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

Projecte

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

Capítol 1: Introducció

1.1 Descripció del projecte

Molts dels motors gràfics 3D els podem trobar a la majoria dels videojocs com són els coneguts *Quake, Half Life, Grand Thief Auto (GTA):Vic City*, *Doom3*,..., entre altres (figura 1.1). Actualment, aquests motors aconsegueixen generar uns gràfics molt realistes i amb una gran interacció davant l'usuari, gràcies a la acceleració 3D per Hardware que ofereixen les targetes gràfiques del mercat d'avui en dia.

Quake 3 (id Software)

Doom 3 (id Software)

Half-Life 2 (Sierra)

Grand Thief Auto Vice City

Figura 1.1 *Alguns dels jocs actuals. Totes les seves renderitzacions disposen de suport 3D, per això, aconsegueixen una qualitat d'imatge excel·lent i alta interacció de moviments.*

El present treball està orientat a dissenyar i implementar un motor 3D per a la consola *Game Boy Advance* (figura 1.3), una consola destinada a executar jocs de la casa Nintendo.

Figura 1.2 *Game Boy Advance*

Es important dir que, la GBA és la consola més popular, però la més lenta que podem trobar al mercat d'avui en dia i només disposa d'un hardware especialitzat de renderització en gràfics 2D. Però alguns dels jocs 3D que han sortit com *Doom*, *Duke Nukem Advance*, i altres, la GBA demostra tenir capacitat per fer un recorregut en móns 3D amb bons gràfics i interactius.

Doom

Duke Nukem Advance

Figura 1.3 *Alguns dels jocs 3D actuals per la consola GBA. Tot i que la consola manca d'acceleració 3D i poca potència hardware, els gràfics aconseguits en aquests jocs són molt bons.*

Realment, els jocs visualitzats a la figura 1.3 aconsegueixen interactivitat sobre la plataforma GBA gràcies a tècniques basades en els motors originaris dels jocs *Doom* o *Wolfenstein* (figura 1.4), entre els anys 1992-1994 on encara no hi havia hardware especialitzat en accelerar gràfics 3D. La sortida d'aquests motors va demostrar que es podia generar escenaris complexes interactius amb poca tecnologia hardware. Aquests motors van significar un salt endavant en tecnologia i gràfics però; més principalment per el motor *Doom*, que va esdevenir un dels jocs més aclamats i més influents a la dècada dels 90.

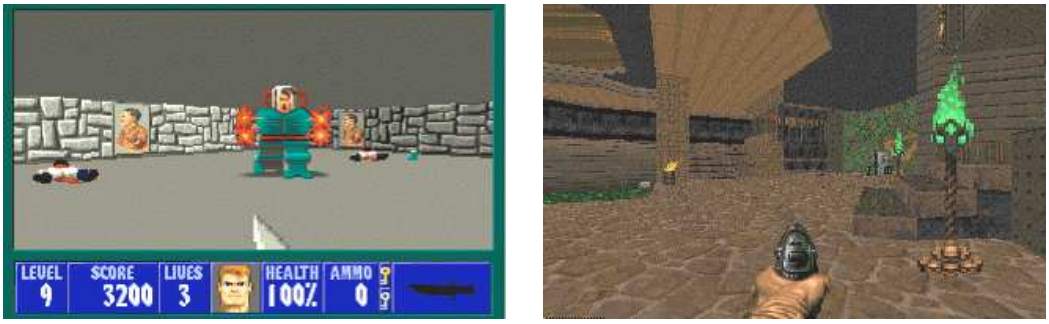


Figura 1.4 *Captures de pantalla dels jocs Wolfenstein 3D (esquerra) i Doom (dreta), ambdós realitzats per la companyia* ***idSoftware*** *entre els anys 1992-1994. La barra d'estat, pistola, columnes, entorxes, etc. són objectes afegits a la renderització d'escenaris.*

Així doncs, tenim un indici de quina arquitectura de motor caldrà dissenyar, que aplicarà un sistema de *renderització* semblant als motors gràfics representats a la figura 1.4. La tecnologia que utilitza el motor del *Wolfenstein* és molt simple, si més no es pot comprovar a la figura 1.4 (esquerra) que només es pinta la paret i sempre té la mateixa altura, apart la connexió entre parets sempre és ortogonal. És per això que el present treball s'ha decantat més per un disseny d'un motor similar al *Doom*, que pot generar móns més variats i interessants.

1.2 Objectius

L'objectiu d'aquest projecte ha estat construir un motor 3D per la consola *Game Boy Advance* (GBA) i que sigui capaç de generar o *renderitzar* interactivament (més de 16 Quadres per segon –FPS-) escenaris semblants als que s'ha observat a la figura 1.4 (dreta).

• Característiques del motor

El motor que s'ha desenvolupat en aquest projecte ha estat capaç de *renderitzar* parets amb angle no ortogonal/no-ortogonal entre ells, escales, finestres, portals,...etc. amb pintat de textura.

En resum, les característiques bàsiques de la base del nostre motor a desenvolupar són:

- Renderització de parets amb connexió ortogonal/ no-ortogonal.
- Renderització d'escenaris amb escales, finestres, pendents, etc.
- Pintat de sostre, terra i paisatges.
- Sistema de col·lisions jugador.

A més a més, en aquest treball s'estudiarà tècniques avançades de renderització 3D. Tal com,

- Renderització efectes de portal.
- Renderització de objectes amb animació.
- Il·luminació per-sector amb efecte de profunditat.
- Efecte de transparències.
- Renderització de món utilitzant una estructura de mapa semi-3D

1.3 Eines i aplicacions

Eines pel desenvolupament GBA

- **Software**

Les eines per software imprescindibles pel desenvolupament GBA en són bàsicament dues:

1 Una eina de compilació perquè passi el nostre codi font (C, PASCAL, etc.) a binari compatible per la plataforma GBA.

2 Una eina que emuli les aplicacions sobre la GBA, mitjançant un emulador.

Afortunadament, els **membres de la comunitat de desenvolupament de la GBA** han posat a la xarxa dos kits no oficials de lliure distribució que ajuden en el desenvolupament de la GBA: **DevKit** i **HAM**.

Dels dos Kits **s'utilitzarà el kit HAM**, el qual està disponible a la plataforma *Windows* i *Linux*. El kit porta integrat les dues eines esmentades anteriorment (el compilador i l'emulador) però, apart, també s'inclouen multitud d'eines que poden facilitar, en major part, les tasques del nostre projecte.

- **Hardware**

Apart de disposa del kit complet de desenvolupament HAM, en aquest treball s'ha adquirit la consola GBA i el gravador **EZ-Flash2** (figura 1.5) amb un cartutx gravable de fins a 256 Mbits (32 Mbytes) de capacitat. Per consegüent, s'ha pogut demostrar la funcionalitat i la *performance* (velocitat d'execució) del motor desenvolupat sobre la GBA.

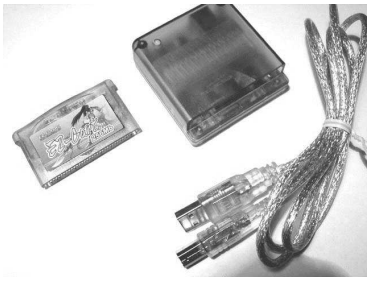


Figura 1.5 *Eina EZ-Flash II. En ella inclou el gravador de roms i un cartutx gravable per GBA.*

Llenguatges de programació

Els compiladors que porta integrats el kit HAM suporta els llenguatges C, C++ i ensamblador ARM.

Els llenguatges que normalment s'utilitzen en les aplicacions GBA (la majoria d'ells jocs) són el C i ensamblador, per aquest motiu aquests seran els llenguatge que farem servir en les nostres implementacions.

Eines pel desenvolupament del nostre motor

El kit de desenvolupament HAM està pensat per desenvolupar jocs 2D. Algunes funcions les utilitzarem per inicialitzar el mode gràfic, la paleta, etc. però la llibreria de HAM no inclou funcions per tractar problemes 3D. El nostre motor és construeix des del principi per tant, durant el desenvolupament del nostre motor caldrà d'introduir nous algorismes amb moltes proves d'execució abans de que funcionin a la perfecció.

A més a més, les eines en compilació i en depuració de HAM són bastant lentes. Si per cada modificació o depuració del nostre motor, ha de dependre de les eines que ens ofereix aquest kit, pot allargar bastant el nostre desenvolupament.

Sabent que és treballarà en el nostre llenguatge de programació serà C, la via més ràpida pel desenvolupament del nostre motor serà la utilització del potent entorn de programació MSDEV de Visual C++ on aquest inclou eines integrades potents com el propi debug, ràpid compilador de codi, eina *profile*, etc. L'entorn MSDEV s'utilitzarà principalment per construir el nostre motor en C. Més tard, es compatibilitzarà el nostre motor a la llibreria HAM.

Donat que utilitzarem l'aplicació MSDEV esta només per la plataforma Windows, farem servir la llibreria SDL, per manejar de manera senzilla els dispositius d'entrada i sortida com el sistema gràfic, el teclat, etc. Apart, una característica molt important d'aquesta llibreria es que és portable a les següents plataformes:

- Win32
- Linux
- BeOs
- MacOs
- BSD

1.4 Temporització del projecte

Han estat molts els camins recorreguts durant el transcurs d'aquest projecte, hagi estat per la falta d'experiència o per la desconeixença d'una causa. Apart, s'ha tingut de construir prototipus implicats al resultat del nostre motor final i altres implementacions per estudiar amb més detall una tècnica concreta que, per consegüent, han fet allargar més del compte aquest projecte. En concret, la duració total d'aquest projecte ha estat de tres anys.

El primera cosa que se'ns va ocorre va ser intentar trobar codi font de lliure distribució d'un motor 3D pel WEB, per adaptar-lo a la GBA. Es va triar un anomenat ***Baltimore 3D*****, un motor que podia renderitzar escenaris tipus *Doom* (molt adequat per els nostres propòsits)**. Malauradament, el motor *Baltimore* només podia *renderitzar* les parets de connexió ortogonal, per la qual cosa es va descartar immediatament. Es va estar buscant altres motors de lliure distribució però no es trobar cap que s'aproximés al proposat pel nostre projecte.

Es va descartar la cerca de motors i es va canviar de pla. Vam dissenyar el prototipus d'un motor ray-casting amb un mapa de cel·les, basat pel tutorial de [TM97] (Veure etapa 2 de la evolució de projecte: **Estudi d'un motor ray-casting**). Aquest va ser l'origen del nostre motor.

Un cop havent estudiat el funcionament ray-casting, es va modificar l'arquitectura del motor ray-casting perquè pogués renderitzar móns amb connexió de paret no-ortogonals, basat per una estructura de mapa en sectors i portals 2D, explicat per [JCC02].

Més tard, es va modificar el motor i mapa per poder renderitzar escenaris amb desnivells ortogonals. Així, el nostre motor va ser capaç de renderitzar un món amb escales, finestres, voreres, etc.

Llavors, és va estudiar el pintat de sostre, terra i paisatges. Per els paisatges, no va resultar cap problema la seva adaptació al nostre motor, ja que mantenia un procés molt similar per el pintat de parets. Pel pintat de plans (sostre i terra), és va estudiar i construir un prototipus "mode 7" per la GBA, una configuració hardware que permet simular un pla 3D de manera interactiva. El fet d'estudiar aquest mode, va ser degut a la necessitat de portar-lo al nostre motor, per i així, guanyar en eficiència a l'hora de *renderitzar* el sostre i terra. Malauradament, no va ser possible adaptar-lo al nostre motor, per consegüent, la *renderització* de sostre i terra va tenir que ser obligatòriament per Software.

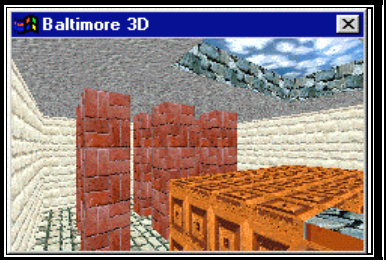
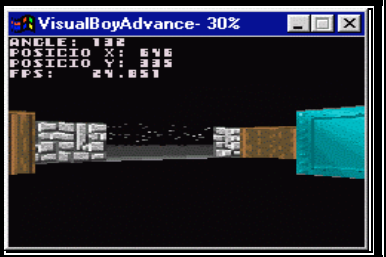
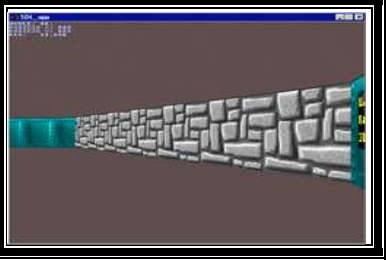
Tot seguit, es va estudiar altres efectes a introduir al nostre motor, tal com efectes de portal, il·luminació, renderització d'objectes, etc.

Per últim, es va estudiar la manera de renderitzar un món amb una estructura de mapa semi 3D.

Tot seguit es dona un resum i captura de pantalla (si escau) dels treballs realitzats en aquest projecte i per ordre cronològic.

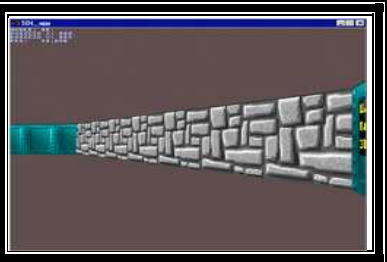
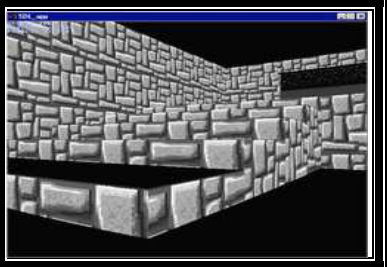
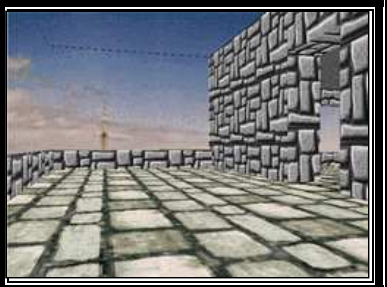
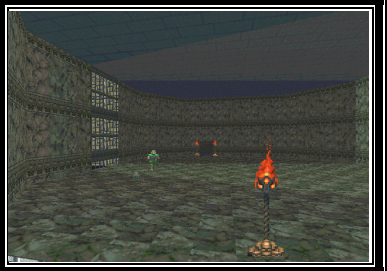
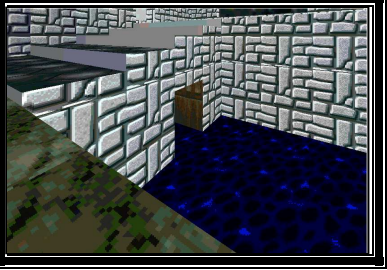
Evolució projecte

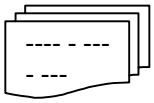
Les tasques que a continuació es descriuen representen, no s'han inclòs en aquesta documentació, totes aquestes es troben explicades (excepte "cerca de motors") a la documentació extra "evolució projecte". Aquestes tasques han donat lloc als fonaments d'aquest motor estudiat.

Data de realització	Descripció del treball	Captura de la imatge
12-06-03 a **30-09-03**	**Cerca de motors** Adaptació del motor Baltimore 3D a la GBA i cerca de altres motors de lliure distribució.	
30-09-03 a **01-12-03**	**Origen del motor** Disseny i implementació d'un prototipus ray-casting, basat en un mapa de cel·les.	
01-12-03 a **03-03-04**	**Base inicial del motor** Motor ray-casting adaptat a un mapa estructurat en sectors i portals.	

Projecte

Mostrem les tasques principals realitzats, en aquest projecte,

01-12-03 a **03-03-04**	**Optimització** Renderització de sector amb el llançament d'un sol raig.	
03-03-04 a **15-05-04**	**1ª Millora del motor** Modificació del motor i mapa per renderitzar un mapa amb desnivells.	
15-05-04 a **20-08-04**	**Prototipus** Estudi i implementació d'un prototipus "mode 7" a la GBA	
20-08-04 a **30-08-04**	**2ª Millora del motor** Pintat de sostre terra i paisatges	
30-08-04 a **15-12-04**	**3ª Millora del motor** Estudi d'altres efectes i filtres al renderitzat del motor: il·luminació, renderització de objectes, estructura de mapa semi 3D, etc	
15-01-05 a **20-03-05**	**4ª Millora del motor** Renderitzat de món utilitzant una estructura de mapa semi-3D.	

20-03-05 a **15-06-05**	**Optimització** Adaptació i estudi de algorismes d'optimització per aconseguir l'execució interactiva del motor a la GBA	
15-03-05 a **15-02-06**	Documentació	

1.5 Estructura de continguts

Els diferents capítols de que consta aquest dossier son els següents:

Capítol 2 Fonaments. En aquí, s'explicaren els conceptes, definicions, etc., imprescindibles abans d'entrar en el disseny del motor.

Capítol 3 Disseny del motor. Explicació del disseny base del nostre motor gràfic. També, l'estudi de efectes de renderització en base l'arquitectura del motor.

Capítol 4 Excepcions. Excepcions de càlculs, casos particulars de problemes, etc, que podem trobar durant la renderització del escenari.

Capítol 5 Optimitzacions Estudi dels mètodes i tècniques òptimes en càlculs i algorísmica per fer més eficient el processat del render. Més tard, configurar el codi sota la plataforma GBA perquè s'executi en òptimes condicions. Tot això per un sol objectiu: aconseguir *renderitzar* interactivament (+ de 16 fps) els escenaris del nostre motor en aquesta màquina.

Capítol 6 Resultats, conclusions i treballs futurs

ANNEXOS

Els annexos que complementaran la informació d'aquest projecte, estan inclosos a la documentació "annexos" i són:

Annex 0 Ensamblador ARM. Descripció detallada de les instruccions ensamblador ARM de la GBA. També directives i exemples de comunicació de funcions ensamblador des de codi C, basat al compilador que utilitzarem.

Annex 1 La Game Boy Advance. Descripció de la GBA, operacions especials i d'altres característiques de la seva arquitectura.

Annex 2 Eines del desenvolupament. Entre elles, *El kit de desenvolupament HAM* on s'explicarà amb detall de les eines i funcions de la llibreria que incorpora aquest kit.

Annex 3 El "mode 7" a la GBA. Es documentarà el disseny i implementació necessària per portar la tècnica del "mode 7" a la GBA.

Annex 4 Aritmètica entera. S'explicaran algorismes/tècniques de aritmètica entera.

Annex 5 El filtre Mip-Maping. Tècnica per arreglar problemes d'aliasing observats en els objectes mapejats amb textura.

Annex 6 Tècniques de paleta. Es descriuen algunes tècniques de paleta, per modes de 8 bpps.

Evolució del projecte

Mostrem les etapes contemplades en el manual històric el qual ha estat l'estudi base d'aquest projecte:

Capítol 1 Estudi d'un motor ray-casting. Estudi d'un motor ray-casting per-columna amb una estructura de mapa de cel·les quadrades.

Capítol 2 Subdivisió del mapa en sectors i portals. Respecte el motor anteriorment estudiat, en aquesta etapa s'estudiarà l'adaptació d'un mapa basat en sectors i portals.

Capítol 2: Fonaments

El present capítol té com objectiu explicar algorismes, conceptes i definicions fonamentals al disseny del nostre motor.

2.1 Conceptes bàsics de gràfics

2.1.1 Definicions

La present secció pretén especificar el sentit del sistema de coordenades i les orientacions així com explicacions o conceptes bàsics relacionats amb la programació de gràfics 3D, que es realitzarà durant el disseny del motor.

2.1.1.1 Orientacions i sistema de coordenades

El sentit dels eixos del sistema de coordenades que utilitzarem pel món 3D serà el que es mostra a continuació. Cal dir que quan estem parlant d'un punt en espai de món, ho especificarem amb superíndex "3D".

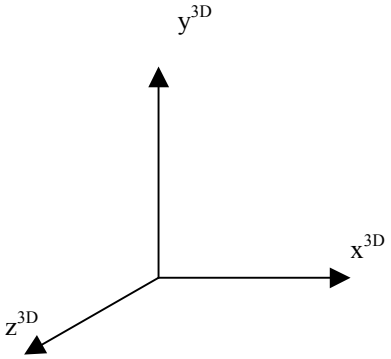


Figura 2.1

Les orientacions que farem servir estaran relacionades a l'espai XZ i seran les següents,

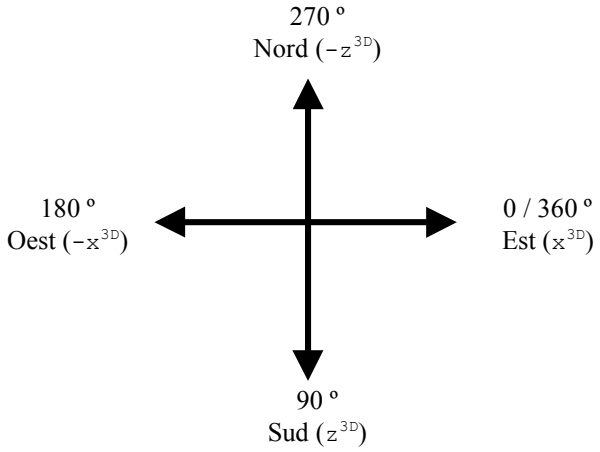


Figura 2.2

2.1.1.2 La pantalla

La pantalla es dispositiu on es mostrarà visualment els escenaris del món 3D renderitzats. La pantalla mostra el color dels píxels que es troben emmagatzemats al *frame-buffer*. El *frame-buffer* és un bufer lineal ubicat a VRAM d'igual dimensió del que permet la resolució de pantalla (x_res*y_res, segons figura 2.3). El *frame-buffer* es representa com un taula 2D on a cada cel·la conté la informació d'un píxel, amb inici del primer píxel (0,0) a l'extrem superior esquerra. La informació de píxel no es res més que el valor de color que es mostrarà per pantalla, ja sigui fent referència a un color de paleta (Sistemes gràfics de 8 bit-per-pixel -8 bpp-) o directament un valor de color RGB codificat (Sistemes gràfics de 12, 15, 16, 24 o 32 bpp).

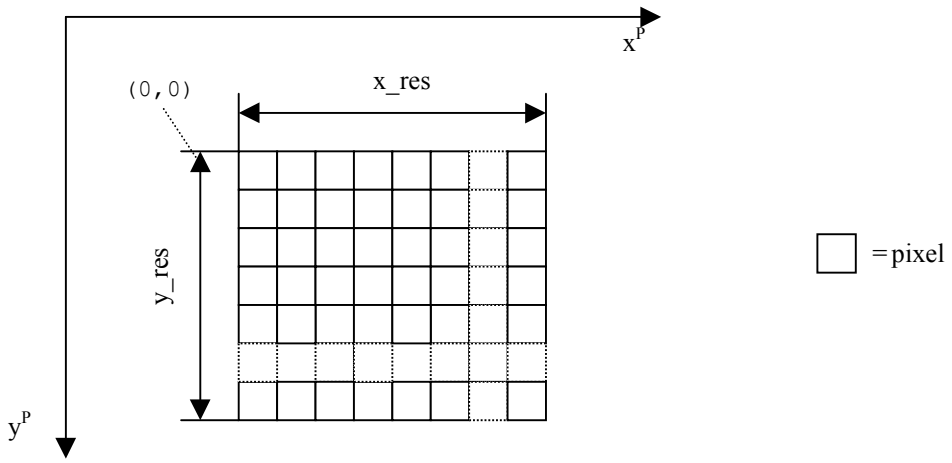


Figura 2.3

2.1.1.3 La textura

La textura és una imatge que s'utilitza, normalment, com element de pintat. La textura està formada per **texels** on cadascun d'ells explica el valor de color. La seva densitat de color pot ser variada (des de 8 a 32 bpps), però és molt recomanable que tingui la mateixa densitat que la del sistema gràfic que es treballi perquè del contrari s'haurà de perdre temps en la codificació i descodificació del color. Igual que el *frame-buffer*, la textura també es representa com una taula 2D i cada casella hi conté la informació de color i el sentit de coordenades és el mateix. A diferencia, els texels s'indexen en coordenades UV.

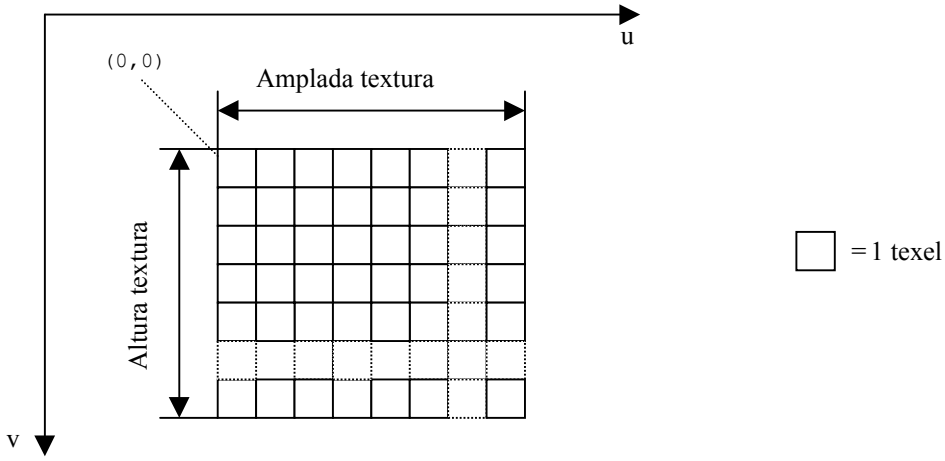


Figura 2.4

En algorismes de pintat dels videojocs, normalment utilitzen dimensions de textura de 2^n amb $n \in \mathbb{N}$ per qüestions d'optimització. Una textura dins un PC, normalment, sempre es carrega de disc. Per contra, la GameBoy Advance (GBA) no disposa de sistema de fitxers pel que una textura en format gràfic ha de ser convertida en un format llegible. A l'annex 2 secció a2.1.3, explica com transformar gràfic per ser llegit posteriorment a la GBA.

2.1.1.4 L'ull

L'ull és un punt virtual definit a l'espai 3D que representa l'element camera on, a partir de la seva posició i orientació, generem el món 3D al pla-projecció (pantalla). En la figura 2.5 s'il·lustra una escena amb objectes tridimensionals i l'ull situat en el vèrtex superior d'una piràmide (on les quatre línies convergeixen intersectant els extrems del pla-projecció), aquest tipus de punt de vista és coneix com a ***Frustum***. Únicament els objectes dins el volum seran dibuixats per pantalla. La figura de la dreta ens mostra els objectes que és veurien des del jugador. Al pròxim punt (espai d'ull) veurem que cal fer una transformada de vista per portar els objectes i punts del seu lloc de món al l'espai d'ull.

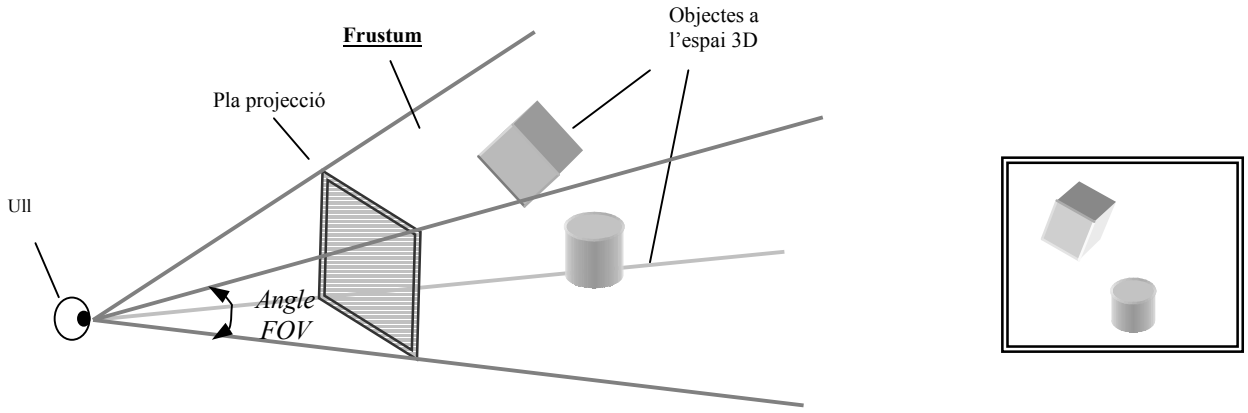


Figura 2.5

El volum del *Frustum* ve determinat per l'angle del **camp de visió** (o en el denominat *Field Of View* -**FOV**-), igual per els 4 costats del *Frustum*.

