquad \textit{Equació a6.2}$$

$$tf = 1.0 - to \qquad \textit{Equació a6.3}$$

transparència $\in [0, 256)$

Sumant les porcions per-component i multiplicades pel pes corresponent al nivell de transparència, aconseguirem la barreja d'ambdós colors.

```
Funcio COLOR_AfegirTransparencia(e ColorObjecte, ColorFons, transparencia)

        to ←transparencia div 256
        tf ← 1.0 – to

        Color_Amb_Transparencia.R ← ColorObjecte.R*to+ColorFons.R*tf
        Color_Amb_Transparencia.G ← ColorObjecte.G*to+ColorFons.G*tf
        Color_Amb_Transparencia.B ← ColorObjecte.B*to+ColorFons.B*tf

        retorna Color_Amb_Transparencia
FAccio
```

Llistat a6.7

Igual que abans, la funció del llistat a6.7 només serveix per sistemes de vídeo RGB. Per aconseguir adaptar-lo a sistemes de vídeo de 8 bpp, caldrà crear una altra taula bidimensional però ara, de transparències. L'algorisme per crear aquesta taula serà el mateix que el del llistat a6.6, però es substituirà la funció *COLOR_AfegirIntensitat()* per la funció *COLOR_AfegirTransparencia()*. Cal tenir en compte que, per cada nivell de transparència, s'haurà de crear una nova taula de 256x256.

Bibliografia

La bibliografia ha estat citada segons la notació MLA (Modern Language Association of America.: www.mla.org

- **Ensamblador ARM**

[ARM01] **ARM** *ARM7TDMI Technical Reference Manual* (2002) 2p. En línia. http://www.arm.com (26 Gener del 2005)

- **La Game Boy Advance**

[GT03] (*GBA TEK)*. En línia. Internet. 20 Agost 2003. Accessible a http://www.work.de/nocash/gbatek.htm

[TPPE01A] *Getting to know the GBA*. En línia. Internet. 30 Setembre 2003. Accessible a http://www.thepernproject.com/English/day_1.html

[TPPE01B] *Direct Memory Acces*. En línia. Internet. 30 Setembre 2003. Accessible a http://www.thepernproject.com/English/tutorial_5.html

[CVHS02] *CowBite Virtual Hardware Specifications*. En línia. Internet. 15 Agost 2003. Accessible a http://www.cs.rit.edu/~tjh8300/CowBite/CowBiteSpec.htm

- **Mode 7**

[TH01] *Mode 7 effects*. En línia. Internet. 20 Agost 2003. Accessible a http://www.taswegian.com/TwoHeaded/mode7.html

[TPP01] *Empezando a conocer la GBA*. En línia. Internet. 30 Setembre 2003. Accessible a http://www.taswegian.com/thepernproject/spanish/tutorial_1.html

[TONC04A] *Mode 7*. En línia. Internet. 12 Febrer 2004. Accessible a http://user.chem.tue.nl/jakvijn/tonc/mode7.htm

[TONC04B] *More mode 7 tricks*. En línia. Internet. 12 Febrer 2004. Accessible a http://user.chem.tue.nl/jakvijn/tonc/mode7ex.htm

- **Filtre MipMapping**

[AF06] *Run-Time MIP-Map Filtering* En línia. Internet. 12 Febrer 2006. Accessible a http://www.gamasutra.com/features/19981211/flavell_01.htm

[HCK97A] *Fundations* (1997). En línia. Internet. 17 Febrer 2006. Accessible a http://www.d6.com/users/checker/pdfs/gdmtex1.pdf

- **Coloització**

[SR06] *Cuantización de color y filtros*. En línia. Internet. 12 Febrer 2006. Accessible a http://www.sromero.org/articulos/gfx/graf14.html