- **Espai d'ull**

Cal mencionar que únicament els objectes que son visibles per l'*ull* son renderitzats. L'*ull* te una posició en coordenades de mon i una direcció, que és emprada per posicionar-la i moure-la. Amb la finalitat de simplificar la projecció i el retallat, l'*ull* i tots els objectes son transformats per la **transformada de la vista**, d'aquesta manera aconseguim situar l'*ull* al seu origen. Com veurem al següent capítol, l'*ull* sempre recorrerà un espai 2D, per tant la transformada de vista també serà 2D. Concretament es transformarà la vista XZ del món a l'espai d'ull, on els eixos de món transformats ($x^{3D'}$,$z^{3D'}$) quedaran com reflexa la figura 2.6.

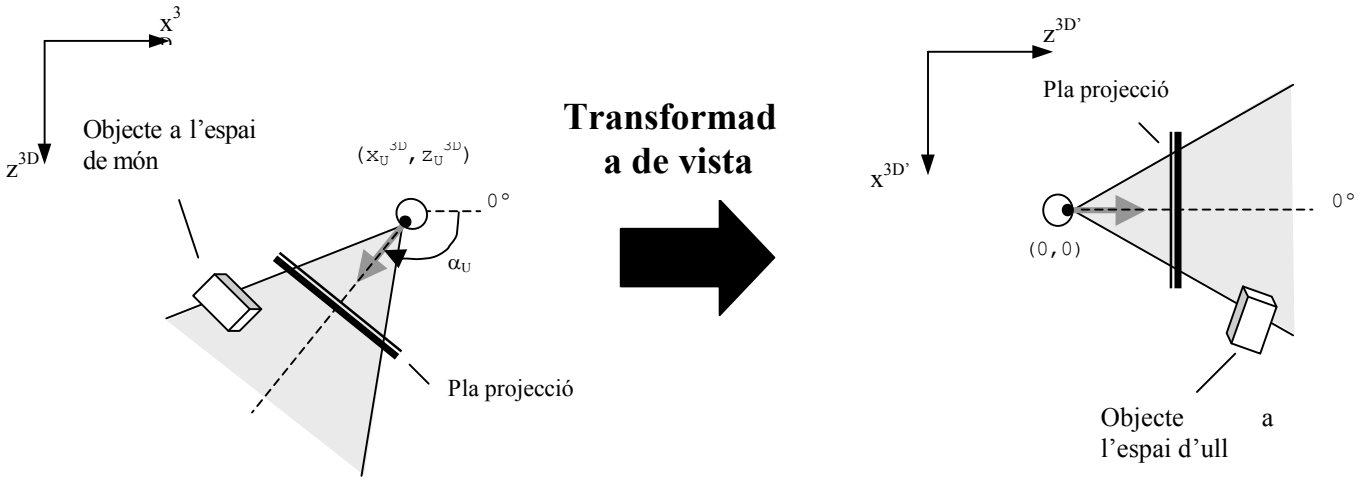


Figura 2.6

- **El pla-projecció**

El pla-projecció (o la pantalla) és troba sempre perpendicular a la direcció de l'ull i a una posició relativa a les coordenades de l'ull. Per convenció en gràfics, l'ull **sempre** es situa al centre de projecció del pla (x_res/2, y_res/2). L'ull es troba a una distància *d* en Z respecte les coordenades de la pla-projecció, tal com és mostra a la figura 2.7.

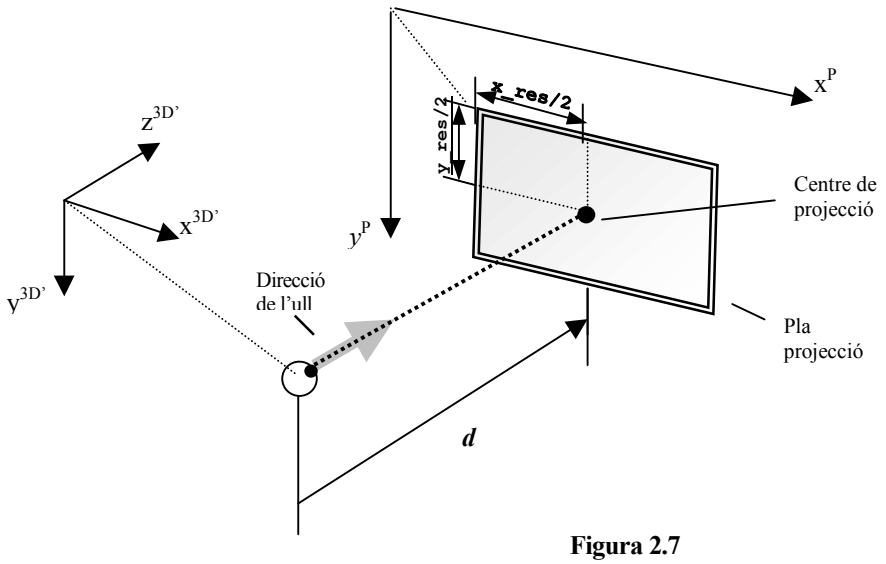


Figura 2.7

- **Obtenció de la distància ull-pla (*d*)**

Per trobar *d*, cal que el pla-projecció estigui situat en el seu **espai**. i segons resolució horitzontal de pantalla (*x_res*) o vertical (*y_res*) i el FOV establert, podem trobar la distància del pla. Fent-ho segons la resolució horitzontal queda un aspecte eixamplat i vertical queda allargat. Nosaltres trobarem *d*, segons la resolució horitzontal, ja que dona un resultat visual bastant millor que no pas verticalment.

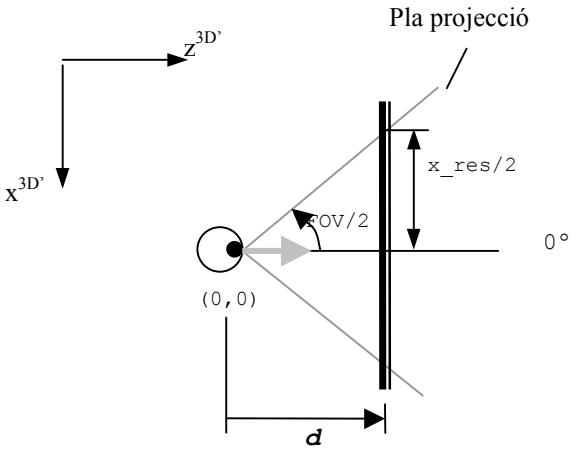


Figura 2.8

Fent referència a la figura 2.8, trobem el valor *d* aplicant trigonometria.

$$\tan(FOV/2) = \frac{x\_res/2}{d} \Rightarrow d = \frac{x\_res/2}{\tan(FOV/2)} \Rightarrow \boxed{d = \frac{x\_res}{2 \cdot \tan(FOV/2)}}$$

Equació 2.1

2.1.2 Transformacions

Moltes vegades s'haurà de rotar punts a l'espai 3D, per això donem les matrius de rotació en els tres eixos (Eix X, Eix Y i Eix Z) d'acord amb les orientacions definides a la secció 2.1.

- **Rotació d'un punt en l'eix X**

$$\mathrm{Rot}(\mathrm{EixX}, \theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) \\ 0 & -\sin(\theta) & \cos(\theta) \end{pmatrix}$$

- **Rotació d'un punt en l'eix Y**

$$\mathrm{Rot}(\mathrm{EixY}, \alpha) = \begin{pmatrix} \cos(\alpha) & 0 & -\sin(\alpha) \\ 0 & 1 & 0 \\ \sin(\alpha) & 0 & \cos(\alpha) \end{pmatrix}$$

- **Rotació d'un punt en l'eix Z**

$$\mathrm{Rot}(\mathrm{EixZ}, \psi) = \begin{pmatrix} \cos(\psi) & \sin(\psi) & 0 \\ -\sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

2.1.3 Projecció de punts de l'espai de món a l'espai de pantalla

Descripció del problema

Donat un punt qualsevol a l'espai de món, el volem saber quina coordenada de pantalla es projecta. Per projectar un punt de món calen dos passos:

1. Portar el punt de món a l'espai d'ull.

2. Un cop portat el punt a l'espai d'ull, es podrà calcular les coordenada XY de projecció. Si alguna de les coordenades projectades surten fora dels intervals de resolució de pantalla, no el considerarem.

1. Portant el punt de mon a l'espai de ull

Per portar un punt a l'espai d'*ull*, cal aplicar una transformada de vista. Recordem que a la secció 2.1.1.4 (a la apartat d'espai de d'*ull*), s'ha restringit que l'*ull* es mourà lliurement per coordenades XZ segons l'angle de direcció que incorpora. Per això, la transformada de vista sobre un punt a l'espai de món consisteix en aplicar-li una rotació $-\alpha_U$ respecte l'eix Y d'*ull* tal com mostra la següent figura.

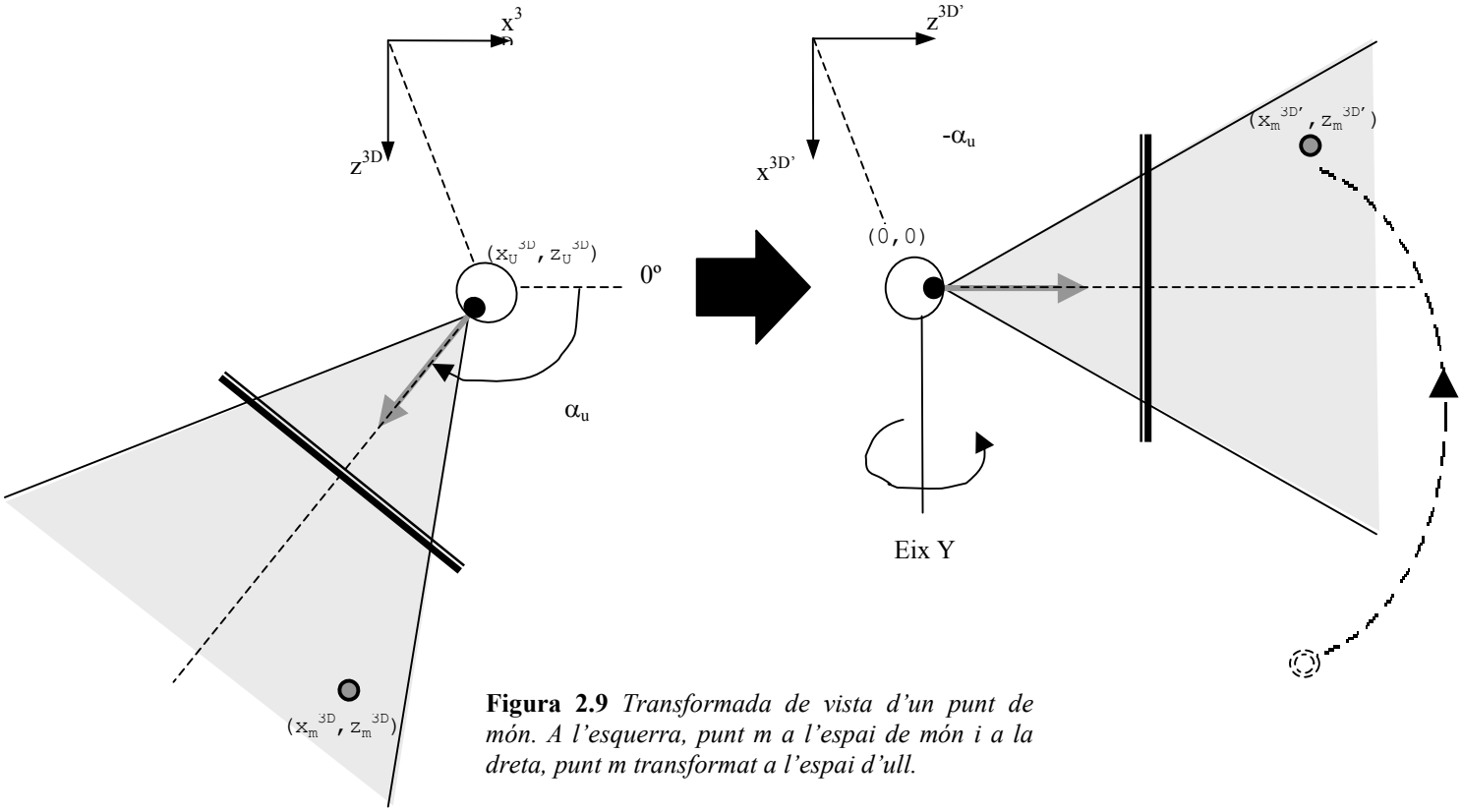


Figura 2.9 *Transformada de vista d'un punt de món. A l'esquerra, punt m a l'espai de món i a la dreta, punt m transformat a l'espai d'ull.*

La matriu de rotació -α en 2D i en l'eix Y, deduïda de la matriu `Rot(EixY, α)` (apartat 2.1.2), és la següent:

$$\texttt{Rot(EixY,}-\alpha\texttt{)} = \begin{pmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{pmatrix}$$

Com que la rotació s'efectua entorn l'eix Y de l'ull cal traslladar x_m^{3D} i z_m^{3D} al seu origen.

$$x\_diff \leftarrow x_m^{3D} - x_u^{3D}$$
$$z\_diff \leftarrow z_m^{3D} - z_u^{3D}$$

Llavors els valors $x_m^{3D'}$ i $z_m^{3D'}$ s'obtenen de la següent manera,

$$\begin{pmatrix} x_m^{3D'} \\ z_m^{3D'} \end{pmatrix} = \texttt{Rot(EixY, } \alpha_u\texttt{)} \cdot \begin{pmatrix} x\_diff \\ z\_diff \end{pmatrix} = \begin{pmatrix} \cos(\alpha_u) & \sin(\alpha_u) \\ -\sin(\alpha_u) & \cos(\alpha_u) \end{pmatrix} \cdot \begin{pmatrix} x\_diff \\ z\_diff \end{pmatrix}$$

$$x_m^{3D'} \leftarrow \cos(\alpha_u)\cdot x\_diff + \sin(\alpha_u)\cdot z\_diff$$
$$z_m^{3D'} \leftarrow -\sin(\alpha_u)\cdot x\_diff + \cos(\alpha_u)\cdot z\_diff$$

2. Projecció del punt a la pantalla

Fent referència al punt de mon en l'espai d'ull ($x_m^{3D'}$, $z_m^{3D'}$), obtingut a l'apartat anterior, podem trobar la formula de projecció en X emprant similitud de triangles.

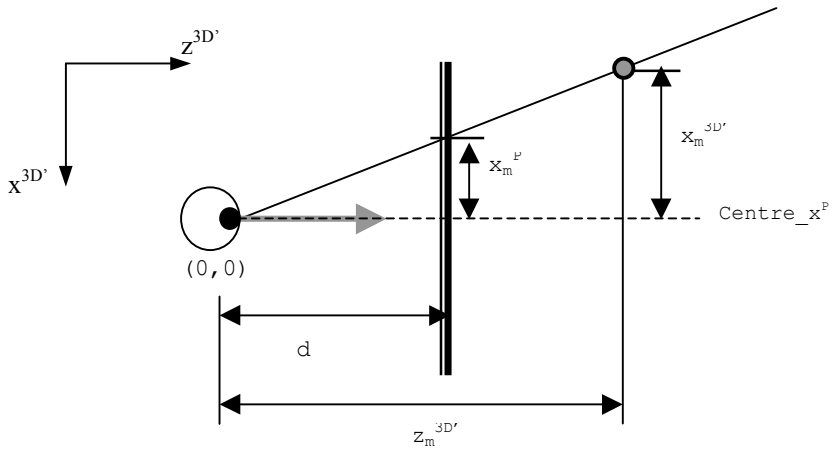


Figura 2.10

On,

$$Centre\_X^{P} = x\_res/2$$

Per similitud de triangles tenim,

$$\frac{x_m^{P}}{d} = \frac{x_m^{3D'}}{z_m^{3D'}}$$

Finalment, podem trobar x_m^P si l'aïllem, però aquest s'ha obtingut amb la posició de l'ull a l'origen del pla-projecció, o sigui, en el píxel (0,0). Com és va esmentar a la secció 2.1.1.4, cal que l'*ull* es posicioni al centre XY per tenir la correcta projecció X, per tant s'haurà de sumar el centre del pla-projecció.

$$x_m^{P} = Centre\_x^{P} + d \cdot \frac{x_m^{3D'}}{z_m^{3D'}}$$

Equació 2.2

Per trobar y_m^P s'ha de saber l'altura del punt en el món (y_m^{3D}), per portar-lo a l'espai l'*ull* ($y_m^{3D'}$). Per fer-ho només cal saber la distància relativa.

$$y_m^{3D'} = y_m^{3D} - y_u^{3D}$$

Segons la figura 2.11

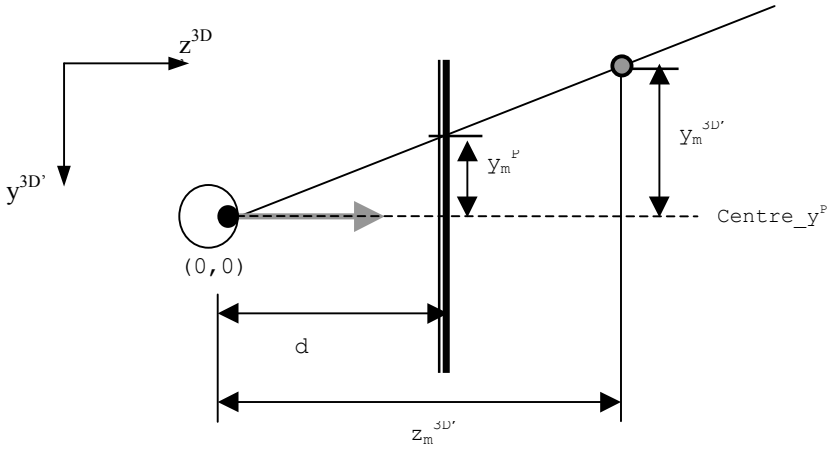


Figura 2.11

Trobem la projecció Ym^P emprant similitud de triangles, tenint en compte que cal afegir l'offset de l'ull al centre Y.

$$y_m^{P} = Centre\_y^{P} + d \cdot \frac{y_m^{3D'}}{z_m^{3D'}}$$

Equació 2.3

Cal adonar-se que quan el límit de cada projecció, quan $Z_m{}^{3D'}$ tendeix a infinit, els valors de projecció s'anulen al centre de la seva corresponent resolució. Per això, moltes vegades *Centre_y*[p] també s'anomena l'horitzó de pantalla.

2.1.4 Raigs

A la secció 1.6.5 (capítol 1 de la documentació "evolució del projecte"), es va comprovar que els angles de raig eren erronis, però que amb un FOV de 60º es dissimulava el seu error. En aquesta secció veurem la manera de llançar correctament els raigs. Cada raig **intersecarà** la columna del pla-projecció per on sigui llançat (figura 2.12). Per aspectes del disseny, els raigs intersectaran al centre de cada columna.

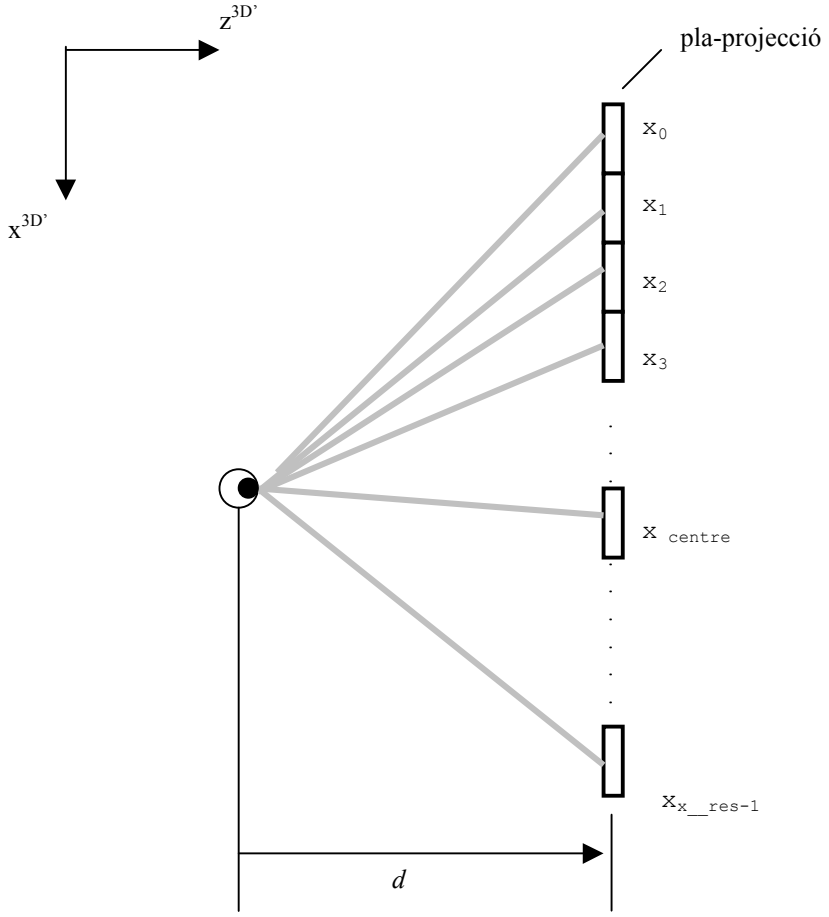


Figura 2.12

Moltes vegades interessarà saber el pendent de cada raig per executar el càlcul d'una intersecció, per exemple. Segons la figura 2.15, es pot deduir la fórmula per obtenir el pendent de qualsevol raig llançat per la columna *n*,

$$Pendent(x_n) = \frac{(x_n + 0.5) - x_{centre}}{d}$$ *Equació 2.4*

L'equació 2.4 la podem modificar a la següent

$$Pendent\ (x_n) = \frac{1}{d}(x_n + 0.5) - \frac{x_{centre}}{d} \qquad \textit{Equació 2.5}$$

Com que l'equació 2.5 té la forma d'una equació lineal (f(x) = Ax + B) i com que d i x_{centre} són constants, és veu que **el pendent-raig es lineal**. També, podem representar l'equació anterior de la següent manera:

$$\boxed{Pendent(x_n) = Pendent(x_{n-1}) + \Delta Pendent}$$

On:

$$\Delta Pendent = \frac{1}{d}$$

Demostració:

$$\frac{1}{d}(x_n + 0.5) - \frac{x_{centre}}{d} = Pendent(x_{n-1}) + \frac{1}{d}$$

$$\frac{1}{d}(x_n + 0.5) - \frac{x_{centre}}{d} = \frac{1}{d}(x_n - 1 + 0.5) - \frac{x_{centre}}{d} + \frac{1}{d}$$

$$\frac{1}{d}(x_n + 0.5) - \frac{x_{centre}}{d} = \frac{1}{d}(x_n + 0.5) - \cancel{\frac{1}{d}} - \frac{x_{centre}}{d} + \cancel{\frac{1}{d}}$$

$$\frac{1}{d}(x_n + 0.5) - \frac{x_{centre}}{d} = \frac{1}{d}(x_n + 0.5) - \frac{x_{centre}}{d}$$

Un dels avantatges que comporta la linealitat de l'equació 2.5 és que els podem pre-calcular en ves de calcular-los a temps d'execució.

2.1.5 Interpolació amb correcció de perspectiva

Introducció

Durant la rasterització de primitives de gràfics linears, tal com línies i polígons, la interpolació linear dels atributs de vèrtexs en l'espai 3D sobre la recta de l'espai de pantalla, generalment no produeix resultats correctes en perspectiva.

Per exemple, suposem que l'*ull* es troba davant d'un segment de línia on s'ha volgut interpolar linealment a l'espai d'ull la distància com atribut, tal com mostra la figura 2.13.

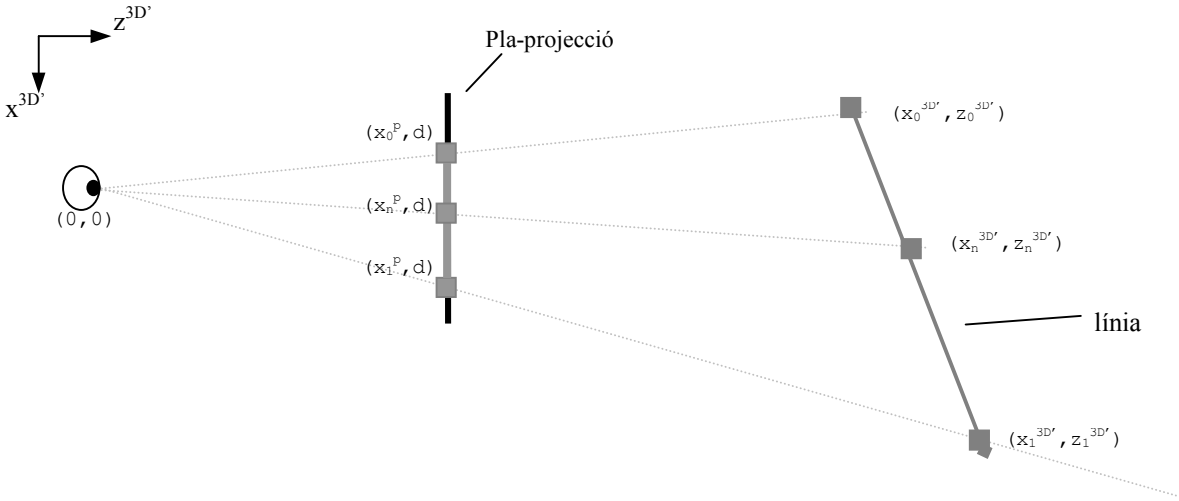


Figura 2.13

Fent referència a la figura 2.13, per interpolació lineal, podem obtenir qualsevol distància mitjançant la següent formula,

$$z_n^{3D'} = z_0^{3D'} + n\frac{(z_1^{3D'} - z_0^{3D'})}{(x_1^{P} - x_0^{P})} \qquad \textit{Equació 2.6}$$

Si el segment de línia representés una paret, utilitzéssim un renderitzador d'escenaris **ray-casting** (se'n parlarà amb més detall al següent capítol) i es fes servir les distàncies obtingudes segons l'equació 2.6 per projectar la paret, els resultats no tindrien un aspecte correcte de perspectiva, tal com mostra següent imatge.

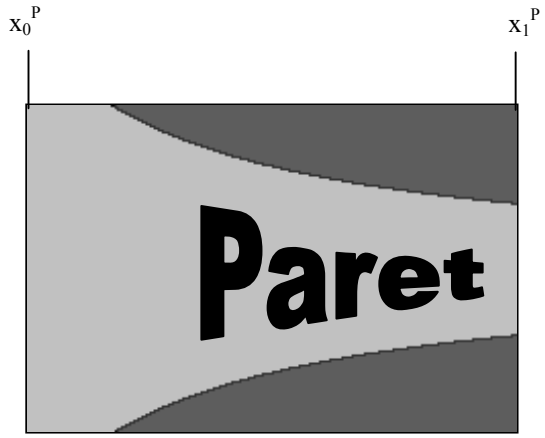


Figura 2.14 *Projecció de paret segons les distàncies obtingudes a l'equació 2.6.*

En aquest exemple, el que s'ha volgut demostrar és que **la variació linear del atribut *distància* (valor-*z*) NO es pot traslladar a la variació linear similar del pla-projecció.**

Així doncs, **qualsevol atribut a l'espai 3D tal com colors, coordenades de textura, etc que estan associats als vèrtexs a la línia de la figura 2.16 TAMPOC tindrien un comportament correcte en perspectiva si els interpolem linealment al seu espai.**

[LOW02] explica la manera d'obtenir els resultats correctes de perspectiva aplicant una interpolació lineal al pla projecció. La interpolació en perspectiva es pot aconseguir mitjançant la interpolació de funcions d'atributs en vés d'interpolar directament els valors d'atributs. Cada resultat interpolat serà transformat per una altra funció (la funció inversa) per aconseguir el valor de atribut final al punt desitjat dins al pla projecció. Veurem que aquestes funcions fan us dels valors-*z* a cada vèrtex.

Interpolació dels valors-*z*

Els valors-*z* es poden tractar com un atribut quals valors varien linearment a través d'una primitiva linear 3D. Com hem pogut observar a la figura 2.17, la interpolació lineal directa del valors-*z* sobre el seu espai no produeixen una perspectiva correcte. Però, seguidament veurem que podem interpolar linearment els valors-*z* inversos per portar a cap els valors correctes de perspectiva.

Semblant a la figura 2.13, a la figura 2.15 podem observar una línia 2D al sistema de coordenades de l'ull que es dins al pla-projecció 1D (entre x_0^P a x_1^P). A la figura s'explica els símbols que farem servir en les formules de derivació. *s* és el paràmetre interpolador en el pla-projecció, i *t* és el paràmetre interpolador de la primitiva linear 3D. Els valors *I* corresponen els valors de atribut que van associats als vèrtexs tal com una coordenada de textura, color, etc i que van lligats a la interpolació de perspectiva, però d'això ja se'n parlarà al punt 3 d'aquesta secció.
El nostre objectiu és derivar la formula d'una interpolació correcta, en el pla-projecció, els valors-*z*. La mateixa derivació pot ser directament aplicada al cas d'una primitiva 3D linear projectada al pla-projecció 2D.

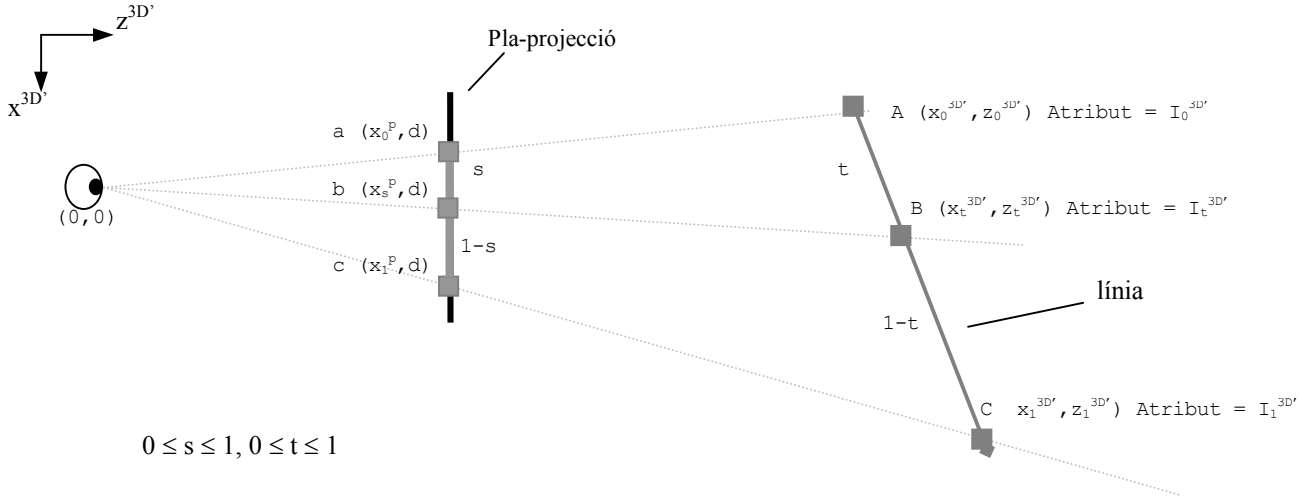


Figura 2.15 *A, B i C són punts de la primitiva amb valors d'atributs $I_0^{3D'}$, $I_1^{3D'}$ i $I_t^{3D'}$ respectivament, i el seu resultat al pla-projecció són a,b i c respectivament. s i t són paràmetres utilitzats per la interpolació linear.*

Referint-se a la figura 2.15, per similitud de triangles aconseguim les següents formules

$$\frac{x_0^P}{d} = \frac{x_0^{3D'}}{z_0^{3D'}} \Rightarrow x_0^{3D'} = \frac{x_0^P \cdot z_0^{3D'}}{d} \qquad \textit{Equació 2.7}$$

$$\frac{x_1^P}{d} = \frac{x_1^{3D'}}{z_0^{3D'}} \Rightarrow x_1^{3D'} = \frac{x_1^P \cdot z_1^{3D'}}{d} \qquad \textit{Equació 2.8}$$

$$\frac{x_s^P}{d} = \frac{x_t^{3D'}}{z_t^{3D'}} \Rightarrow z_t^{3D'} = \frac{d \cdot x_t^{3D'}}{x_s^P} \qquad \textit{Equació 2.9}$$

Per interpolació linear al pla-projecció, aconseguim

$$x_s^P = x_0^P + s\left(x_1^P - x_0^P\right) \qquad \mathbf{s \in [0..1]} \qquad \textit{Equació 2.10}$$

Per interpolació lineal a través de la primitiva a l'espai de ull, aconseguim

$$x_t^{3D'} = x_0^{3D'} + t\left(x_1^{3D'} - x_0^{3D'}\right) \qquad \textit{Equació 2.11}$$

$$z_t^{3D'} = z_0^{3D'} + t\left(z_1^{3D'} - z_0^{3D'}\right) \qquad \textit{Equació 2.12}$$

Substituint l'equació 2.11 i l'equació 2.10 dins l'equació 2.9,

$$z_t^{3D'} = \frac{d\left(x_0^{3D'} + t\left(x_1^{3D'} - x_0^{3D'}\right)\right)}{x_0^P + s\left(x_1^P - x_0^P\right)} \qquad \mathbf{t \in [0..1]} \qquad \textit{Equació 2.13}$$

Llavors, substituint l'equació 2.7 i l'equació 2.8 dins l'equació 2.13,

$$z_t^{3D'} = \frac{d\left(\frac{x_0^P \cdot z_0^{3D'}}{d} + t\left(\frac{x_1^P z_1^{3D'}}{d} - \frac{x_0^P z_0^{3D'}}{d}\right)\right)}{x_0^P + s\left(x_1^P - x_0^P\right)} = \qquad \textit{Equació 2.14}$$

$$\frac{x_0^P \cdot z_0^{3D'} + t\left(x_1^P z_1^{3D'} - x_0^P z_0^{3D'}\right)}{x_0^P + s\left(x_1^P - x_0^P\right)}$$

Substituint l'equació 2.12 dins l'equació 2.14, aconseguim

$$z_0^{3D'} + t\left(z_1^{3D'} - z_0^{3D'}\right) = \frac{x_0^P \cdot z_0^{3D'} + t\left(x_1^P z_1^{3D'} - x_0^P z_0^{3D'}\right)}{x_0^P + s\left(x_1^P - x_0^P\right)} \qquad \textit{Equació 2.15}$$

L'equació 2.15 pot ser simplificada com

$$t = \frac{s z_0^{3D'}}{s z_0^{3D'} + (1 - s) z_1^{3D'}} \qquad \textit{Equació 2.16}$$

substituint l'equació 2.16 dins l'equació 2.12, aconseguim

$$z_t^{3D'} = z_0^{3D'} + \frac{s z_0^{3D'}}{s z_0^{3D'} + (1 - s) z_1^{3D'}} \left(z_1^{3D'} - z_0^{3D'}\right) \qquad \textit{Equació 2.17}$$

l'equació 2.17 pot ser simplificada com

$$\frac{1}{z_t^{3D'}} = \frac{1}{\frac{1}{z_0^{3D'}} + s\left(\frac{1}{z_1^{3D'}} - \frac{1}{z_0^{3D'}}\right)}$$

Equació 2.18

L'equació (equació 2.18) explica que els valors-*z* poden ser correctament derivats per una interpolació lineal entre $1/z_0^{3D'}$ i $1/z_1^{3D'}$, i llavors computar la inversa del valor interpolat per aconseguir el valor-*z* correcte en perspectiva.

Interpolació dels valors de atribut

En aquí, volem derivar la formula per interpolar correctament en perspectiva, al pla-projecció, els valors d'un atribut *I*. Referint-se un altre cop a la figura 2.15, per interpolació lineal del valor d'atributs a través de la primitiva a l'espai d'ull aconseguim

$$I_t^{3D'} = I_0^{3D'} + t\left(I_1^{3D'} - I_0^{3D'}\right)$$

Equació 2.19

Substituint l'equació 2.16 dins l'equació 2.19, aconseguim

$$I_t^{3D'} = I_0^{3D'} + \frac{sz_0^{3D'}}{sz_0^{3D'} + (1-s)z_1^{3D'}}\left(I_1^{3D'} - I_0^{3D'}\right)$$

Equació 2.20

que pot ser arreglat com

$$I_t^{3D'} = \frac{\left(\frac{I_0^{3D'}}{z_0^{3D'}} + s\left(\frac{I_1^{3D'}}{z_1^{3D'}} - \frac{I_0^{3D'}}{z_0^{3D'}}\right)\right)}{\left(\frac{1}{z_0^{3D'}} + s\left(\frac{1}{z_1^{3D'}} - \frac{1}{z_0^{3D'}}\right)\right)}$$

Equació 2.21

Des de l'equació 2.18, podem veure que el denominador en l'equació 2.21 és justament $1/Z_t$. Per això,

$$I_t^{3D'} = \frac{\left(\frac{I_0^{3D'}}{z_0^{3D'}} + s\left(\frac{I_1^{3D'}}{z_1^{3D'}} - \frac{I_0^{3D'}}{z_0^{3D'}}\right)\right)}{\frac{1}{z_t^{3D'}}} \qquad \textit{Equació 2.22}$$

L'equació 2.22 explica que el valors de atribut poden ser correctament derivats per una interpolació linear entre $\mathbf{I_0^{3D'}/Z_0^{3D'}}$ i $\mathbf{I_1^{3D'}/Z_1^{3D'}}$, i llavors dividir el resultat interpolat per $1/Z_t^{3D'}$, el qual ell mateix pot ser derivat per una interpolació linear al pla-projecció com s'ha vist a l'equació 2.18.

Algorisme interpolador amb correcció de perspectiva

Tot havent demostrat amb les equacions anteriors la linealitat dels valors-*z* i valors-*I* al pla projecció, donem l'algorisme base per interpolar qualsevol atribut *I* amb correcció de perspectiva sobre una línia a l'espai d'ull.

```
Algorisme Interpolador_Amb_Correccio_Perspectiva
    { Obtenció de les coordenades 3D (x0^3D',z0^3D'), (x1^3D',z1^3D') i els valors d'atributs (I0^3D',I1^3D')}

    {...}

    { Valors de projecció x0^p - x1^p }

    x0^p ← Centre_X+d·(x0^3D'/z0^3D')
    x1^p ← Centre_X+d·(x1^3D'/z1^3D')

    { funció transformació lineal dels valors-z a l'espai de pantalla (inversa dels valors-z) }

    z0^p ← 1/z0^3D'
    z1^p ← 1/z1^3D'

    { funció transformació lineal dels atributs-I  a l'espai de pantalla (divisió I entre el corresponent valor-z)}

    I0^p ← I0^3D'/z0^3D'
    I1^p ← I1^3D'/z1^3D'

    { derivades de la interpolació lineal del valors-Z i valors-I}

    inc_dzdX ← (z1^p – z0^p)/ (x1^p – x0^p)
    inc_dIdX ← (I1^p – I0^p)/ (x1^p – x0^p)

    { inicialització derivades d'interpolació}

    (dz, dI) ← (z0^p, I0^p)

    per x desde x0^p fins x1^p fer

            { Inversa de dz per obtenir el valor Zt^3D' amb perspectiva correcta }
            Zt^3D' ← 1/dz

            { Obtenció del atribut I amb perspectiva correcta}
            It^3D' ← dI* Zt^3D'

            {...}

            { Actualització lineal de les derivades dI i dZ}
            dI ← dI  + inc_dIdX
            dz ←dz  + inc_dzdX
    fper
fAlgorisme
```

2.4 Geometria

2.4.1 Interseccions

- **Intersecció entre 2 línies.**

Donades el valor de dues línies de coordenades XZ conegudes $((x1_0, z1_0), (x1_1, z1_1))$ i $((x2_0, z2_0), (x2_1, z2_1))$, podem trobar el punt de l'espai on intersecten.

Recordant l'equació d'una recta,

$$z = m \cdot x + b$$

On:

- z: Variable dependent.
- x: Variable independent.
- m: Representa el pendent de la línia.
- b: Es el desplaçament que presenta la línia quan talla a l'eix d'ordenades.

Si sabem el valor de les dues coordenades que defineixen la línia, llavors podem

$$m = \frac{z_1 - z_0}{x_1 - x_0}$$

calcular m.

I llavors trobem b.

$$b = z_0 - x_0 \cdot m$$

Coneixent m i b, associem una equació de recta a la primera línia i una altra a la segona,

$$z_i = m1 \cdot x_i + b1 \qquad \textit{Equació 2.25}$$

$$z_i = m2 \cdot x_i + b2 \qquad \textit{Equació 2.26}$$

Ara tenim dues equacions i dues incògnites, un problema solventable que el resoldrem pel mètode d'igualació (igualant la incògnita z_i),

$$m1 \cdot x_i + b1 = m2 \cdot x_i + b2$$

per tant x_i serà,

$$x_i = \frac{(b2 - b1)}{(m1 - m2)}$$

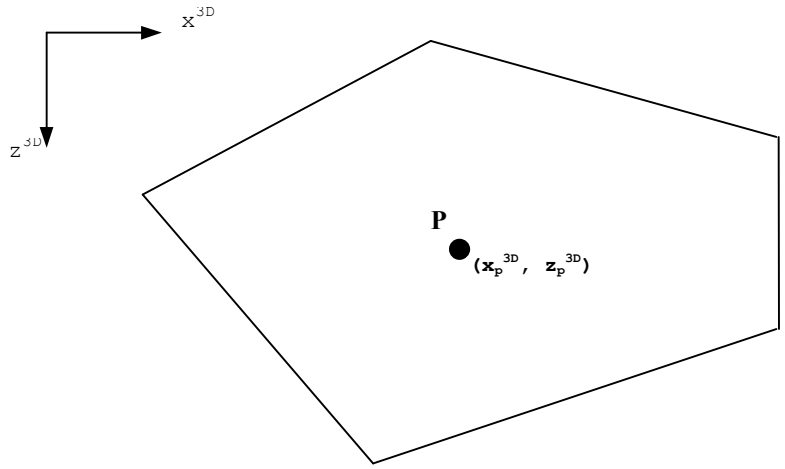
per consegüent, podem saber z_i, substituint x_i en una de les equacions 2.25 o 2.26.

2.4.2 Pertinença d'un punt a un polígon convex

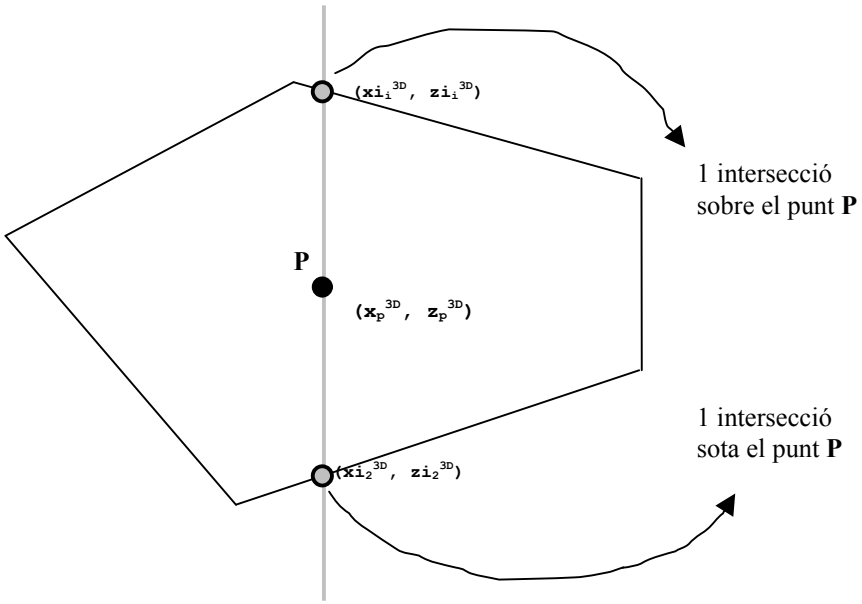
Un dels problemes de geometria més importants que podem trobar en el disseny del nostre motor serà determinar si un punt pertany a un polígon 2D convex. Aplicarem un algorisme senzill de geometria computacional per resoldre aquest problema. Abans però, presentem el seu estudi.

- **Estudi**

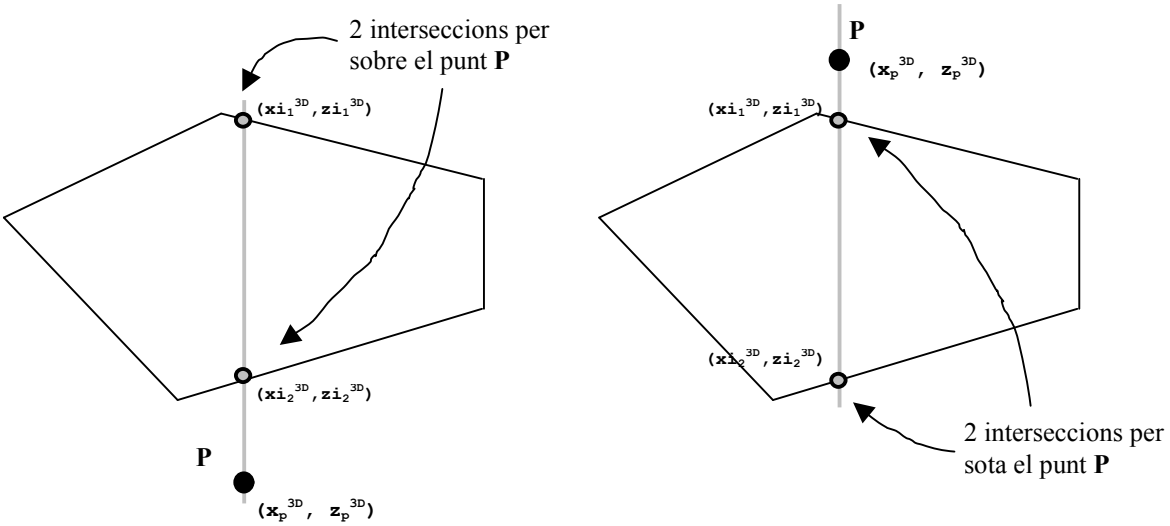
Donada la situació del punt P següent,



Podem observar que està a dins d'un polígon convex. Si llencem un raig vertical, sempre trobarem 2 interseccions (xi_1^{3D}, zi_1^{3D}) i (xi_2^{3D}, zi_2^{3D}): una sobre el punt U i una altre sota seu.



En canvi **si el punt P està fora del sector** i llencem un raig vertical **sempre** trobarem **dues interseccions sobre el punt P o dos interseccions sota seu**.

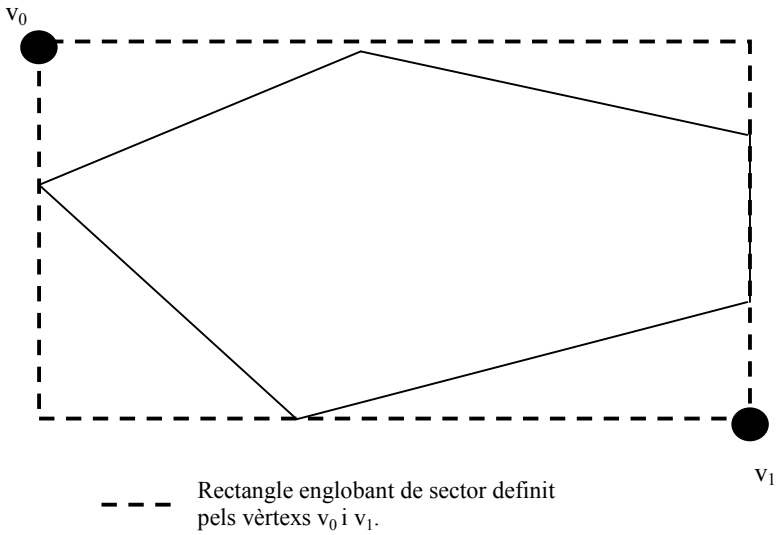


Per tant, un dels processos d'aquest algorisme serà el de calcular les dues interseccions resultants del raig vertical. Tot seguit, comprovà les interseccions respecte l'origen Z de l'ull.

$$z1_{origen} \leftarrow zi_1^{3D} - z_p^{3D}$$
$$z2_{origen} \leftarrow zi_2^{3D} - z_p^{3D}$$

Per acabar, cal saber els signes respectius de les variables $z1_{origen}$ i $z2_{origen}$. **Si ambdues tenen el mateix signe**, **llavors** sabem que les dues interseccions estan per sobre o sota del punt i per tant, **el punt NO pertany al sector**. Altrament, **si els signes són diferents, el punt pertany al sector**.

Una manera de descartar més ràpid la inexistència del punt és saber si aquest no pertany al rectangle englobant del sector que té definit pels dos vèrtexs extrems.



- **Algorisme**

L'algorisme que determina la pertinença d'un punt a un polígon convex seguint el model estudiat, seria el següent:

```
Funció Punt_Pertany_Poligon_Convex(e Punt, Poligon) retorna booleà
{ Pre: El poligon és convex, té definit el quadrat englobant i està format per línies}
Var
        Xi : taula [2] de real
        Zi:  taula[2] de real
        N_liniaPoligon, N_interseccions: enter
FVar

Si Punt Pertany a Poligon.Quadrat_Englobant llavors

        N_liniaPoligon = 1;
        N_interseccions = 0;

        mentre (N_Interseccio < 2) i (N_liniaPoligon < Poligon.N_LiniesPoligon) fer

                si Intersecta(LiniaPoligon[N_liniaPoligon], Raig_Vertical) llavors

                        (Xi[N_interseccio], Z[N_interseccio] )←Calcular_Interseccio(Raig_Vertical,Linia)
                        N_interseccio ← N_interseccio + 1
                fsi

                N_liniaPoligon ← N_liniaPoligon + 1
        fmentre


        si (N_interseccions = 2) llavors { Determinem signes de cada intersecció }

            si ((Punt.Z – Zi[1] <= 0) i (Punt.Z – Zi[2] >= 0)) o
               ((Punt.Z – Zi[1] >= 0) i (Punt.Z – Zi[2] <= 0)) llavors {Punt pertany al poligon}

                    Punt_Pertany_Poligon_Convex ← CERT
            Altrament
                    Punt_Pertany_Poligon_Convex ← FALS
            Fsi
        Altrament
            Punt_Pertany_Poligon_Convex ← FALS
        Fsi
Altrament
    Punt_Pertany_Poligon_Convex ← FALS
FSi

FFunció
```

Llistat 2.1

Capítol 3: Disseny de motor gràfic

Com es va dir a la introducció, el nostre motor gràfic a desenvolupar només generarà (o *renderitzarà*) els escenaris 3D deixant de banda la *renderització* d'objectes 3D per treballs futurs. Un motor gràfic que només *renderitzi* escenaris 3D està format per tres parts: **L'estructura de dades del mapa de món**, **el jugador** i el **renderitzador d'escenaris 3D**.

- **L'estructura de dades del mapa:** La manera com és guardarà la informació del món.

- **El jugador :** Personatge virtual que recorre el mapa per mitjà els esdeveniments d'usuari.

- **Renderitzador d'escenaris**: És qui s'encarrega de generar per pantalla els escenaris 3D en funció de la informació de jugador i l'estructura de dades.

Cal dir que **la part més important** i base **d'un motor gràfic és l'estructura de mapa**, ja que la tècnica/mecànica del *renderitzador* va en funció del tractament de la informació font, es a dir, de la informació de mapa d'aquesta estructura dependrà fer més o menys complex el nostre *renderitzador* a dissenyar.

Com es va dir a la introducció d'aquest treball, volem que el nostre motor sigui capaç de *renderitzar* escenaris com el vista a la figura 1.3 (esquerra), conscients que ha d'executar-se sobre la plataforma *Game Boy Advance* (GBA). Recordem que aquesta consola no disposa d'operacions 3D integrades per hardware i, a més, és lenta amb comparació la tecnologia que s'utilitza avui en dia. Es per això que totes les operacions implicades al nostre motor hauran de ser forçosament per software, per la qual cosa implica l'ús/disseny de tècniques algorísmiques potents que assegurin la interactivitat en el recorregut virtual. Afortunadament, la tècnica dels **portals i sectors,** documentada a [JCC02], permetrà aconseguir el nostre objectiu proposat.

Fonamentalment, la tècnica dels portals i sectors consisteix en tenir la estructura dels nostres mapes subdividits en sectors connectats per portals que apunten a un altre sector (figura 3.1). El *renderitzador* traurà màxim partit d'aquesta estructura, generant escenaris a grans velocitats. Es per això que **el nostre model de mapa s'estructurarà en sectors i portals**.

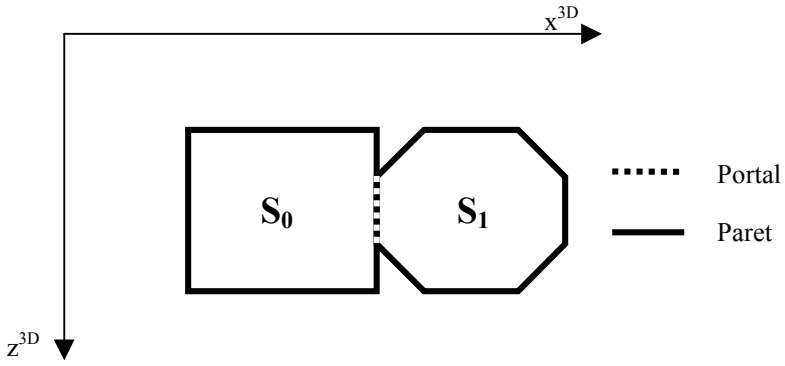


Figura 3.1 *Model de estructura de mapa que utilitzarà el nostre renderitzador. A la figura, tenim el mapa subdividit en dos sectors (S_0 i S_1) connectats per una línia discontinua (portal).*

3.1 El mapa

Com s'ha dit en la introducció d'aquest capítol, **l'estructura del mapa estarà format per sectors i portals** però, encara queda per decidir si la informació de mapa estarà representat en 3D o en 2D.

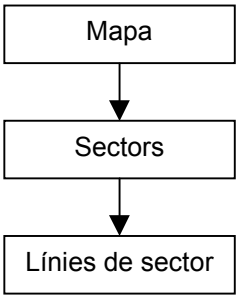
Si un mapa està representat en 3D significa tenir més quantitat de dades, fer el món molt complex i, per conseqüent, el motor gràfic haurà de tractar/processar més informació durant el *renderitzat*, per la qual cosa no es recomanable en el present treball. Està clar que nosaltres volem aconseguir la interacció, per tant utilitzarem mapes representats en 2D (un pla). Per això, l'estructura de informació del nostre mapa seguirà el model de la figura 3.1.

Però, com es podrà generar un escenari 3D a partir de la informació d'un mapa 2D?

Al a la secció 3.1.2 veurem com, a partir de imposar una alçada per-sector en terra i sostre, serà possible la generació d'un món 3D. A continuació, descrivim les estructures de dades que es faran servir pel mapa del joc.

3.1.1 Estructura de mapa

L'estructura del mapa està constituït per la següent jerarquia d'objectes.



Tot seguit s'explica l'estructura de cadascun d'aquests objectes, començant pel nivell més baix (les línies).

- **Les línies de sector**

Les línies tindran un número identificador i, com és normal, estaran definides per dos vèrtexs en pla XZ (figura 3.3).

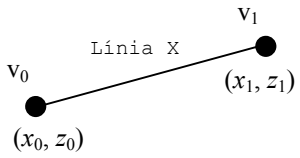


Figura 3.2 *Exemple d'una línia amb identificador X, definida per un vèrtex inicial (v_0) i un vèrtex final (v_1).*

Cal distingir dos tipus de línies dins un sector: línies sòlides i les línies transparents, que, respectivament, **representen la paret** i **el portal** corresponent a la renderització de l'escenari 3D.

Una vista de l'estructura de línia de sector seria la següent:

```
TUPLA tLiniaSector

   Id: enter
   Linia: tLinia
   Portal: enter

FTUPLA
```

On:

- L'identificador de la línia.
- L'estructura de "tLinia" és simplement la disposició dels dos punts que defineixen la línia en l'espai 2D:

```
TUPLA tLinia

   x0, z0, x1, z1 : enter

FTUPLA
```

- "Portal" declara la línia com a paret si el seu valor és = 0. Quan el valor sigui > 0 serà portal on seu valor identificarà el sector que connecta.

- ## El sector

Un sector esta definit per una sèrie de línies que formen una figura, la qual ha de ser **convexa (veure figura 3.3), convenció ja establert a la tècnica dels portals i sectors**. Cada sector tindrà un numero identificador dins la seva estructura de dades.

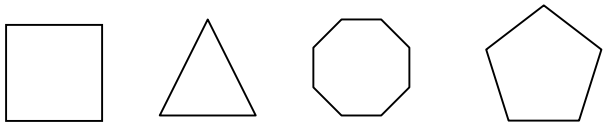


Figura 3.3 *col·lecció de figures convexes*

Una vista general de l'estructura de sector seria la següent,

```
TUPLA tSector
   Id : enter          {identificador del sector }
   Nlinies: enter
   LiniaSector: taula [0,...,Nlinies] de tLiniaSector
FTUPLA
```

De vegades, l'estructura de sector esta feta principalment de vèrtexs i prou. Nosaltres ho hem estructurat per línies ja que disposaran de informació per accelerar el processat durant el seu renderitzat.

Assumim que **les línies** de sector **hauran d'estar ordenades creixentment i en sentit horari** (figura 3.4). Naturalment, cada línia haurà d'estar connectada de forma correcta, o sigui, el vèrtex final d'una línia ha de coincidir amb el vèrtex inicial de la següent.

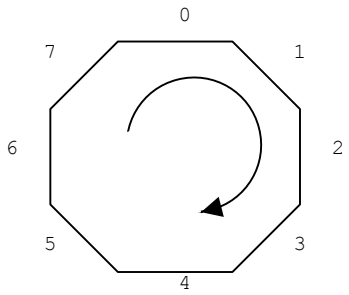


Figura 3.4 *Sector format per 8 línies ordenades en sentit horari*

- **El mapa**

Finalment, el mapa contindrà la informació de tots els sectors emmagatzemats en una taula. L'estructura del mapa és tant simple com la següent:

```
TUPLA tMapa

   NSectors: enter

   LiniaSector: taula [0,...,NSectors] de tSector

FTUPLA
```

3.1.2 Atributs de nivells d'altura

Així com ja s'ha vist, els nostres mapes estan definits en 2D. No podem generar un món 3D a partir de la informació de mapa que disposem, però en canvi, si imposem un alçada a les parets que tindrà el nostre món i, el terra i sostre d'aquest són completament plans, el renderitzador podrà serà capaç de generar el respectiu món 3D per pantalla. Per exemple, podríem imposar un nivell d'altura global de 64 unitats en el món 3D resultant la *renderització* equivalent com la que s'observa en la figura 3.5.

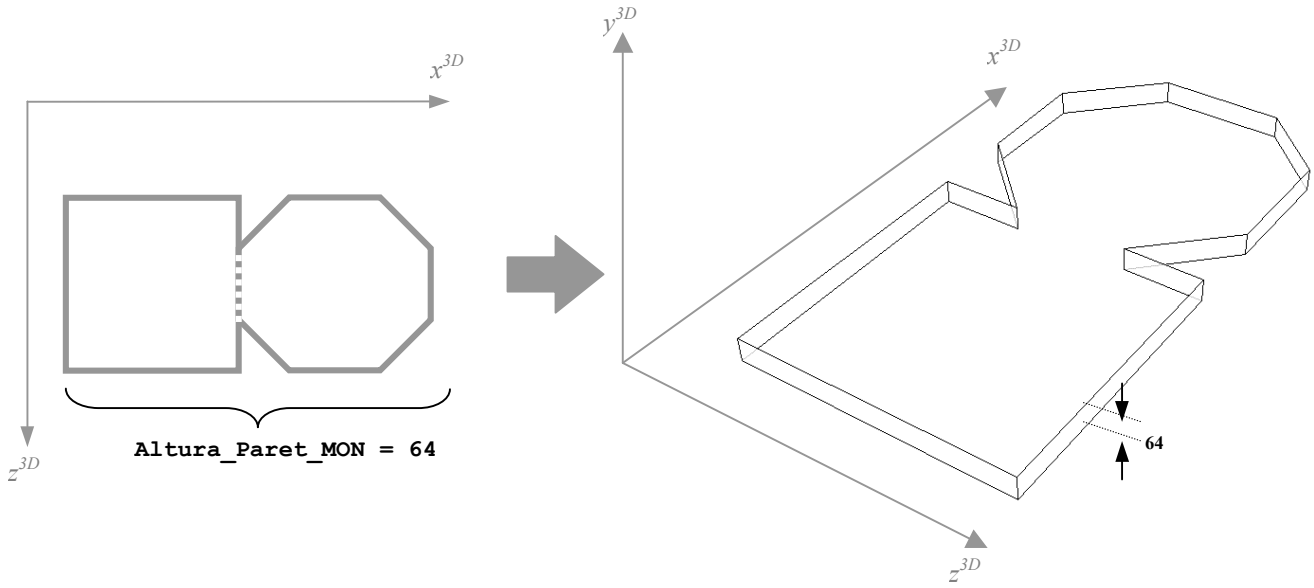


Figura 3.5

Tal com podem observar, el terra i sostre del mon generat en l'exemple 3.5 és completament pla i amb una altura de paret constant de 64 unitats en alçada, un tipus de món poc atractiu.

D'altra banda, si en vés d'imposar una altura de paret global s'hi guarda nivell de terra-sostre per sector s'aconseguiria *renderitzar* un món 3D com el que s'observa a l'exemple de la figura 3.6. En definitiva, **l'afegit d'aquests atributs a l'estructura de sector permetrà *renderitzador* escales, finestres, piscines, portes, voreres, etc.**

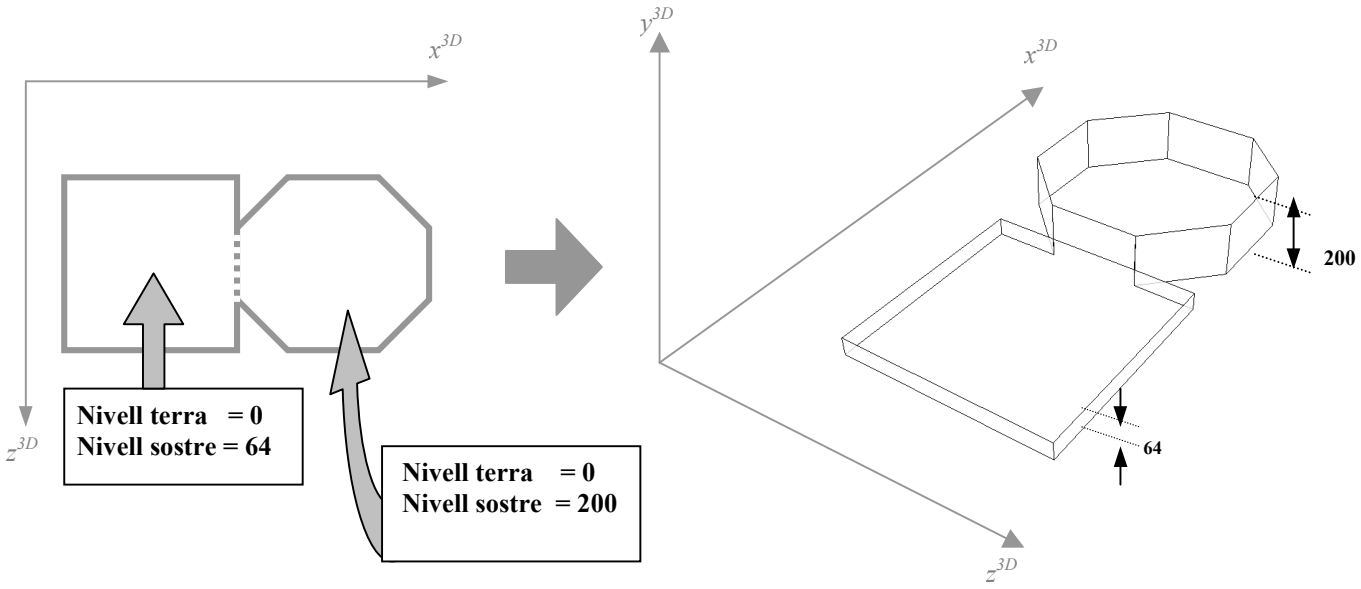


Figura 3.6

L'estructura de sector hi caldria afegir els camps AlturaTerra i AlturaSostre,

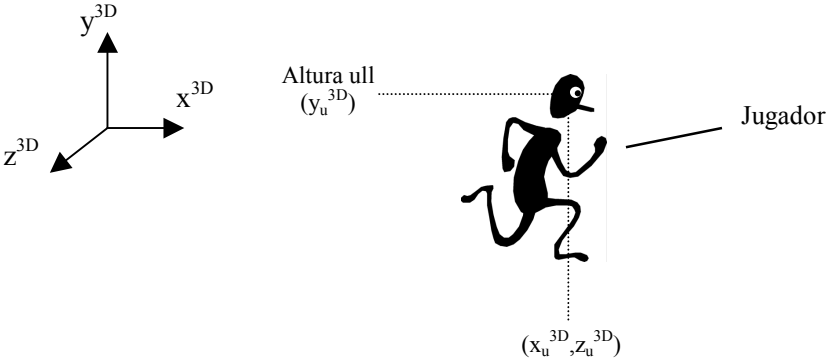
```
TUPLA tSector
   (...)
   AlturaTerra, AlturaSostre: enter
   (...)
FTUPLA
```

A la següent figura mostra un exemple de escenari renderitzat amb desnivells,

3.2 El Jugador

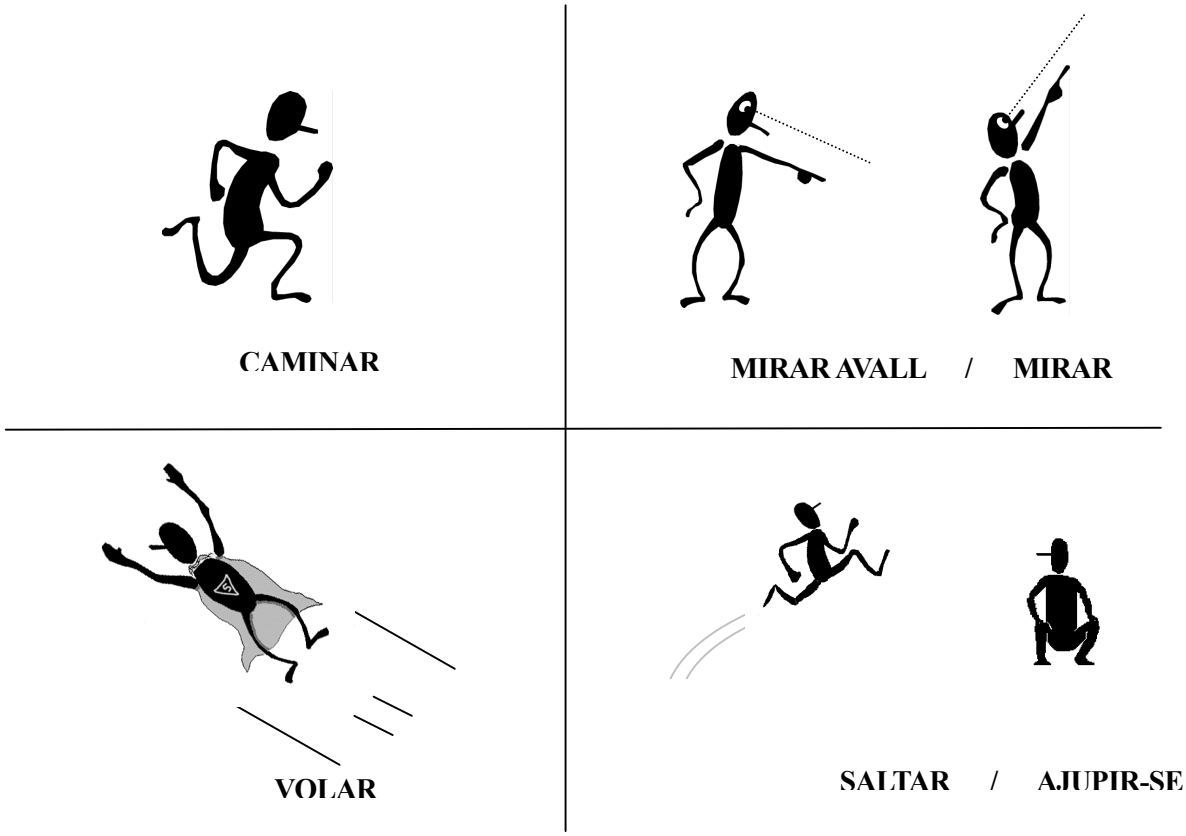
- **Descripció**

El jugador és el personatge virtual que efectua les accions d'agafar un objecte, desplaçar-se, saltar, ajupir, etc en vers el món virtual que l'envolta, controlades per part de l'usuari mitjançant els controls de la màquina. En el jugador s'incorpora l'*ull* amb les mateixes coordenades XZ de jugador, però, igual que en l'estructura de dades del sector, caldrà afegir una variable que expliqui la seva altura en el món 3D, imprescindible a l'hora de renderitzar el món i, també, per fer gestió de col·lisions.



- **Accions que podrà efectuar el jugador**

Les següents accions que podrà realitzar el nostre jugador entorn el món 3D, seran les següent:



3.2.1 Realització de les accions

3.2.1.1 Acció "caminar"

- **"Caminar Endavant"**

L'acció "caminar endavant" es podrà realitzar mitjançant el control de direcció endavant de la màquina. Donat que el mapa està definit en 2D l'acció "caminar endavant" del jugador és un simple desplaçament 2D. El desplaçament 2D es pot fer de manera simple a partir de l'angle de direcció, definit en el jugador.

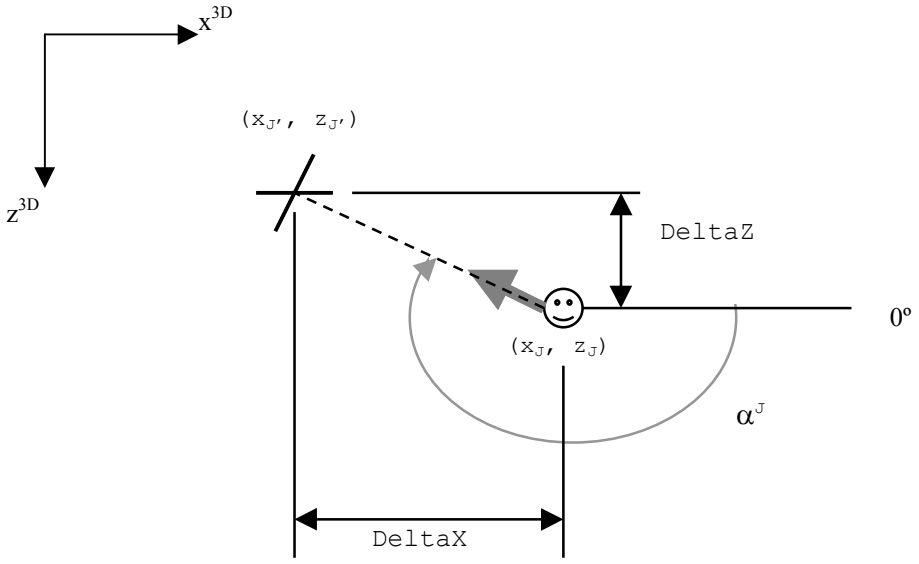


Figura 3.7a

A la figura 3.7a podem observar que les coordenades de jugador (x^J, z^J), associades a les coordenades de ull, i amb l'angle de direcció α^J apuntant al 3er quadrant (segons les orientacions definides a la secció 2.1.1) . Podem obtenir els valors delta (deltaX i deltaZ) amb la porció de desplaçament adequat i amb el signe corresponent segons el quadrant que apunta l'angle de direcció, amb les següents formules trigonomètriques:

$\texttt{Delta\_X} \leftarrow \cos(\alpha^J)$
$\texttt{Delta\_Z} \leftarrow \sin(\alpha^J)$

Les funcions *sin()* i *cos()* retornen números a l'interval de [0,...,1]. Si volem aconseguir més velocitat, caldrà multiplicar-ho per una constant de velocitat al resultat de cada funció trigonomètrica.

$\textbf{Delta\_X} \leftarrow \cos(\alpha^J) * \textbf{Velocitat}$ *Equació 3.1*
$\textbf{Delta\_Z} \leftarrow \sin(\alpha^J) * \textbf{Velocitat}$ *Equació 3.2*

Com que les funcions trigonomètriques retornen del signe adequat segons el quadrant que apunta angle de direcció, només cal sumar els increments a les coordenades de jugador. D'aquesta manera s'aconsegueix desplaçar el jugador per l'espai 2D.

$$x^{J'} \leftarrow x^{J} + Delta\_X$$
$$z^{J'} \leftarrow z^{J} + Delta\_Z$$

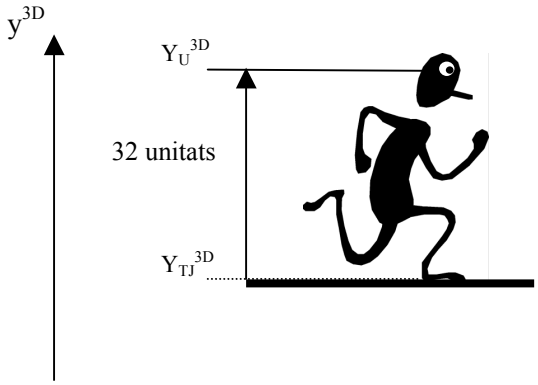
- **"Caminar Enrera"**

L'acció "caminar enrera" es podrà realitzar mitjançant els control de direcció enrera de la màquina. Igual que abans el problema és un desplaçament 2D, a diferència només cal invertir les deltes XZ, obtingudes a les respectives equacions 3.1 i 3.2.

$$x^{J'} \leftarrow x^{J} - Delta\_X$$
$$z^{J'} \leftarrow z^{J} - Delta\_Z$$

- **Altura d'*ull* en acció caminar**

S'ha establert que, quan el jugador estigui caminant, l'altura d'ull estarà a 32 unitats respecte el terra del sector on resideix el jugador (sector-jugador).



On:

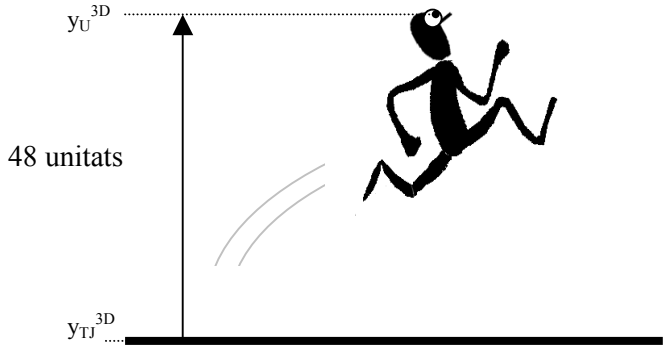
y_U^{3D} = Altura ull
y_{TJ} = Altura terra del sector on resideix el jugador

3.2.1.2 Acció "Saltar" "Caure" i "Ajupir-se"

L'acció "saltar"/ "ajupir-se" podrà realitzar-se mitjançant un botó/tecla preprogramat de la màquina. L'acció "caure" passarà simplement quan el jugador travessi un portal on existeix un desnivell profund entre el sector de jugador i el sector connectant.

- **"Saltar"**

Per donar l'efecte de salt quan es premi el botó pertinent al de "saltar", només caldrà posicionar l'*ull* a una altura superior a la altura original (32 unitats respecte terra). S'ha decidit que el salt situarà l'*ull* a una altura màxima de **48 unitats** respecte el terra del sector-jugador.



Pel motiu de fer l'acció de salt més suau durant la renderització, s'utilitzarà la següent fórmula física. En ella explica **el llançament vertical cap amunt** d'un objecte amb una velocitat inicial (v_0) determinada. Segons el temps transcorregut, incrementarem l'altura d'*ull* al valor de velocitat mitjançant de l'equació 3.3.

$$Y_U^{3D} = Y_U^{3D} + v_0 \cdot t + \frac{1}{2} \cdot g \cdot t^2 \qquad \text{(unitats/segon)} \qquad \textit{Equació 3.3}$$

$$g = 9'8$$

- **"caure"**

Quan el jugador arriba a tocar el punt màxim del salt o passa per un desnivell de terra per sota l'actual, **començarà el període de caiguda** amb velocitat inicial 0. Per simular una caiguda més real, s'utilitzarà la següent fórmula física que explica **el llançament vertical cap avall** d'un objecte amb velocitat inicial (v_0) nula.

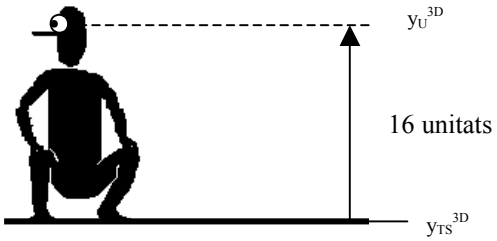
$$Y_U^{3D} = Y_U^{3D} - g \cdot t \qquad \text{(unitats/segon)} \qquad \textit{Equació 3.4}$$

$$g = 9'8$$

Cal observar que, tant l'equació 3.3 com la 3.4 es poden precalcular en una taula (*Look Up Table* –LUT-) ja que només va en funció de *t*.

- **"Ajupir-se"**

Quan es premi el botó pertinent al de "ajupir-se", només caldrà posicionar la altura d'*ull* inferior a la altura original. S'ha decidit que la altura d'*ull* quan el jugador estigui sentat serà de **16 unitats** respecte el terra sector-jugador.



3.2.1.3 Acció " vall"

L'acció "mirar amunt"/"mirar avall" es podrà realitzar mitjançant dos botons/tecles preprogramades de la màquina.

Al capítol 2 (fonaments) es va parlar que, per convenció, l'ull es situava al centre de projecció y (*centre_y*^{P}), on el seu valor era la meitat de resolució y (y_res/2).

Per fer l'efecte de mirar amunt/avall, farem variable *centre_y*^{P}, per consegüent, es desplaçarà la imatge renderitzada verticalment segons el valor actual. La figura 3.7b, mostra un exemple en que *centre_y*^{P} es variable a l'interval [y_res/2 -y_res,.., y_res/2 +y_res/2]. La part discontinua de la figura A, indica el desplaçament màxim/mínim del pla projecció que causarà la variació del *centre_y*^{P}.

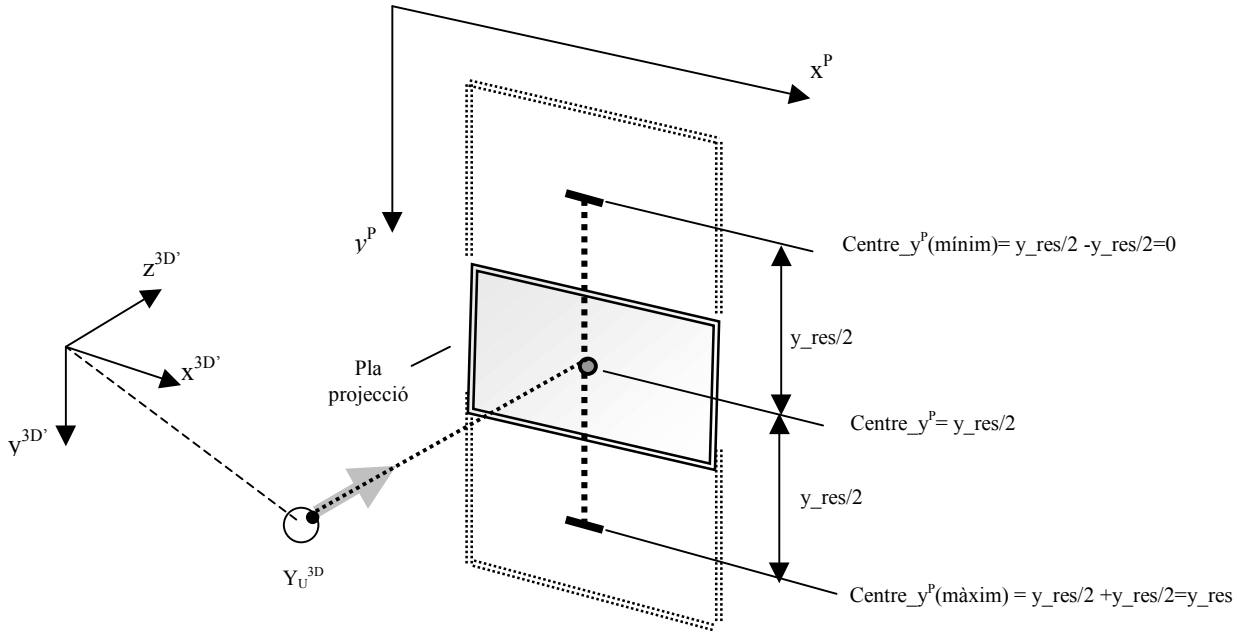


Figura 3.7b

Així doncs per permetre al jugador efectuar l'acció de mirar amunt/avall, segons el que s'ha explicat, caldria executar el senzill tros de codi següent:

```
const MIN_PITCH            0
const MAX_PITCH            y_res
const VELOCITAT_PITCH      8       {Per exemple }


si Apreta_Tecla_Mirar_Amunt() llavors

     si((centre_y^p + VELOCITAT_PITCH) > (MAX_PITCH))
            centre_y^p ← MAX_PITCH
     altrament
            centre_y^p ← centre_y^p + VELOCITAT_PITCH
     fsi

altrament

     si Apreta_Tecla_Mirar_Avall()llavors
         si((centre_y^p -VELOCITAT_PITCH) < (MIN_PITCH)) llavors
                    centre_y^p ← MIN_PITCH;
         altrament
            centre_y^p ← centre_y^p - VELOCITAT_PITCH
         fsi
     fsi
fsi
```

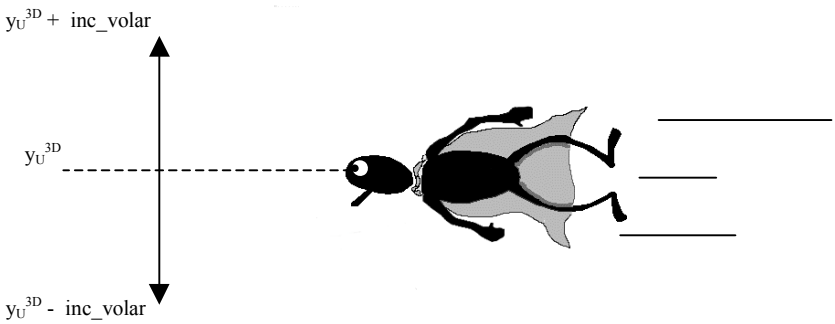
VELOCITAT_PITCH, és un valor constant, que explica la velocitat de gir de coll del jugador.

3.2.1.4 Acció "volar"

Abans de poder "volar", el jugador haurà de entrar a mode vol (només volar) quan es premi un botó preprogramat de la màquina i es tornarà a mode normal (caminar, saltar, etc.) quan es torni a prémer el mateix botó.

Un cop el jugador hagi entrat en mode vol, disposarem de dos altres botons preprogramat que li permetrà al jugador "volar amunt" o "volar avall".

Per realitzar l'acció "volar", només cal incrementar o es decrementar de manera lineal l'altura d'*ull* (y_U^{3D}) amb un valor constant de velocitat amb un valor constant.



3.3 El renderitzador 3D

El **renderitzador 3D** és el mòdul que **s'encarregarà de mostrar** per pantalla el resultat del món 3D vist des del FOV de *l'ull*. Darrera el procés de visualització existeixen processos que resolen problemes geomètrics, realitzen càlculs de perspectiva, pinten els resultats per pantalla pixel-per-pixel, etc. Tots aquests algorismes a implementar han de ser eficients, perquè, tal com portem dient des del principi, volem aconseguir un renderitzador a temps real sota la consola GBA.

- ## Ray-Casting

Una de les tècniques més conegudes eficients i simples utilitzades en renderitzacions 3D, és l'anomenada tècnica *ray-casting* (llançament de raigs). El capítol 1 del manual històric del projecte (evolució projecte), es va estudiar un ray-casting amb la intenció incorporar-lo al nostre renderitzador 3D final. El *ray-casting* estudiat consistia en llançar un raig per-columna traçar el raig pel mapa fins a intersecar una línia sòlida. Per tant, amb la distància observador-intersecció raig-línia s'aconseguirà escalar una tira de paret per pantalla.

Durant el traçat de raig, es resolen problemes geomètrics per esbrinar la línia de sector que interseca amb el raig (línia intersecció). Si la línia és portal l'ignorarà i prosseguirà el seu recorregut pel mapa fins a intersecar una paret, tal com podem mostrar l'exemple de la figura 3.8.

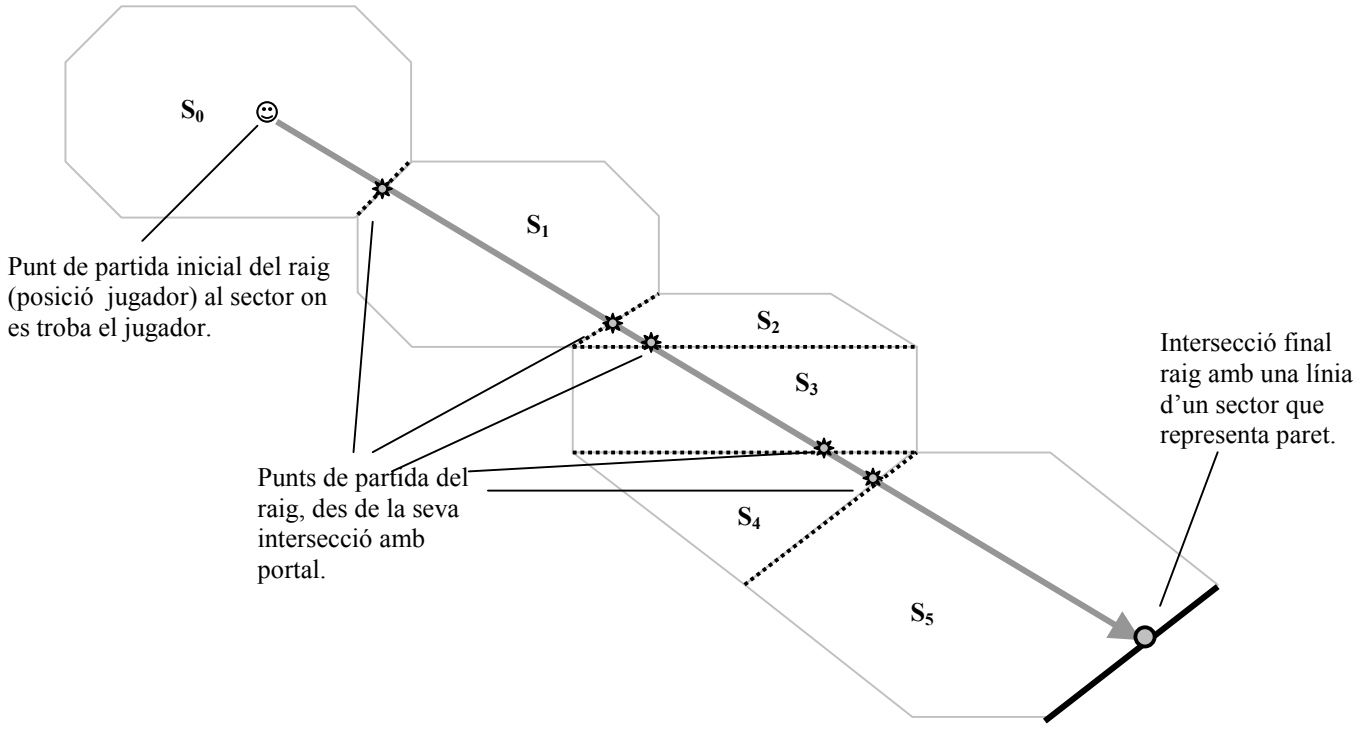


Figura 3.8 *exemple gràfic d'un traçat de raig en un mapa basat en portals i sectors.*

Quan trobi la intersecció raig-paret, es retornarà la informació de intersecció i/o altres paràmetres necessaris pel *renderitzat*.

El procés de traçat de raig es defineix en el següent pseudocodi de funció recursiva,

```
Funció TraçarRaig(entrada columna, sector)
        Obtenir_Linia_Interseccio_Raig(Columna,Sector)
        { Processar altres informacions...}
        Si raig interseca portal llavors { Segueix el seu transcurs recursivament}
                Retorna TraçarRaig(columna, portal. ID_Sector)
          Altrament {Atura el traçat de raig i  retornant info necessaria pel renderitzart paret}
                Retorna Informació_necessaria_pel_renderitzat
        FSi
FFunció
```

Llistat 3.1a

Podem comprovar que si la pantalla disposa de **n** columnes i, en el cas pitjor, cada raig traspassa **m** portals abans de topar amb la paret, l'ordre algorísmic és de O(n·m). És per aquest motiu que es va descartar utilitzar aquesta tècnica. En aquest capítol s'estudiarà una manera perquè amb només **el llançament d'un sòl raig s'aconsegueixi *renderitzar* tot un sector**.

• Renderització per-sector, la tècnica

Com s'ha dit, amb del llançament es tindrà la informació necessària com per ***renderitzar* un sector.** Si durant la *renderització* del sector inicial es troba un portal, es renderitzarà els sector compresos pel portal de forma recursiva**.** Cal dir que els sectors a *renderitzar* tenen un ordre de processament i que, per la forma del nostre algorisme, sempre es *renderitzaran* primer els sectors pròxims al jugador i en últim lloc els sectors més llunyans a ell. Aquest ordre de *renderització* també se'l anomena renderització *front to back drawing* (de endavant cap enrera).

Per deixar clar la mecànica de la *renderització* utilitzada en el nostre motor, a la següent pàgina donem un exemple.

- **Exemple d'una renderització per-sector recursiu**

Donat el sector A inicial (sector-jugador), segons la direcció d'angle d'*ull* és llançaria el primer raig per la columna 0 de pantalla causant la primera part de *renderització* del sector A (dreta) des de la columna 0 fins trobar el portal que connecta al sector B (veure figura 3.9)

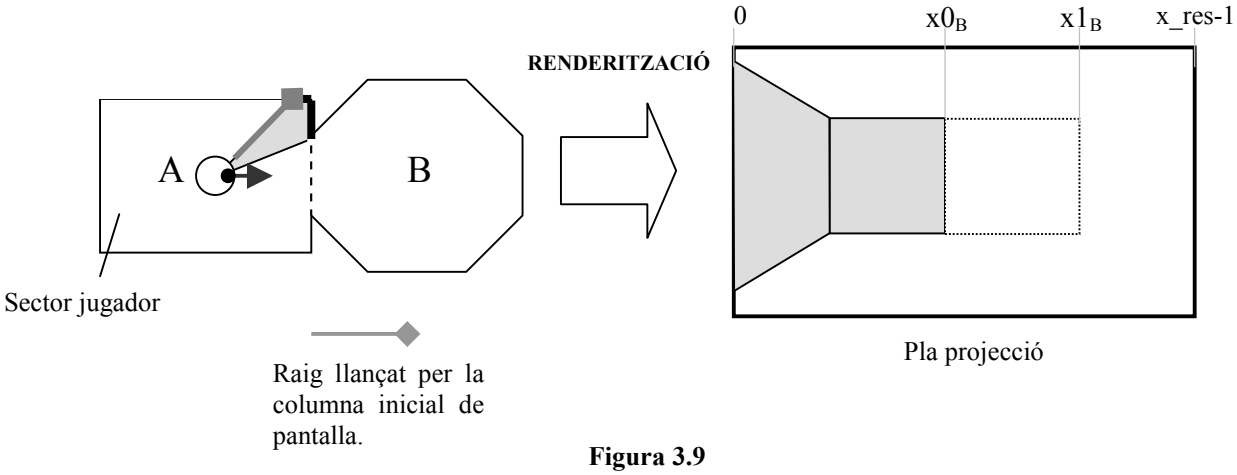


Figura 3.9

Tot seguit es crida la funció de *renderització*, perquè processi el sector B $\in[x0_B..x1_B]$. Igual que abans, es llançarà un raig per la columna $x0_B$ i, així, s'aconseguiria *renderitzar* tot el sector B complet ja que no existeix cap més portal definit en ell. Si el sector B tingues altres portals a *renderitzar*, caldria doncs es cridaria de manera recursiu els sectors implicats i així successivament (veure figura 3.10).

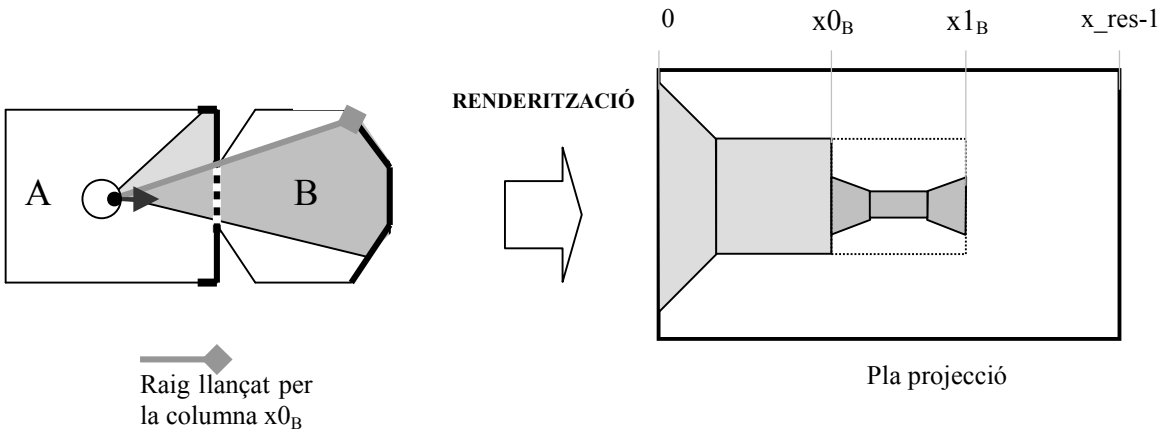


Figura 3.10

Per últim, s'acabaria de *renderitzar* el sector A al haver acabat de *renderitzar* el sector B a interval [$x1_B$.. x_{N-1}], com es veu a la figura 3.11.

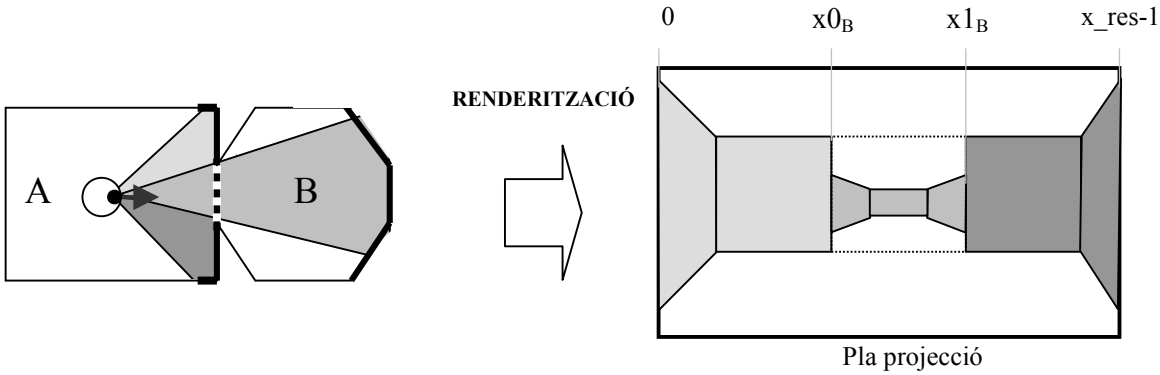


Figura 3.11

- ## Base del renderitzador

Vist l'exemple de *renderització*, dona la idea de quina serà la base del nostre renderitzador, que serà l'algorisme que *renderitza* un sector, molt semblant al vist del llistat 3.1a.

```
Acció RenderitzarSector(entrada x1,x2, Observador,sector)
        Obtenir_Linia_Interseccio_Raig(x1,Sector)
        { Processar altres informacions...}
      per cada línia ∈ sector fer
        (x1,x2) ← ProjeccióPantalla(linia)
        RenderitzarPartSector(x1,x2, Jugador, linia) {Renderitza paret o nivells inferiors/superiors portal[1]}

        Si linia és portal llavors { Es renderitza els sector/s recursivament}
            RenderitzarSector(x1,x2, Jugador, ID_Sector(linia))
        FSi

        Linia ← LiniaSeguent(Linia)
      FPer
    FAcció
```

Llistat 3.1b

[1] A la secció 3.3.2 veurem que la renderització d'una part de sector implica tenir una la renderització paret o la renderització dels desnivells d'un portal.

3.3.1 Renderització d'un sector

Aquesta secció s'explicarà l'algorisme base de la *renderització* d'un sector. Abans però, veiem un exemple de *renderització* d'un sector (figura 3.12).

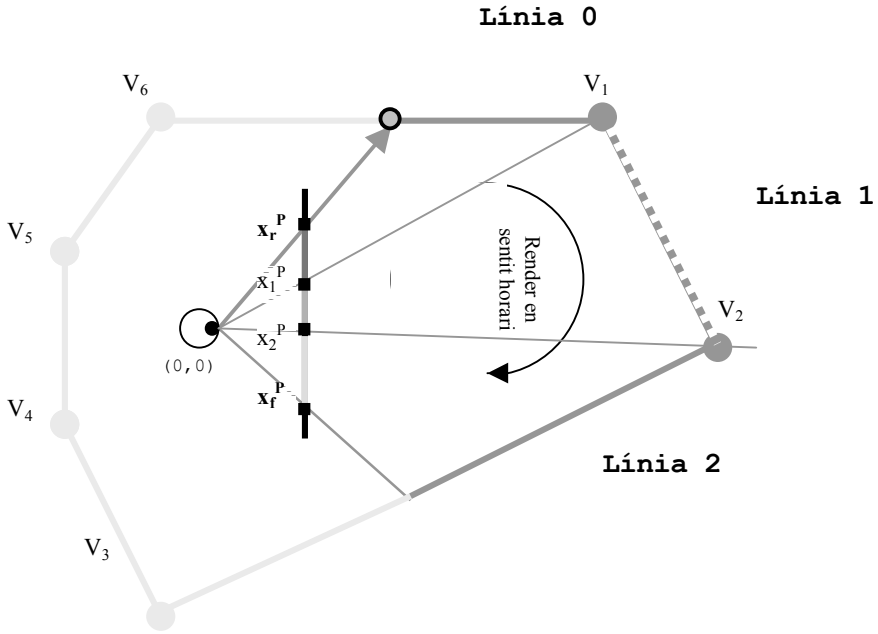


Figura 3.12

A l'exemple de la figura 3.12, podem observar la *renderització* d'un sector al interval de columnes $[x_r^P, x_f^P)$. Es llença el raig per la primera columna (x_r^P), tal que interseca amb la **línia 0** del sector. Per projecció del vèrtex V_1, s'obté x_1^P i llavors es *renderitza* la part de sector a l'interval $[x_r^P, x_1^P)$.

A la línia següent (**línia 1**) i per projecció del vèrtex V_2, s'obté x_2^P, i és *renderitza* la part de sector al interval de columnes $[x_1^P, x_2^P)$, però cal tenir en compte que la línia 1 és portal per la qual cosa s'efectuarà la crida recursiva per poder *renderitzar* el sector connectant.

Un cop acabada la *renderització* del sector connectant, es passa a la **línia 2** i es projecta el vèrtex V_3. Podem comprovar que la projecció surt fora l'interval màxim de la *renderització* (x_f), per tant, finalitza el procés amb la *renderització* de la part de sector a l'interval $[x_2^P, x_f^P)$.

En resum, l'algorisme base de la *renderització* d'un sector serà el següent:

```
Accio RenderitzarSector(entrada xr, xf, Observador, Sector) { Renderització sector [xr...xf] }

    LiniaActual ← LiniaInterseccio(xr, Observador, Sector) { S'Obté la línia de intersecció }

    x1 ← xr
    fi ← FALS

    mentre no fi fer {Renderització en sentit horari[2] }

            x2 ← ProjectarVertex(Sector.Linia[LiniaActual].V1)

            si x2 > xf llavors
                    fi ← CERT
                    x2 ← xf
            fsi

            Parametres← CalcularParametresRenderitzacióPartSector(Observador,Linia)

            {Renderitza paret, o nivells inferiors/superiors de portal}

            RenderitzarPartSector(x1, x2-1, Parametres, Linia)

            Si Linea és portal llavors {Renderitza portal de manera recursiva}
                    RenderitzarSector(x1,x2-1, Observador, ID_Sector(LiniaActual))

            FSi


            LiniaActual ← LiniaSeguent(LiniaActual)
            x1 ← x2

    Fmentre
FAccio
```

Llistat 3.1c

[2] Recordem que a la secció 3.2.1 (sectors), s'ha convingut la connexió de línies en sentit horari.

3.3.2 La línia de intersecció

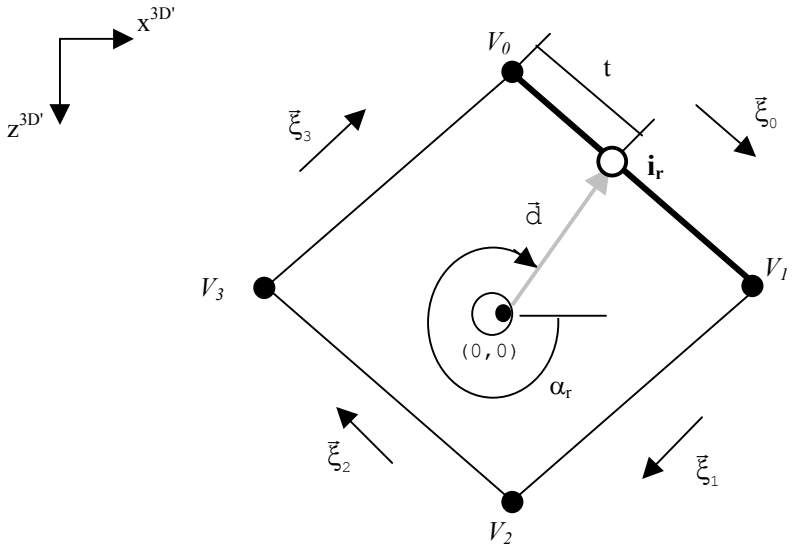


Figura 3.13

Observant figura 3.13, sigui ***i*** l' índex al vèrtex i de línia de sector, $\vec{\xi}_i$ és el vector de direcció de la línia,

$$\vec{\xi}_i = V_{i+1} - V_i$$ *Equació 3.5*

$\vec{d}$ és el vector de direcció del raig, que es,

$$\vec{d} = \begin{pmatrix} \sin(\alpha_r) \\ \cos(\alpha_r) \end{pmatrix}$$ *Equació 3.6*

t és la distància intersecada de la línia des del seu origen en tant per 1.

$$t = \frac{\left| i_r - V_i \right|}{\left| V_{i+1} - V_i \right|}$$ *Equació 3.7*

Pel fet de que un vèrtex és comú a la línia actual i l'adjacent. Per convenció, establim que el primer vèrtex pertany a la línia actual mentre que l'últim vèrtex pertany a la línia següent. Per això, si t = 1, voldrà dir que la intersecció coincideix amb l'últim vèrtex de la línia actual, però cal retornar la següent. Per això que perquè sigui una intersecció valida, **$t \in [0,1)$**.

L' algorisme per trobar la línia de intersecció és el següent,

```
Algorisme LiniaInterseccio

        interseccio ← FALS
        i ← 0

        mentre (i < n_vertexs) i no (interseccio) fer
                Si ( ( ξ_i  x  d ) ≤ 0) llavors
                        Calcular t_i
                        Si t_i ∈ [0,1) llavors
                                interseccio ← CERT
                        Fsi
                Fsi

                i ← i+1

        Fmentre
Falgorisme
```

Llistat 3.2

On,

$(\vec{\xi}_i \times \vec{d})$: És el producte vectorial, on es determina la intersecció segons la direcció de línia i raig. El resultat es treu de la següent matriu,

$$\vec{\xi}_i \times \vec{d} = \begin{pmatrix} i & j & k \\ d.x & d.z & 0 \\ \xi_i.x & \xi_i.z & 0 \end{pmatrix} = \left(0,\ 0,\ d.x \cdot \xi_i.z - d.z \cdot \xi_i.x \right)$$

Podem observar que, concretament es comprova el sentit en la component Y: si Y > 0 està per sota el pla, altrament està per sobre.

L'efecte que té al nostre algorisme es el següent,

> La direcció de línies segons disposem a la estructura de dades és en sentit horari pel que si el producte vectorial $(\vec{\xi}_i \times \vec{d})$ és positiu, vol dir que la direcció de línia no compleix un sentit correcte per la seva intersecció (anti-horari, per exemple), respecte el sentit del raig. En canvi si el producte vectorial és negatiu, vol dir que la línia és candidata a ser intersecada.

- **Càlcul de t**

L' últim pas per determinar si la nostra línia és la de intersecció és calcular ***t***. Si t $\in$ [0,1) serà la nostra L.I. Primerament definim les equacions del raig (r_p) i la línia (r_c),

$$r_p = r_0 + s \cdot \vec{d}$$ *Equació 3.8*

$$r_c = v_i + t \cdot \vec{\xi}_i$$ *Equació 3.9*

Volem trobar el punt on la recta i el raig intersecten, es a dir r_p=r_c. Igualant equacions,

$$r_0 + s \cdot \vec{d} = v_i + t \cdot \vec{\xi}_i$$

El vector r_0 és l' origen del observador pel que queda l'anul·lat,

$$s \cdot \vec{d} = v_i + t \cdot \vec{\xi}_i$$

Desenvolupant,

$$s \cdot \vec{d} = v_i + t \cdot \vec{\xi}_i$$

$$s \cdot \begin{pmatrix} \sin(\alpha_r) \\ \cos(\alpha_r) \end{pmatrix} = \begin{pmatrix} v_i.x \\ v_i.z \end{pmatrix} + t \cdot \begin{pmatrix} \xi_i.x \\ \xi_i.z \end{pmatrix}$$

$$\begin{pmatrix} s \cdot \sin(\alpha_r) \\ s \cdot \cos(\alpha_r) \end{pmatrix} = \begin{pmatrix} v_i.x + t \cdot \xi_i.x \\ v_i.z + t \cdot \xi_i.z \end{pmatrix}$$

Observem que tenim un sistema de 2 equacions amb dues incògnites (s, t) i, per tant, el sistema té solució.

$$s \cdot \sin(\alpha_r) = v_i.x + t \cdot \xi_i.x$$
$$s \cdot \cos(\alpha_r) = v_i.z + t \cdot \xi_i.z$$

El resoldrem el mètode de igualació,

$$s \cdot \sin(\alpha_r) = v_i.x + t \cdot \xi_i.x \Rightarrow s = \frac{v_i.x + t \cdot \xi_i.x}{\sin(\alpha_r)}$$

$$s \cdot \cos(\alpha_r) = v_i.z + t \cdot \xi_i.z \Rightarrow s = \frac{v_i.z + t \cdot \xi_i.z}{\cos(\alpha_r)}$$

Desenvolupant trobem t com,

$$\frac{v_i.x + t \cdot \xi_i.x}{\sin(\alpha_r)} = \frac{v_i.z + t \cdot \xi_i.z}{\cos(\alpha_r)}$$

$$\cos(\alpha_r)(v_i.x + t \cdot \xi_i.x) = \sin(\alpha_r)(v_i.z + t \cdot \xi_i.z)$$

$$\cos(\alpha_r) \cdot v_i.x + \cos(\alpha_r) \cdot t \cdot \xi_i.x = \sin(\alpha_r) \cdot v_i.z + \sin(\alpha_r) \cdot t \cdot \xi_i.z$$

$$t \cdot (\xi_i.x \cdot \cos(\alpha_r) - \xi_i.z \cdot \sin(\alpha_r)) = \sin(\alpha_r) \cdot v_i.z - \cos(\alpha_r) \cdot v_i.x$$

$$t = \frac{\sin(\alpha_r) \cdot v_i.z - \cos(\alpha_r) \cdot v_i.x}{\xi_i.x \cdot \cos(\alpha_r) - \xi_i.z \cdot \sin(\alpha_r)}$$

Si $t \in [0,1)$ s'ha trobat la L.I i, per consegüent, per mitjà l'equació 3.9 podem trobar la intersecció $\mathbf{i_r}$.

3.3.3 La renderització de part de sector

3.3.3.1 Renderització de parets

La *renderització* d'una paret serà **aconseguir pintar per pantalla una col·lecció de tires verticals** entre la columna x_0^P i la columna x_1^P de diferents escales segons la distància respecte l'observador, prenent com exemple la figura 3.14. En aquesta figura, es vol *renderitzar* una paret coneixent només la informació dels vèrtexs V0,V1 del segment de línia i l'alçada de terra i sostre del sector S.

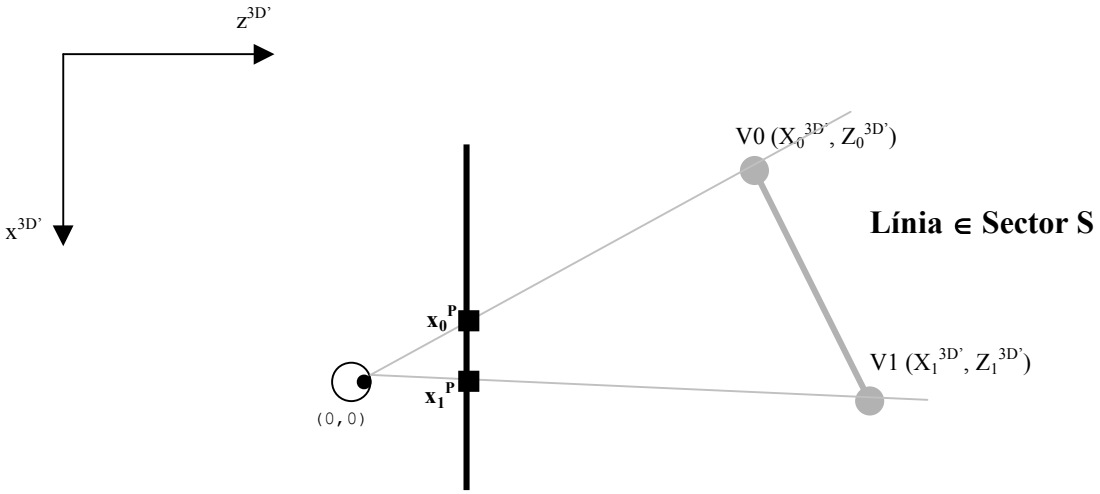


Figura 3.14

El procés del *renderitzat* de paret és el següent:

1. **Amb els valors d'alçada de terra i sostre que conté la informació del sector S**, projectem en *y* els punts de sostre i terra obtenint les cotes $y_{0\_sostre}^P$, $y_{1\_sostre}^P$, $y_{0\_terra}^P$ i $y_{1\_terra}^P$ (figura 3.15).

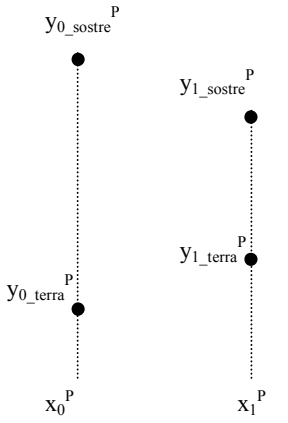


Figura 3.15 *Projecció en alçada*

2. Amb dos algorismes anomenats *Digital Diferential Analizer* (DDA), es rasteritza els extrems de paret (sostre i terra), d'aquesta manera s'obtenen les cotes $y1^P$, $y2^P$ pel pintat de tira en cada iteració (veure figura 3.16).

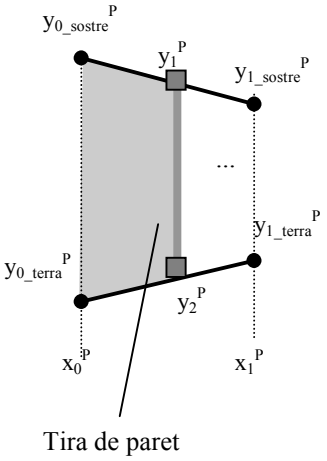


Figura 3.16 *Rasteritzat d'extrems i pintat de tires de paret*

y^p s'aconsegueix mitjançant les interpolacions línies dels extrems.

DDA sostre:

$$y_1^{\,P} = y_{0\_sostre}^{\,P} + i \cdot \left(\frac{y_{1\_sostre}^{\,P} - y_{r\_sostre}^{\,P}}{x_1^{\,P} - x_0^{\,P}} \right)$$

DDA terra:

$$y_2^{\,P} = y_{0\_terra}^{\,P} + i \cdot \left(\frac{y_{1\_terra}^{\,P} - y_{r\_terra}^{\,P}}{x_1^{\,P} - x_0^{\,P}} \right)$$

$$i \in [x_0^{\,P} \ldots x_1^{\,P})$$

A l'algorisme de renderització de part de sector es precomputaria dos pendents, i durant el rasteritzat només caldria actualitzar les cotes com mostra al següent algorisme.

```
Accio RenderitzarPartSector(entrada Parametres)
Var
        Y_sostre, IncYdx_sostre:real
        Y_ terra,  IncYdx_terra: real
        y1, y2: enter
Fvar
          (...)

          IncYdx_sostre ← (Parametres.y1_sostre – Parametres.y0_sostre)/(Parametres.x1 – Parametres.x0)
          IncYdx_terra  ← (Parametres.y1_terra  – Parametres.y0_terra) / (Parametres.x1 – Parametres.x0)
          Y_sostre      ← Parametres.y0_sostre
          Y_terra       ← Parametres.y0_terra

          (...)

          Per x des de Parametres.x0 a Parametres.x1 fer
                    (...)

                     y1 ← enter(Y_sostre)
                     y2 ← enter(Y_terra)

                    PintarTira(y1,y2, ...)

                   { S'actualitza dda de la Parametres }

                    Y_sostre← Y_sostre + incYdx_sostre
                    Y_terra  ← Y_terra  + incYdx_terra
          Fper
FAcció
```

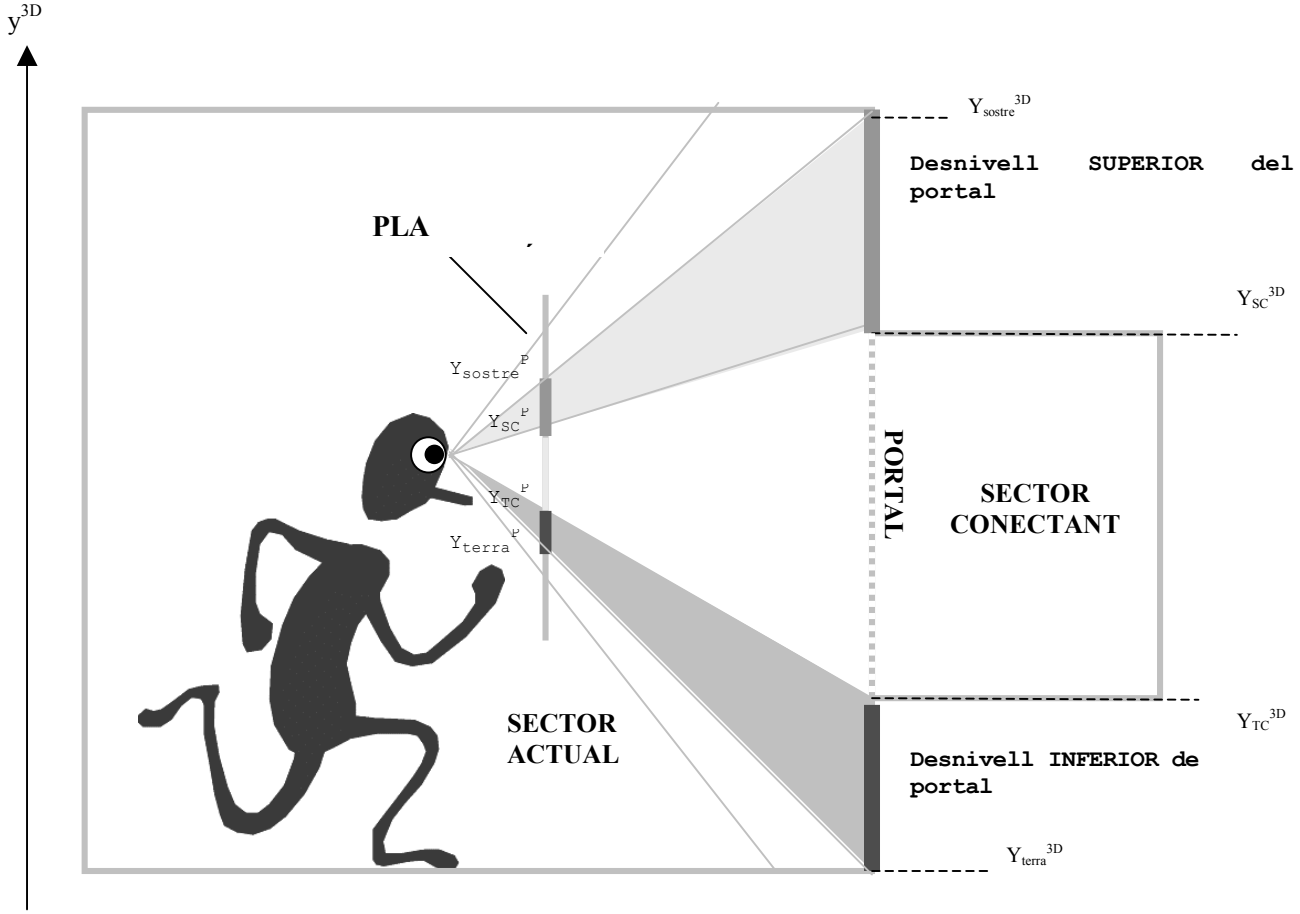
Llistat 3.3

I és així com aconseguim la *renderització* de paret. Cal observar que no hi ha cap càlcul de perspectiva per projectar les tires de paret, el rasteritzat utilitzat dona les cotes *y* de pantalla adequades per cada columna.

3.3.3.2 Renderització de desnivells

Quan una línia de sector representi un portal, es possible que presenti un desnivell de terra-sostre del sector actual respecte el terra-sostre del sector connectant. La renderització de desnivells és molt semblant el efectuat per les parets. En vés de traçar una tira de paret per iteració en el bucle de pintat, se'n traçaran dues: la tira que representa el desnivell inferior i la tira que representa el desnivell superior, sempre i quan les diferencies de altura entre sector actual i connectant ho facin possible.

La *renderització* del desnivell superior serà possible si i només si el sostre adjacent està per sota de l'actual i la *renderització* del desnivell inferior (tira inferior de portal) serà possible si i només si el terra adjacent està per sobre l'actual. A la figura 3.17, podem veure l'exemple de la projecció d'ambdós desnivells.



On:

Y_{SC}^{3D} = Altura terra sector connectant
Y_{TC}^{3D} = Altura terra sector connectant

Figura 3.17 *Exemple de les projecció dels dos desnivells (superior i inferior) del portal.*

Per explicar el seu funcionament, partirem també d'un exemple pràctic. Fent referència a la de la figura 3.17, si la línia a *renderitzar* fos un portal amb desnivells inferior-superior visualitzables, el procés pel seu *renderitzat* seria el següent:

1. Fent referència a la figura 3.18, es projecta els nivells d'altura Y_{sostre}^{3D}, Y_{terra}^{3D}, Y_{SC}^{3D} i Y_{TC}^{3D}.

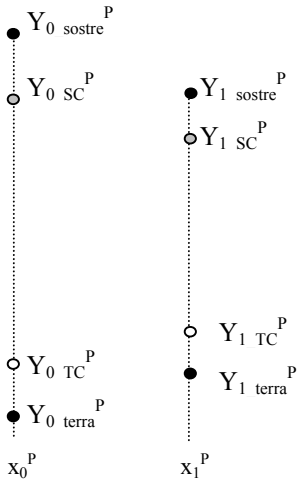
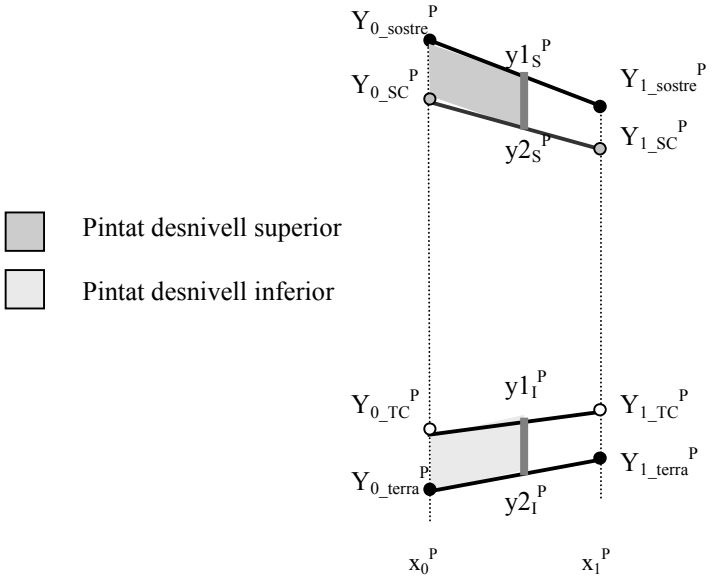


Figura 3.18

2. Tenint tota la informació dels punts anteriorment projectats pintarem ambdós nivells amb tant sols 4 DDAs:

   2 DDAs els utilitzarem pel pintat del desnivell superior de portal. Amb el DDA de sostre actual obtindrà les cotes *y* superiors ($y1_S^P$) i el DDA sostre connectant obtindrà les cotes *y* inferiors ($y2_S^P$), pel pintat de tira en cada iteració.

   De manera anàloga, utilitzarem 2 DDAs pel pintat del desnivell inferior de portal. Amb el DDA terra connectant s'obtindran les cotes superiors ($y1_I^P$) i amb el DDA terra actual s'obtindran les cotes inferiors ($y2_I^P$), pel pintat de tira en cada iteració.



DDA sostre actual:

$$y1_S^P = y_{0\_sostre}^P + i \cdot \left(\frac{y_{1\_sostre}^P - y_{r\_sostre}^P}{x_1^P - x_0^P} \right)$$

DDA sostre connectant:

$$y2_S^P = y_{0\_SC}^P + i \cdot \left(\frac{y_{1\_SC}^P - y_{r\_SC}^P}{x_1^P - x_0^P} \right)$$

DDA terra connectant:

$$y1_I^P = y_{0\_TC}^P + i \cdot \left(\frac{y_{1\_TC}^P - y_{r\_TC}^P}{x_1^P - x_0^P} \right)$$

DDA terra actual:

$$y2_I^P = y_{0\_terra}^P + i \cdot \left(\frac{y_{1\_terra}^P - y_{r\_terra}^P}{x_1^P - x_0^P} \right)$$

$i \in [x_0^P \ldots x_1^P)$

Fent referència a l'algorisme del llistat 3.3, s'haurà de modificar pel següent (llistat 3.4). En aquest cas, s'ha afegit codi necessari per pintar les tires inferiors/superiors en cas que la línia sigui portal.

```
Accio RenderitzarPartSector(entrada Parametres, LiniaSector)
Var
        Y_sostre, Y_sc, Inc_Ydx_sostre, IncYdx_terra: real
        Y_terra, Y_tc, Inc_Ydx_tc, Inc_Ydx_sc: real
       y1, y2: enter
FVar
        { Es precalcular els pendents pertinents al sector connectant –sc-}
        IncYdx_SC ← (Parametres.y1_sostre_connectant –  Parametres.y0_sostre_connectant)     /  (Parametres.x1 – Parametres.x0)
        IncYdx_TC  ← (Parametres.y1_terar_connectant – Parametres.y0_terra_connectant)     /  (Parametres.x1 – Parametres.x0)

        { Cotes rasterització inicial del sector connectant}
        Y_SC, Y_TC) ← (Parametres.y0_SC, Parametres.y0_TC)

        { Es precalcular els pendents pertinents al sector actual}
        IncYdx_sostre ← (Parametres.y1_sostre – Parametres.y0_sostre)/(Parametres.x1 – Parametres.x0)
        IncYdx_terra  ← (Parametres.y1_terra  – Parametres.y0_terra)/(Parametres.x1 – Parametres.x0)

        { Cota rasterització inicial del sector actual}
        (Y_sostre, Y_terra) ← (Parametres.y0_sostre, Parametres.y0_terra)

        Per x des de Parametres.x0 a Parametres.x1 fer
                si(LiniaSector és portal) llavors { Es renderitzar tires inferiors/susperiors de portal }
                        (...)
                        (y1, y2) ← (enter(y_sostre), enter(y_SC))

                        { Es renderitzar tira susperior... }

                        PintarTira(y1,y2, ...)

                        { Es renderitzar tira inferior... }
                        (y1, y2) ← (enter(y_TC), enter(y_terra))

                        PintarTira(y1,y2, ...)

                altrament { Es renderitza paret }
                        (...)
                        y1 ← enter(y_sostre)
                        y2 ← enter(y_terra)
                        PintarTira(y1,y2, ...)
                FSi

                ( Y_sostre,Y_terra)← (y_sostre + incYdx_sostre, y_terra  + incYdx_terra)
                ( Y_SC,Y_TC)     ← (y_SC + incYdx_SC, y_TC  + inc_Y_dx_TC)
        Fper
FAcció
```

Llistat 3.4

3.3.3.3 Pintat amb textura

Pels pintat de tires de paret amb una textura, cal explicar la manera de trobar les coordenades uv segons la perspectiva. Donat que la visualització de paret són col·leccions de tires pintades per-columna, el procés de lectura de textura serà igualment **per-columna**. Això vol dir que, si haguéssim de pintar les tires de paret a mà, el procés seria simplement seleccionar una columna de textura i estirar-la verticalment fins als extrems de paret projectats.

Primer de tot **caldrà trobar la coordenada u** de textura **la qual serà fixa en tot el pintat (u_a^T a la figura B).** D'altra banda, les **coordenades v estaran associada a les coordenades de tira paret projectades** i per això **caldrà calcular un escalat en v,** un valor constant que ens donarà els successius valors v de textura. Assumim que, pel pintat sempre partirem amb offset inicial v (v_0^T a la figura B) igual a 0 des de l'extrem inferior de projecció de tira ($y1^P$ a la figura B).

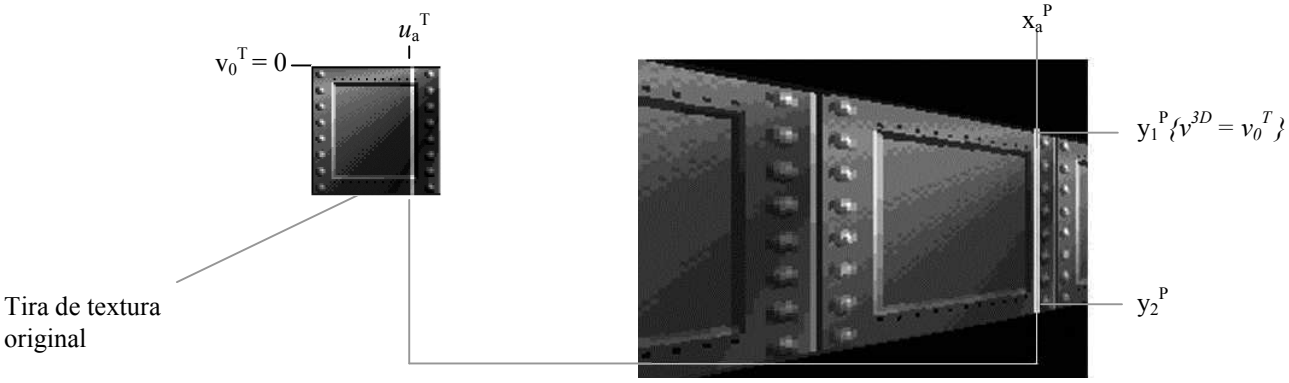


Figura B. *Tira de textura **escalada** verticalment segons la projecció de la tira, que representa la seva distancia.*

Tal com s'ha explicat, el pintat de tira consistirà d'un increment lineal en v^T i fix en u^T, com mostra el llistat de codi 3.10:

```
{Obtenir coordenada u_a^T}
{Obtenir Escalat_v^3D}

v^3D ← v_0^T

Per y des de y1 a y2 fer        {Bucle del pintat de tira-paret amb textura}

        { Selecció de coordenada v mitjançant el mòdul d'alçada }
        v^T ← enter(v^3D) mod alçada_textura

        texel ← textura[v^T][u_a^T]
        Escriure_Pixel(x_a^P,y,texel)

        v^3D ← v^3D + escalat_v^3D
FPer
```

Llistat 3.10

- **Obtenció de u^T**

Associem les coordenades u a la distància de paret respecte el vèrtex 0 de línia que s'està renderitzant.

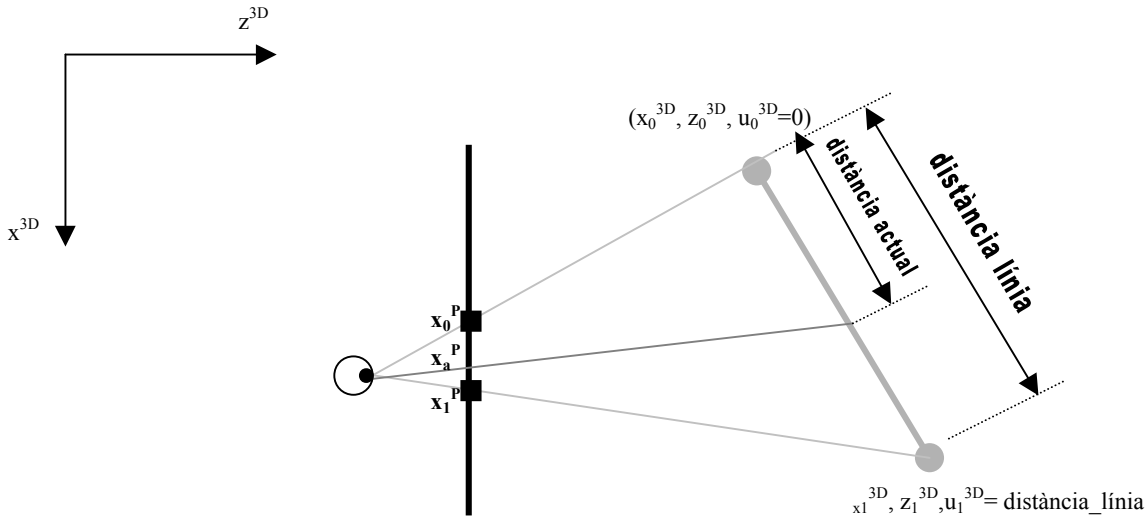


Figura 3.19

Segons la figura 3.19, es disposa *renderitzar* una paret amb pintat de textura. Podem observar a la figura que el vèrtex inicial comença amb una distància inicial de 0 (u_0^{3D}= 0) fins al vèrtex final que té la distància màxima de línia (u_1^{3D}=distància_línia).

A la secció 2.1.4 vam explicar la possibilitat de interpolar linealment amb correcció de perspectiva qualsevol atribut 3D associat als vèrtexs d'un segment de línia. Fent referència a la figura, si convertim els valors u_0^{3D} i u_1^{3D} a l'espai de pantalla i llavors els interpolem amb correcció de perspectiva, podrem calcular la distància 3D a cada columna en $[x_0^P, x_1^P]$.

Segons la figura, podem saber la distància 3D a la columna x_a^P, a partir d'aquesta podrem saber la columna de textura que representa. Per descomptat, una textura no sempre tindrà una amplada igual a la distància total de línia, s'ha de seleccionar la columna aplicant el seu mòdul respecte la seva amplada, es a dir:

$$u_a^T \leftarrow \text{Distància\_Actual}_a^{3D} \textit{ mod } \text{Amplada\_Textura}$$

- **Càlcul de Escalat**

L'escalat explica quina variació vertical té la tira de paret a l'espai d'ull en cada increment unitari del pla projecció. Fent referència a següent figura.

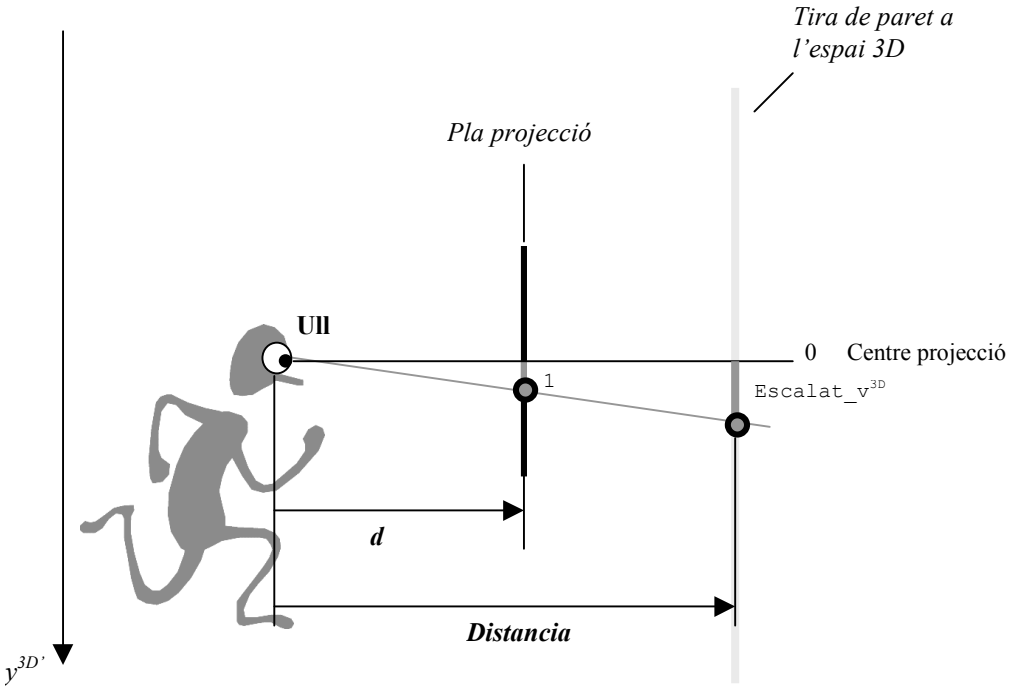


Figura 3.20

Emprant teoria de similitud de triangles,

$$\frac{1}{d} = \frac{Escalat\_v^{3D}}{Distancia} \qquad \textit{Equació 3.10}$$

Aïllem la variable Escalat i ja l'hem trobat.

$$Escalat\_v^{3D} = \frac{Distancia}{d}$$

3.3.3.4 MipMapping

Quan les parets es pinten amb una escala molt gran, fet que succeeix quan estan lluny de l'observador, hi pot haver problemes d'aliasing com el que podem observar a la següent figura.

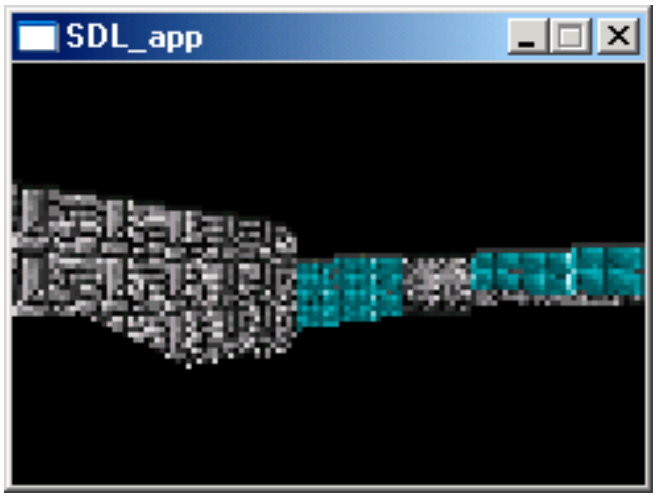


figura 3.21

El problema es deu quan l'escala supera les dimensions de la imatge les coordenades UV comencen a ser aleatories i, per aquest motiu, es presenta l'efecte de aliasing durant el mapeig. Si més no podem observar un exemple.

Exemple.

Suposem que d=120 i volem mapejar una tira de paret que es troba a una distància de 7800 unitats. Llavors la escala seria de 7800/120 = 65. Si volguéssim mapejar una paret amb una textura de 64x64 texels amb $v_0^T = 0$, les coordenades succesives,

$V_0^{3D} = (\ v_0^T\ )$ **mod** 64 = (0) **mod** 64 = 0
$V_1^{3D} = (V_0^{3D} + 65)$ **mod** 64 = (0 + 65) **mod** 64 = 1
$V_2^{3D} = (V_1^{3D} + 65)$ **mod** 64 = (1 + 65) **mod** 64 = 2
$V_3^{3D} = (V_2^{3D} + 65)$ **mod** 64 = (2 + 65) **mod** 64 = 3

...

Es a dir, té el mapeig efectiu de escala 1, impropi per una distancia de 7800 unitats.

O, si l'observador es troba a una distància de 7560 unitats de la paret, la escala seria de 7560 /120 = 63. Les coordenades successives de v seria,

$V_0^{3D} = (\ v_0^T\ )$ **mod** 64 = (0) **mod** 64 = 0
$V_1^{3D} = (V_0^{3D} + 63)$ **mod** 64 = (0 + 63) **mod** 64 = 63
$V_2^{3D} = (V_1^{3D} + 63)$ **mod** 64 = (63 + 63) **mod** 64 = 62
$V_3^{3D} = (V_2^{3D} + 63)$ **mod** 64 = (62 + 63) **mod** 64 = 61

...

Es a dir, sortiria el mapeig de tira de textura invertida.

Els problemes del exemple anterior sempre passaran quan l'escala obtinguda sigui superior a 1, la qual cosa vol dir que s'ignora part d'informació de l'espai de imatge i per consegüent perdem la seva qualitat de representació (decreix el nivell de qualitat).

Existeix una solució aplicant la tècnica mipmapping (veure annex 6 per més informació). Només cal crear una tira de mipmaps per cada textura. Cada mipmap representa la derivació de la imatge anterior reduïda a la meitat i amb tècniques d'interpolació per aconseguir la màxima qualitat (figura 3.22), fins arribar a la màxima simplificació de la imatge. De la llista de mipmaps creada, seleccionariem un d'aquests segons el nivell de qualitat del mapeig

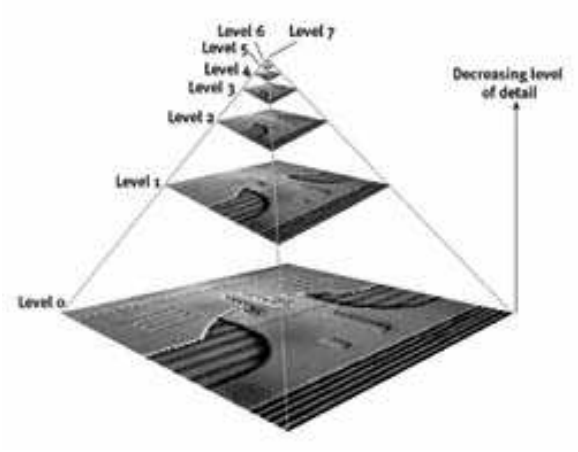


Figura 3.22

Per exemple, per una textura de 64x64 tindríem els mipmaps de 32x32, 16x16, 8x8, 4x4, 2x2 i 1x1, és a dir que, un total de 6 mipmaps acompanyarien aquesta textura.

La qualitat de mapeig, com ja s'ha dit ho determina l'escala, per això, en funció d'aquest factor, es seleccionarà el mipmap com,

Si escala < 1 → Seleccionem MipMap 0 de 64x64 *{Màxim qualitatl}*
Si $1 \leq$ escala < 2 → Seleccionem MipMap 1 de 32x32
Si $2 \leq$ escala < 4 → Seleccionem MipMap 2 de 16x16
Si $4 \leq$ escala < 8 → Seleccionem MipMap 3 de 8x8
Si $8 \leq$ escala < 16 → Seleccionem MipMap 4 de 4x4
Si $32 \leq$ escala < 32 → Seleccionem MipMap 5 de 2x2
Si escala > 32 → Seleccionem MipMap 6 de 1x1 *{Mínima qualitat}*

Donat que, podem aplicar textures més petites a la original, l'escala i les components inicials UV han de ser modificats segons el mipmap escollit per fer el mapeig correcte, amb la següent formula.

$Escala \leftarrow Escala/(2^m)$
$U_0 \leftarrow U_0/(2^m)$
$V_0 \leftarrow V_0/(2^m)$

On *m* es el numero de mipmap seleccionat.

Aplicant la tècnica de mipmaps en l'escenari renderitzat anterior, observem que la imatge aconseguim una millora en la seva qualitat d'imatge.

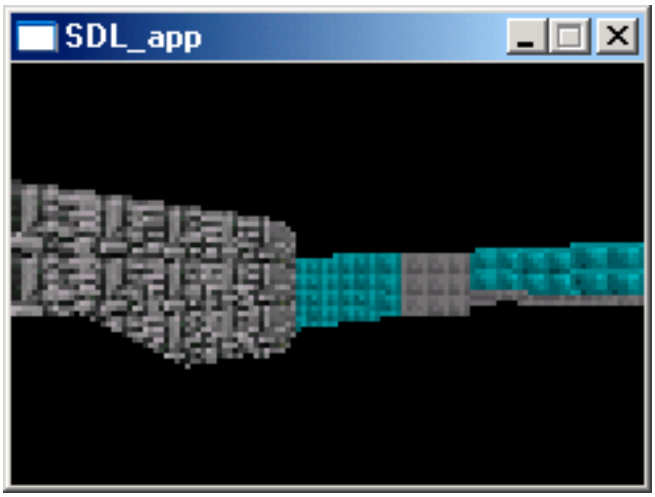


figura 3.23

3.3.4 Renderització dels paisatges

Per la *renderització* de paisatges, utilitzarem imatges de 360º (figura 3.26), anomenats també panorames, ja que són molt adequats per propòsits de fons en motors 3D.

Figura 3.26 *Exemple d'una imatge 360º (Montevideo –Uruguay-)*

Donat que la *renderització* de parets es fa **per-columna**, la *renderització* del paisatge també es farà per-columna, normalment, al marge superior de la paret (figura 3.27).

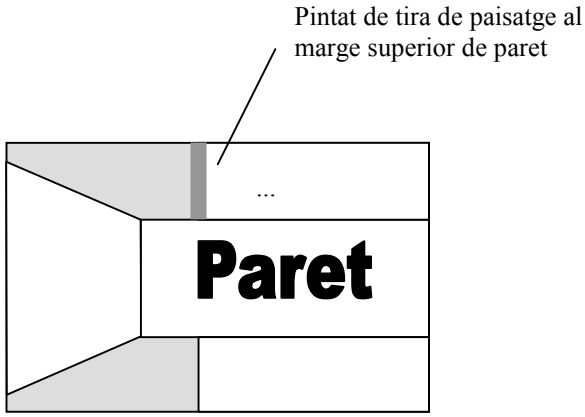


Figura 3.27

El procés de renderització de paisatge, simplement consistirà en escollir la columna de paisatge i pintar-la amb l'escala adequada. Abans, però, cal saber el numero de columnes (o amplada) cal que tingui la imatge-360, perquè es vegi amb un aspecte correcte.

- **Amplada de la imatge**

En funció del FOV i la resolució x de pantalla (*x_res*) podrem saber quina amplada de imatge serà l'adequada per la seva *renderització*.

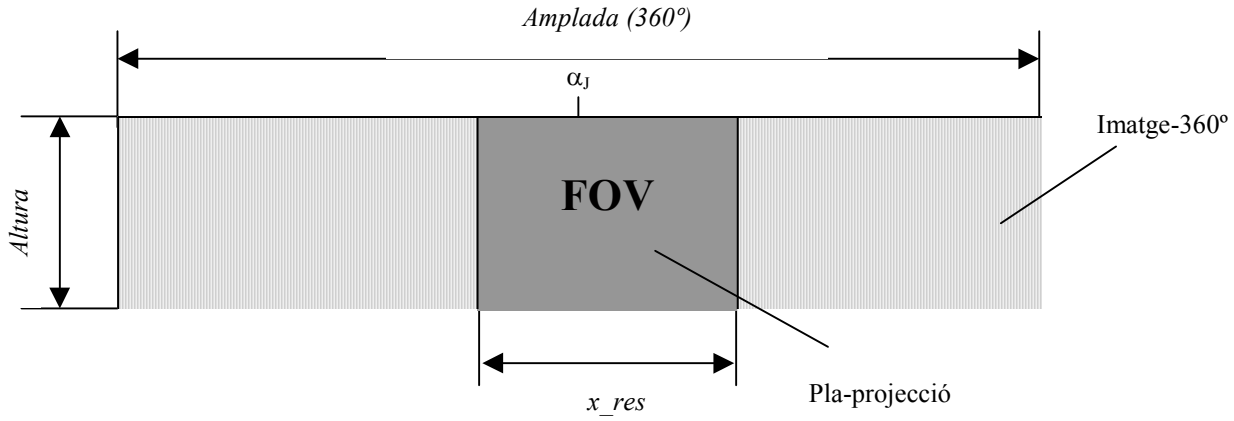


Figura 3.28

Segons la figura 3.28, si sabem que l'amplada de la imatge representa els 360º i el FOV establert compren tota la resolució x, llavors per trobar l'amplada la imatge adequada s'ha d'aplicar una senzilla regla de tres:

$$\texttt{Amplada\_Paisatge} = \frac{360 \cdot \texttt{x\_res}}{\texttt{FOV}}$$

Exemple.

Si disposem d'un FOV 90º, la resolució x de pantalla és de 240 llavors la amplada adequada del

paisatge seria de,

$$\texttt{Amplada\_Paisatge} = \frac{360 \cdot 240}{90} = 960\,\texttt{texels}$$

- **Altura de la imatge**

Perquè la imatge es vegi de manera adequada, la altura de imatge haurà de tenir les mateixes dimensions que la resolució *y* de la pantalla virtual. Recordem, però que, a la secció 3.2.1.3 es va dir que per donar l'efecte de mirar amunt/avall en el jugador, es donaria uns marges extres de resolució virtual de *+y_res/2* píxels, segons l'exemple de la figura 3.29. Segons l'exemple de la figura 3.29, **caldria una altura de paisatge de fins al doble de resolució *y*** que te la pantalla efectiva (*2*y_res*) per donar la possibilitat d'observar les zones superiors inferiors de paisatge i, així, donar la sensació del moviment de pitch del jugador.

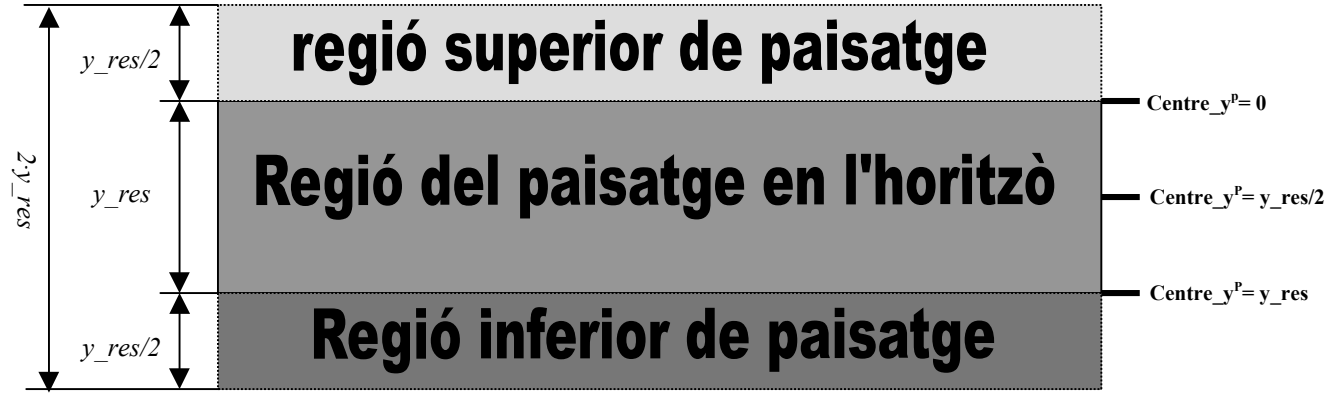


Figura 3.29

• Obtenció de la columna de paisatge

Per la renderització cal saber la manera d'obtenir la columna de paisatge segons la columna x_N que s'està *renderitzant*. Per obtenir-la, primer cal saber la columna de paisatge que apunta la direcció de jugador (α_J) (figura 3.28). Per fer-ho cal saber primer de tot**, l'angle per columna que tenim segons el FOV establert en la resolució *x* de pantalla (x_res)**.

$$\mathtt{Angle\_Per\_Columna} = \frac{\mathtt{FOV}}{\mathtt{x\_res}}$$

$$u_{centre} = \frac{\alpha_j}{\mathtt{Angle\_per\_columna}}$$

$$\alpha_J \in [0,1,...,360)$$

Un cop sabem l'angle per columna, podem trobar la columna que apunta l'angle de direcció del jugador (α_J).

I per últim un simple offset x_N respecte el centre de projecció (x_res/2) sabrem la columna de paisatge que cal utilitzar. Caldrà un mòdul igual a l'amplada de la imatge per evitar sortir del bufer.

$$u_{PAISATGE} = \left(u_{centre} + \left(x_N - \frac{\mathtt{x\_res}}{2}\right)\right) \bmod \mathtt{Amplada\_Paisatge}$$

3.3.5 Renderització del sostre i terra.

La renderització del terra i sostre es farà per-columna i la renderització tindrà efecte al marge superior/inferior de paret al retallat.

Pel mapeig de sostre i terra assumim, temporalment, que:

- Son sempre plans (no hi ha pendents).
- Les coordenades UV de textura estaran associades a les coordenades XZ de mapa.

Per trobar la intersecció XZ al pla, donades les coordenades x^p, y^p actuals seguirà els passos següents:

1. A l'espai YZ d'ull, llençar el raig per la fila y^p actual i trobar la distància de la tira de la intersecció el raig-pla, on totes les columnes del scanline que apunta y^p hi intersecten. A la vegada, aquesta distància representarà la intersecció $zi^{3D'}$.
2. Un cop trobada la distància ($zi^{3D'}$), podrem trobar la intersecció $xi^{3D'}$.
3. Per últim, caldrà portar la intersecció ($xi^{3D'}$,$zi^{3D'}$) a l'espai de món.

Ho expliquem amb més detall aquests passos a les següents seccions.

Intersecció $Zi^{3D'}$

Fent referència a la següent figura, on podem observar l'ull llançant el raig (línia discontinua) fins intersecar al pla:

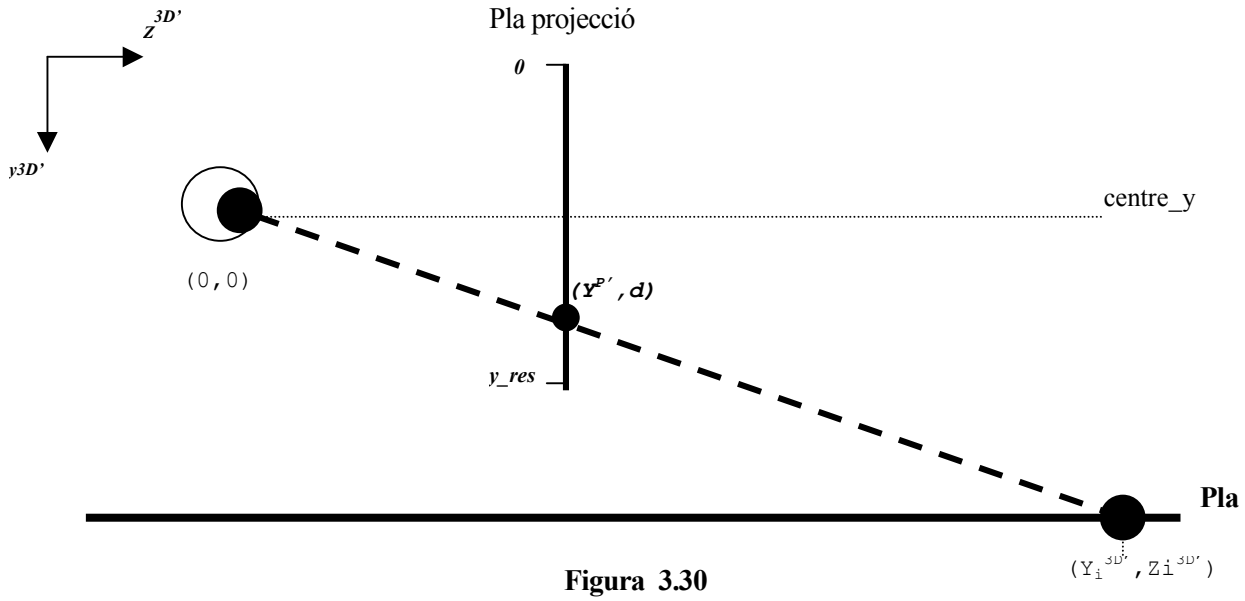


Figura 3.30

Emprant similitud de triangles segons la fig. 3.30, s'arriba a la següent equació:

$$\frac{Y^{P'}}{d} = \frac{Y_i^{3D'}}{Z_i^{3D'}}$$

Aïllant,

$$Z_i^{3D'} = d \cdot \frac{Y_i^{3D'}}{Y^{P'}}$$

Yp' es la fila actual respecte l'horitzó o centre de projecció Y (y_res/2). $Y_i^{3D'}$és la distància del pla respecte l'ull, la qual es coneguda. Per tant trobem la coordenada $Zi^{3D'}$de la intersecció raig pla.

$$Z_i^{3D'} = d \cdot \frac{Y_U^{3D} - Y_{Pla}^{3D}}{Y^P - centre\_y} \qquad \textit{Equació 3.11}$$

Intersecció $Xi^{3D'}$

Un cop trobada la intersecció $Zi^{3D'}$, aquesta representa la distància en que es troba la línia perpendicular en $Z^{3D'}$on **totes les columnes del scanline y^p hi intersecten**. Observant la següent figura:

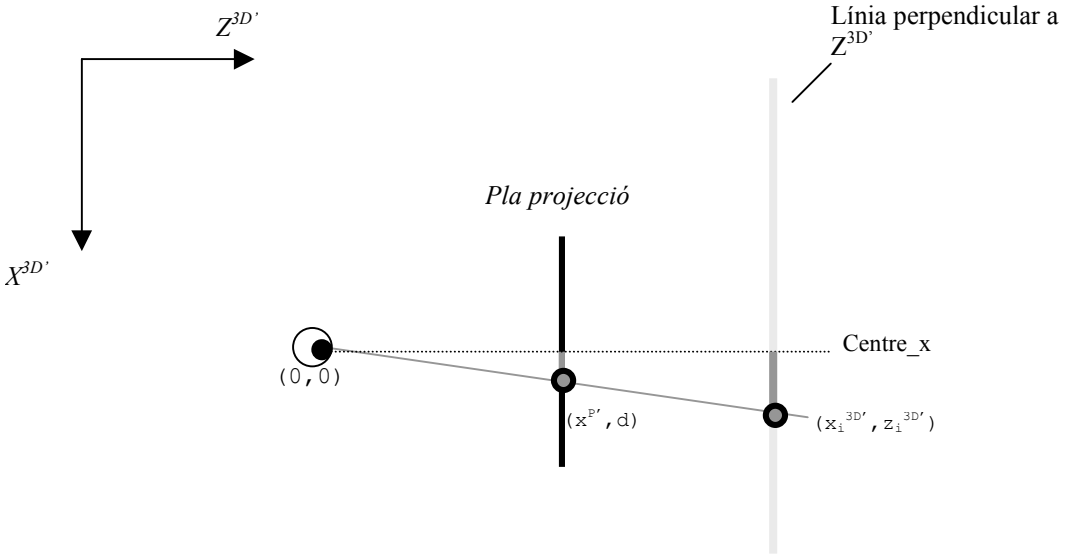


Figura 3.31

El problema és similar al anterior cas, a diferencia, la intersecció es troba a l'espai XZ de l'observador. Segons la figura 3.31, la equació per saber $X_i^{3D'}$ serà doncs,

$$X_i^{3D'} = \frac{Z_i^{3D'} \cdot (X^P - centre\_x)}{d} \qquad \textit{Equació 3.12}$$

Portar la intersecció a l'espai de món

Finalment, caldrà transformar la intersecció trobada al pla a l'espai de món, que no es res més que realitzar l'operació inversa explicada a la secció 2.1.3 (apartat 1), es a dir, aplicar una rotació en Y de α_J.

$$\begin{pmatrix} X_i^{3D} \\ Z_i^{3D} \end{pmatrix} = Rot(EixY, \alpha_J) \cdot \begin{pmatrix} X_i^{3D'} \\ Z_i^{3D'} \end{pmatrix} = \begin{pmatrix} \cos(\alpha_J) & -\sin(\alpha_J) \\ \sin(\alpha_J) & \cos(\alpha_J) \end{pmatrix} \cdot \begin{pmatrix} X_i^{3D'} \\ Z_i^{3D'} \end{pmatrix} = \begin{pmatrix} X_i^{3D'} \cdot \cos(\alpha_J) & -Z_i^{3D'} \cdot \sin(\alpha_J) \\ X_i^{3D'} \cdot \sin(\alpha_J) & Z_i^{3D'} \cdot \cos(\alpha_J) \end{pmatrix}$$

I com que s'havia dit anteriorment que les coordenades de textura estarien associades a les coordenades XZ de món, la intersecció X_i^{3D}, Z_i^{3D} representarien les coordenades UV de textura.

L'algorisme

Havent explicat els passos per renderitzar un pla de sostre i terra, mostrem l'algorisme al següent llistat de codi.

```
Accio RenderitzarPla(entrada x_actual,y1,y2,AlturaPla:enter; Jugador:tJugador;Textura:^car)

    Var
    u,v,y_actual: enter
    Zi3D_ull, Xi3D_ull, Xi3D_Mon,Zi3D_Mon:real
    Fvar

    Per y_actual des de y1 a y2 fer

        { Pas1. Zi3D' }
        zi3d_ull ← d*(Jugador.Y-AlturaPla)/(Y_actual-Centre_Y)

        { Pas2. Xi3D' }
        xi3d_ull ← d*(X_actual-Centre_X)/( zi3d_ull)

        { Pas3. Xi3D, Zi3D }
        xi3D_mon ← xi3d_ull*cos(Jugador.Angle)-zi3D*sin(Jugador.Angle)
        zi3D_mon ← xi3d_ull*sin(Jugador.Angle)+zi3D*cos(Jugador.Angle)

        u ← enter(xi3D_mon)
        v ← enter(zi3D_mon)

        { Pintar pixel }

        Escriure_Pixel(x_actual, y_actual, Textura[v][u])

    Fper

FRenderitzarPla
```

Llistat 3.3

3.4 Preprocessat de l'escena

Ja hem vist a la secció 3.3, que la base del motor es recursiu. També, sabem que la renderització del sector al que apunta el portal ve limitada pel retallat resultant del sector adjacent (o sector anterior) i, per això**, tot sector depèn del sector anteriorment renderitzat**.

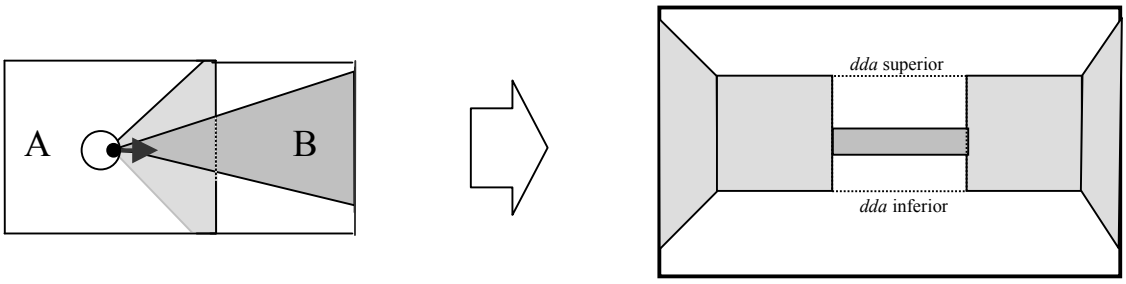


Figura 3.32

A la figura 3.32 hi tenim un exemple molt senzill. Per renderitzar el sector B cal renderitzar abans el sector A. El sector B quedarà limitat al retallat definit pel DDA inferior/superior del portal del A.

Així mateix, afirmem que si cada sector fos apuntat únicament per un sol portal en tot el mapa, com el cas de la figura 5.4, el cost de renderització seria de O(n).

Malauradament, en els nostres mapes no sempre es tindrà configurat el mapa com el de la figura 3.32.

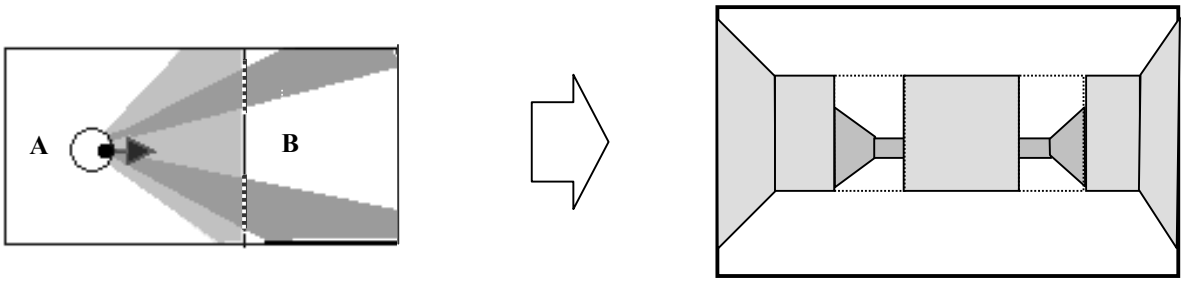


Figura 3.33

Podem observar a la figura 3.33 que, el sector B s'ha renderitzat parcialment 2 cops. Per consegüent, a la següent renderització parcial del sector B, s'ha de tornar a calcular de nou L.I, les interpolacions, etc. Cal tornar a calcular com si d'un nou sector es tractés.

Però, la gravetat del cost no arriba fins aquí. Observem que passa amb la renderització d'un sector transparent (sector configurat per només portals), el sector C de la figura 3.34.

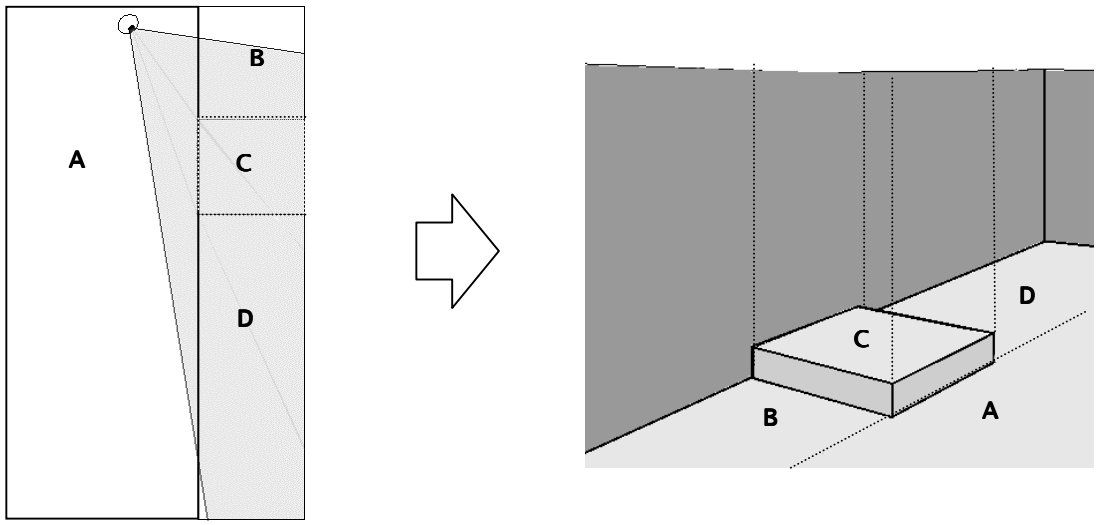
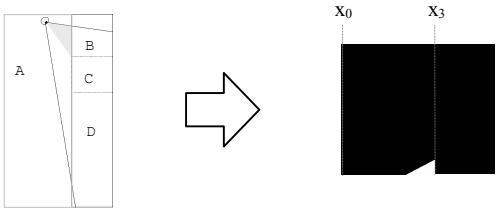


Figura 3.34

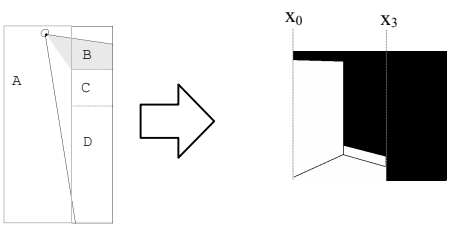
Veiem el curs de la renderització que s'ha seguit per arribar a renderitzar aquest tros d'escena.

A)



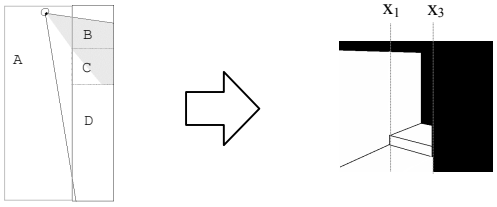
Es començaria renderitzant la primera part de sector A, entre x_0 i x_3.

B)



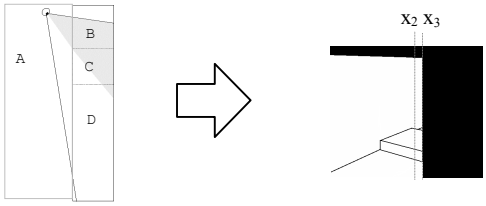
Llavors es renderitza recursivament el sector B, a través del portal del sector A entre x_0 i x_3.

C)



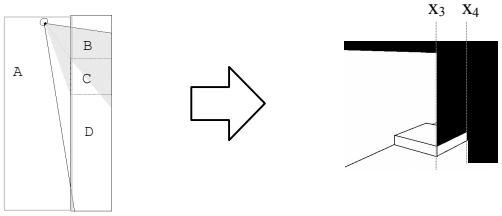
Llavors es renderitza recursivament el sector C, a través del portal del sector B, entre x_1 i x_3.

D)



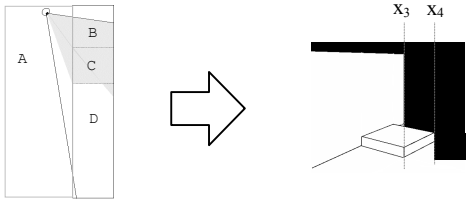
S'acabaria la recursió renderitzant el sector D, entre x_2 i x_3.

E)



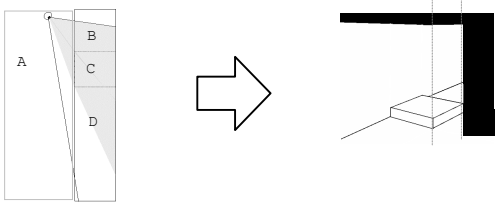
Tornant de la recursió, es continua amb al renderització del sector A, entre x_3 i x_4.

F)



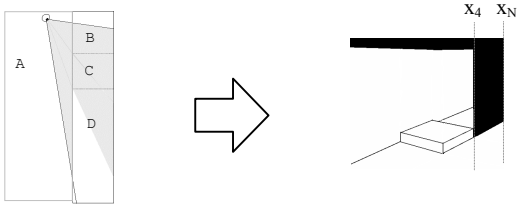
Llavors i, **per 2ª vegada,** es torna a renderitzar de nou el sector C, entre x_3 i x_4.

G)



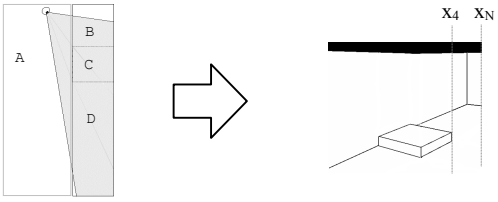
Llavors i, **per 2ª vegada,** es torna a renderitzar de nou el sector D, entre x_3 i x_4.

I)



S'acaba de renderitzar el sector A, entre x_4 i x_N.

J)



Per acabar, es renderitza el sector D **per 3ª cop,** entre x_4 i x_N.

Figura 3.35

A l'exemple de la figura 3.35, el sector B, C i D s'han renderitzat parcialment: 2 cops el sector B, 2 cops el sector C i 3 cops el sector D. Per tant, per 4 sectors que implicava la renderització de FOV, a tingut el cost efectiu de renderitzar 9 sectors.

Cal observar, però, una peculiaritat en aquests tipus de renderitzacions i es que **són connectables**, ja que la columna x inicial d'un sector parcialment a renderitzar, coincideix amb la columna x final del mateix sector parcial renderitzat anteriorment. Per exemple, segons la figura 3.35, podríem **connectar** les columnes que impliquen la renderització del sector C per lo que, més tard, podríem renderitzar-lo **completament entre les columnes x_1 a x_4**.

Però, caldrà conèixer prèviament les columnes que impliquen la renderització de cada sector realitzant amb posterioritat la connexió pertinent (si es possible). Per això, caldrà fer un preprocessat de l'escena, es a dir, guardar els sectors connectats en una llista per, més tard, poder-los renderitzar al complet. Aquest algorisme de preprocessat s'estudia a continuació.

3.4.1 Algorisme de preprocessat

L'algorisme del preprocessat examinarà recursivament els sectors implicats a la renderització, actualitzant columnes finals (x2) si són connectables, altrament es guardarà com si d'un nou sector es tractés. Els sectors que s'han pogut connectar, o no, seran guardats en una llista que, més tard, caldrà processar per renderitzar els sectors implicats d'un sol cop.

La informació bàsica necessària que caldrà guardar en la llista serà la següent:

```
tInformacióLlista TUPLA
                  sector: Identificador del sector.
                  Llista_Sector_Parcials: Llista on s'hi guardaran els sector
                                          parcialment tractats, conectats, o no.
FTUPLA

tInformacióParcial TUPLA
        x1, x2    : Intervals del sector a renderitzar-se totalment
        sector    : Identificador del sector.
FTUPLA
```

Tot seguit anem a descriure l'algorisme de l'acció de preprocessat. Bàsicament, tindrà la mateixa base a la de la funció *RenderitzarSector()* vist en el llistat 3.1b (capítol 3).

```
Acció ObtenirLlistaSectorsTotals(entrada x1,x2, Observador, sector entrada/sortida llistasectors)

        Obtenir_Linia_Interseccio_Raig(Columna,Sector)

        { Busquem sector per si ja estar a la llista...}
        SectorCercat ← CercarSector(sector, llistasectors)

        Si no SectorCercat = NULL llavors {Ja existeix a la llista! Mirem la ultima columna que s'ha renderitzat}
          Si SectorCercat.x2 = x1 llavors {Sector Connectable → Només cal actualitzar ultima columna}
                SectorCercat.x2 ← x2
          Altrament {Sector parcial no conectable → El guardem a la llista de sectors parcials }
                GuardarNouSectorParcial(x1,x2,SectorCercat. Llista_Sector_Parcials)
          Fsi
        Fsi
        Altrament {Nou sector, el guardem a la llista }
              GuardarNouSector(x1, x2, sector, llistasector)
        FSi

      { Processar sector...}
      per cada línia ∈ sector fer

        (x1,x2) ← ProjecciópPantalla(linia)

        Si linia és portal llavors { Es anem obtenint els resultat de llista recursivament}

                ObtenirLlistaSectorsTotals (x1,x2, Jugador, ID_Sector(linia), llistasectors)
         FSi

        Linia ← LiniaSeguent(Linia)
      FPer
Facció
```

Llistat 3.4

Un cop obtinguda la llista, la base de renderitzador caldria substituir-la per el següent codi.

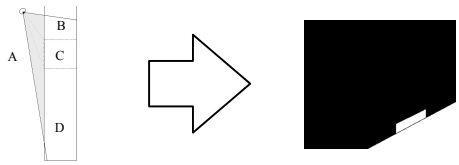
```
Acció Renderitzar(entrada Observador,sector)

        Llista ← ObtenirLlistaSectorsTotals(0, MAX_AMPLADA_PANTALLA-1,observador,sector_observador)

        Mentre no Fi(Llista) fer

                InfoSector ← ElementActual(Llista)
                LlistaParcial ← InfoSector. Llista_Sector_Parcial

                Mentre no Fi(LlistaParcial) fer

                        RenderitzarSector(Llista_Parcial.x1, Llista_Parcial.x1, Observador, Llista_Parcial.sector)
                        SeguentElement(Llista_Sector_Parcials)
                Fmentre

                SeguentElement(Llista)
        Fmentre

FAccio
```

Cal observar que el cor del render ha canviat: **la funció *RenderitzarSector()* ja no és recursiva**. Ara, el procés de renderització el marcarà el recorregut d'aquesta llista, que **renderitzarà els sectors en l'ordre establert**.

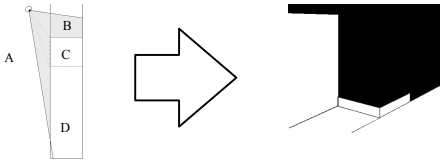
3.4.1.1 Ordre llista de preprocessat

Anteriorment, hem donat tot el procés per aconseguir renderitzar una escena de sectors al complet, gràcies al resultat de connexió de llista construïda. Però, aquesta llista no es pot guardar els sectors en qualsevol ordre, ja que, tal com s'ha dit al començament d'aquesta secció, **tot sector depèn de l'anteriorment renderitzat degut al retallat resultant**.

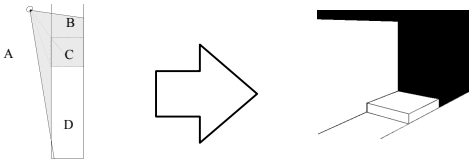
Veiem un exemple, segons la situació de l'escena de la figura 3.36,



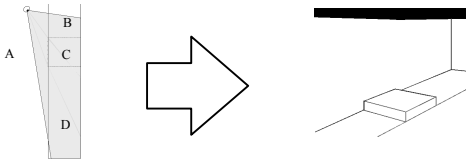
En primer lloc, caldria renderitzar el sector A, que es l'únic que no depèn de ningú. Si més no es el sector inicial de l'observador.



El sector D depèn de C i A. El sector C depèn de A i B. B depèn de A. Com que A ja s'ha renderitzat toca renderitzar B.



Com que C depèn A i B i ambdós sectors s'han renderitzat, renderitzem C.



Per últim, renderitzariem D, ja que tots els seus sectors dependents ja s'han renderitzat.

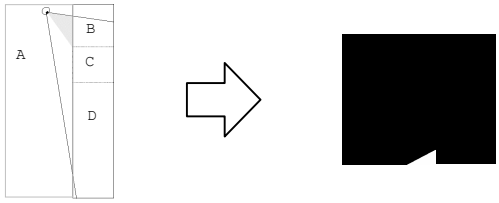
Figura 3.36

Es a dir l'ordre resultat de sectors a renderitzar, segons la figura 3.36 seria:

A, B, C i D

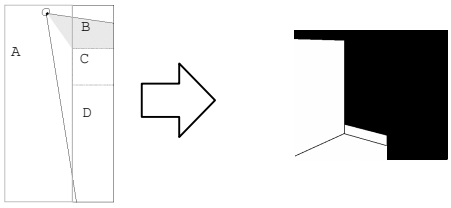
Fem la traça.

A)



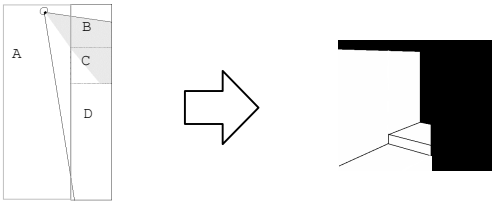
Comencem guardant el sector A.
llista actual : ***A***

B)



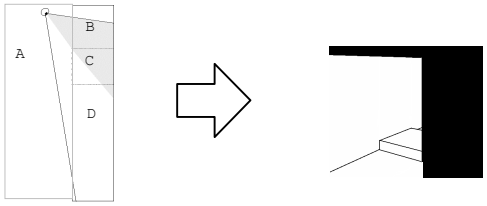
El sector B no el tenim a la llista, el guardem darrera el A, per ser el seu dependent recursiu
llista actual: *A, B*

C)



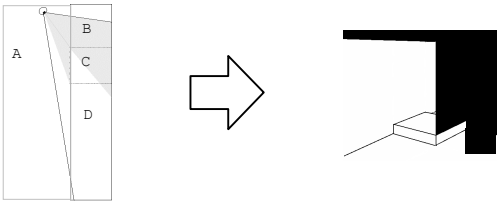
El sector C no el tenim a la llista, el guardem darrera el B, per ser el seu dependent.
llista actual: *A, B, C*

D)



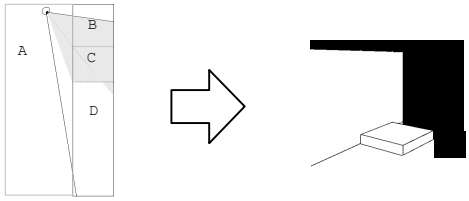
El sector D no el tenim a la llista, el guardem darrera el C, per ser el seu dependent.
llista actual: *A, B, C, D*

E)



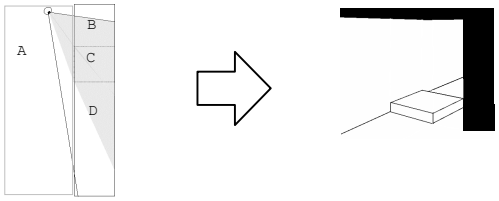
Tornant de la recursió, es continua amb el processat del sector A.
llista actual: *A, B, C, D*

F)



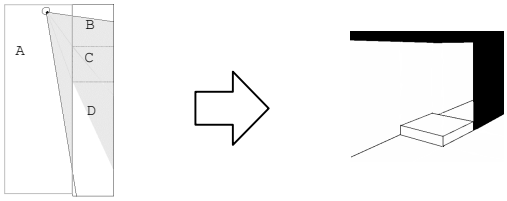
Es processa part del sector C.
llista actual: *A, B, C, D*

G)



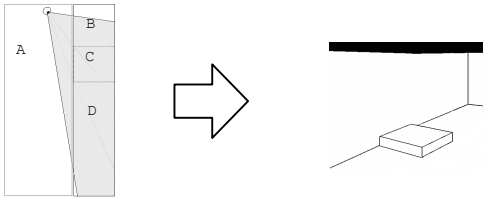
Es processa part del sector D.
llista actual: *A, B, C, D*

I)



Es processa part del sector A.
llista actual: *A, B, C, D*

J)



Es processa part del sector D.
llista actual: *A, B, C, D*

Figura 3.37

Veiem un altre exemple,

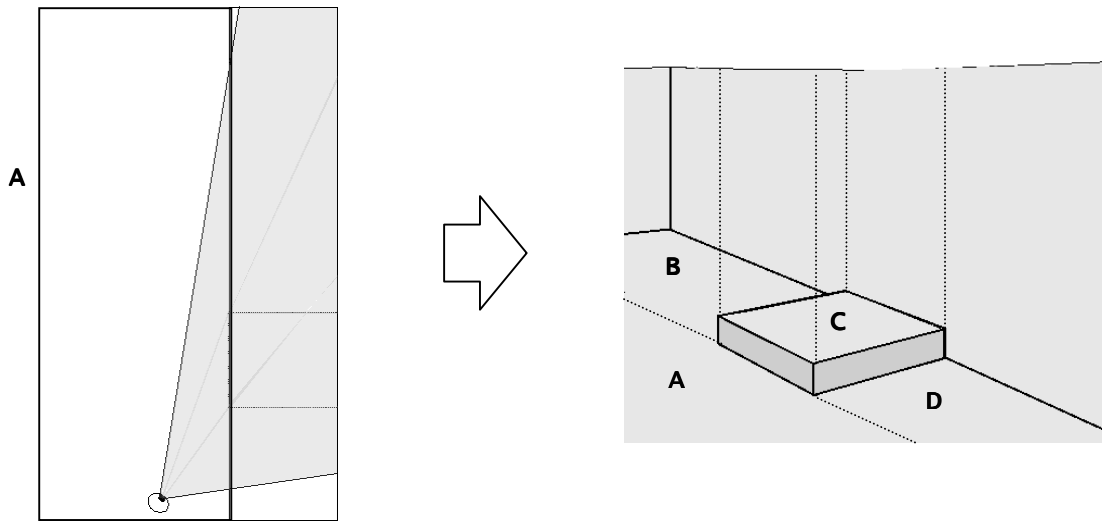
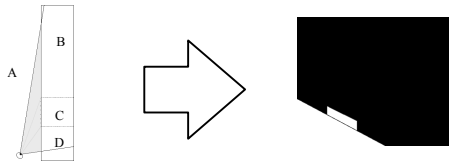


Figura 3.39

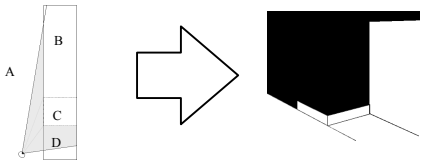
Pel renderitzat de la figura 3.39,

A)



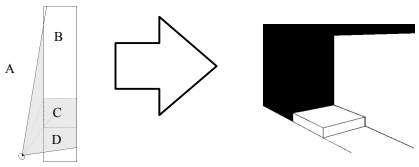
En primer lloc, caldria renderitzar el sector A, que es l'únic que no depèn de cap sector.

B)



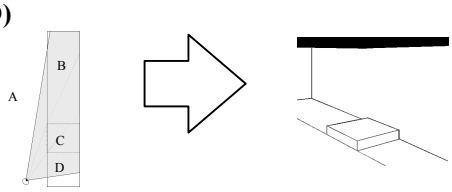
Tot seguit, caldria renderitzar D, ja que l'únic sector dependent és A i que ja s'havia renderitzat.

C)



Com que C depèn A i D i ambdós sectors s'han renderitzat, processem C.

D)



Per últim, renderitzariem B, ja que tots els seus sectors dependents ja s'han renderitzat.

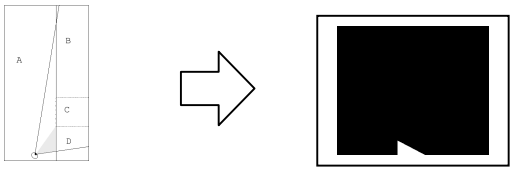
Figura 3.40

En resum, caldria llista els sectors en aquest ordre,

A,D, C i B

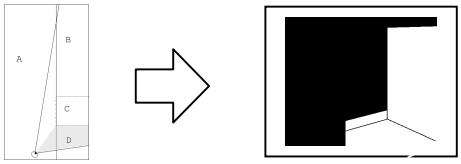
Fem la traça de creació de llista,

A)



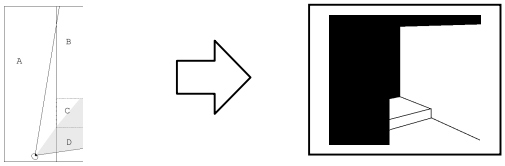
Comencem guardant el sector A.
llista actual : *A*

B)



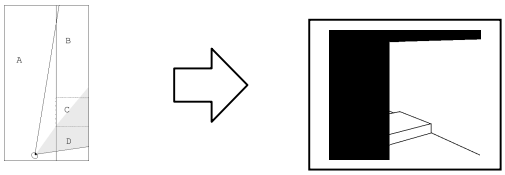
El sector D, no el tenim, el guardem darrera el sector dependent, es a dir darrera el sector A
llista actual: *A, D*

C)



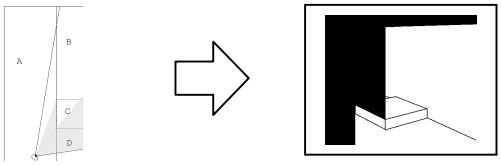
El sector C no el tenim a la llista, el guardem darrera el D, per ser el seu dependent.
llista actual: *A, D, C*

D)



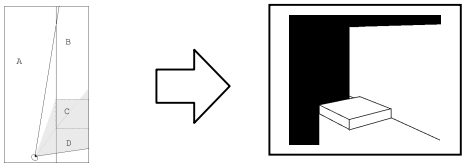
El B no existeix. El guardem darrera del C.
llista actual: *A, D, C, B*

E)



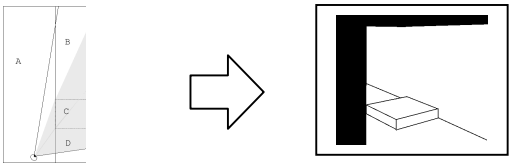
Es processa part del sector A.
llista actual: *A, D, C, B*

F)



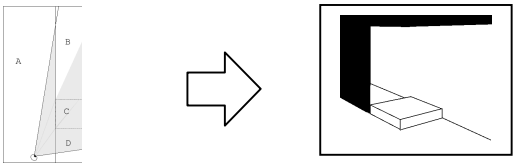
Es processa part del C.
llista actual: *A, D, C, B*

G)



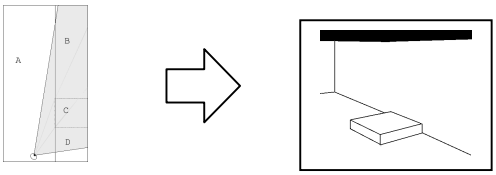
Es processa part del sector B.
llista actual: *A, D, C, B*

I)



Es processa part del A.
llista actual: *A, D, C, B*

J)



Es processa part del sector B.
llista actual: *A, D, C, B*

Figura 3.41

Així doncs, havent fet les traces anteriors sabem que a l'hora de guardar sectors nous, **cal guardar-los darrera el dependent anteriorment guardat (sector dependent)**. Per tant, perquè compleixi l'ordre de processat correcte, cal que tot **nou sector** a inserir es guardi **després del sector dependent**. Veiem tot seguit la modificació necessària respecte el vist en el llistat de codi 3.4:

```
Acció ObtenirLlistaSectorsTotals(entrada x1,x2, Observador, sector, sector_dependent entrada/sortida llistasectors)

        Obtenir_Linia_Interseccio_Raig(Columna,Sector)

        { Busquem sector per si ja estar a la llista...}
        SectorCercat ← CercarSector(sector, llistasectors)

        Si no SectorCercat = NULL llavors {Ja existeix a la llista! Actualitzem cotes finals}
              (...)
        Fsi
        Altrament {Nou sector, el guardem a la llista després del sector dependent}
                  GuardarSectorDespresDeSectorDependent(x1, x2, sector, sector_dependent, llistasector)
        FSi

      { Processar sector...}
      per cada línia ∈ sector fer

        (x1,x2) ← ProjecciópPantalla(linia)

        Si linia és portal llavors { Es anem obtenint els resultat de llista recursivament}

                  ObtenirLlistaSectorsTotals (x1,x2, Jugador, ID_Sector(linia), sector, llistasectors)
         FSi

        Linia ← LiniaSeguent(Linia)
      FPer
Facció
```

Llistat 3.5

Cal observar que, a la funció *ObtenirLlistaSectorsTotals()* se li passa el paràmetre *sector* que passarà a ser el sector dependent del sector de portal.

3.4.2 Optimització del preprocessat

S'ha estudiat la manera òptima de processar escenes amb sectors transparents, però si el preprocessat de l'escena es costosa, no arribarem tampoc a aconseguir la optimització objectiva. En aquí, veurem la manera més ràpida de crear la llista del preprocessat.

- Llista dinàmica, amb suport de índex

Per crear la nostre llista, es segur que caldrà fer-hi moltes insercions i moltes consultes. Si fem ús d'una llista estàtica ordenada, les consultes serien immediates ja que mitjançant el seu l'índex, podem anar a parar directament al node que ens interessa, a demés de poder utilitzar cerques avançades com la *cerca dicotòmica* (eficiència de $O(\log_2)$). Malgrat la seva eficiència en consultes té un desavantatge important a l'hora de fer-hi modificacions; al ser ordenada, cal desplaçar la informació de llista per donar/treure un buit de la llista i inserir/borrar la informació pertinent a la casella seleccionada.

Pel contrari, les llistes dinàmiques és molt ràpides en les insercions, però són costoses en consultes, perquè s'han de fer cerques linealment (eficiència O(n)).

Com que nosaltres volem velocitat a tot cost, farem servir una llista dinàmica amb suport d'índex, un índex que contemplarà tots els sectors enumerats del l'estructura de món.

Per exemple, per la cas preprocessar l'escena de la figura 3.36, es tindria el resultat de llista següent,

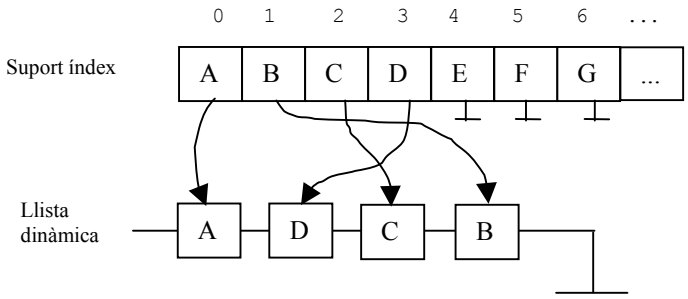


Figura 3.42

Observem a la figura 3.42 que, per fer cerques en un sector és immediat, doncs només s'ha de indexar pel identificador de sector que el defineix, on hi tindrà un apuntador al node de sector de la llista, tan sols un accés (eficiència O(1)). Per fer insercions/borrats, només cal fer reserva/borrat del node i fer les conexions/desconexions pertinents als seus nodes veïns.

- Guardar càlculs durant/després el/del preprocessat de la escena.

Podem amortitzar bastant el cost de preprocessat si, durant aquest, és calculen un sol cop i prou. Per aconseguir-ho s'haurà de extendre la informació de llista i fer-hi les consultes durant tot el preprocessat/processat de la escena. Per exemple, no cal tornar a calcula la L.I quan s'entra a processar un sector per segona vegada consecutiva, si aquest és connectable amb l'anterior. En aquest cas s'aprofitaria l'anterior L.I calculada, en aquest mateix sector.

Igualment, no cal tornar a calcular la transformació de vèrtexs a l'espai d'observador, si anteriorment ja s'havien transformat. Podem tenir una taula amb una longitud total als que disposem a l'estructura de món amb la següent informació.

```
TUPLA tInfoVertex

        IsVisited: boolea
        VertexTransformat:tPunt2D

FTUPLA

InfoVertex: taula[1..MAX_VERTEXS_MON] tipus tInfoVertex
```

I en el codi, a l'hora de calcular un vèrtex a l'espai de observador,

```
Si InfoVertex[VertexActual .id_vertex].IsVisited llavors {L 'Aprofitem}

        VertexTransformat ← InfoVertex[VertexActual.id_vertex]. VertexTransformat

Altrament { Cal calcular-lo i guardar-lo a llista}

        VertexTransformat ← VertexEspaiObservador(VertexActual, Observador)
        InfoVertex[VertexActual .id_vertex].VertexTransformat ← VertexTransformat

        { Notifiquem que s'ha visitat...}
        InfoVertex[VertexActual .id_vertex].IsVisited ← TRUE
FSi
```

Per el cas de aprofitar els càlculs implicats per la projecció de vèrtexs, s'hauria de fer el mateix.

3.5. Tècniques avançades de motor

A la següent secció estudiarem tècniques de ambientació, geomètrica, efectes de portal etc, per tal de donar més realisme al recorregut interactiu del nostre motor.

3.5.1 Il·luminació

Explicarem de quina manera aconseguir que cada sector del mapa disposi d'una il·luminació constant o variable. Més tard, s'estudiarà l'efecte de il·luminació amb profunditat.

3.5.1.1 Il·luminació per-sector

Per aconseguir il·luminar cada sector d'una intensitat de llum, ens caldrà afegir un atribut de intensitat dins la seva estructura. Aquest atribut representarà la intensitat de il·luminació que variarà des de l'interval 0 (mínim) fins al 255 (màxim).

```
{Obtenir coordenada u_a^T}
{Obtenir Escalat_v^3D}

v^3D ← 0

Per y des de y1 a y2 fer        {Bucle del pintat de tira-paret amb textura}

        { Selecció de coordenada v mitjançant el mòdul d'alçada }
        v^T ← enter(v^3D) mod alçada_textura

        texel ← textura[v^T][u_a^T]]

        COLOR_AfegirIntensitat(texel, Sector.Intensitat)

        Escriure_Pixel(x_a^P,y,texel)


        v^3D← v^3D + escalat_v^3D

FPer
```

Llistat 3.6

Al codi del Llistat 3.6 podem observar l'afegit de la funció *COLOR_AfegirIntensitat()*, aquesta s'explica a la secció a6.6 del annex 6. Quan la intensitat de sector (*Sector.Intensitat*) es troba per sota de 128 aquesta funció atenua la intensitat RGB del valor de texel, i el l'amplifica quan es troba per sobre de 128.

Cal mencionar que, al llistat, mostrem l'exemple de pintat amb intensitat per sistemes de vídeo RGB (modes empaquetats).

Si s'utilitzés un sistema de vídeo paleta (8 bits per pixel –8 bpp-), caldria crear una taula de intensitats tal com s'explica a la secció a6.3 del annex 6, substituint la funció *COLOR_AfegirIntensitat(texel,Sector.Intensitat)* de sistema de vídeo RGB, per l'equivalent *COLOR_AfegirIntensitat[texel][Sector.Intensitat]* en 8 bpp.

3.5.1.2 Funcions de il·luminació

Ara el render està preparat perquè cada sector tingui la seva pròpia intensitat de il·luminació. Per fer més atractiu l'escenari, podem afegir funcions de il·luminació, en un interval repetitiu de temps *t* com, per exemple, una simulació d'un parpalleig de flourescent en mal estat (Figura 3.43a) o un efecte de apagat i encès suau, també anomenat *fade-in*/*fade-out* (Figura 3.43b)

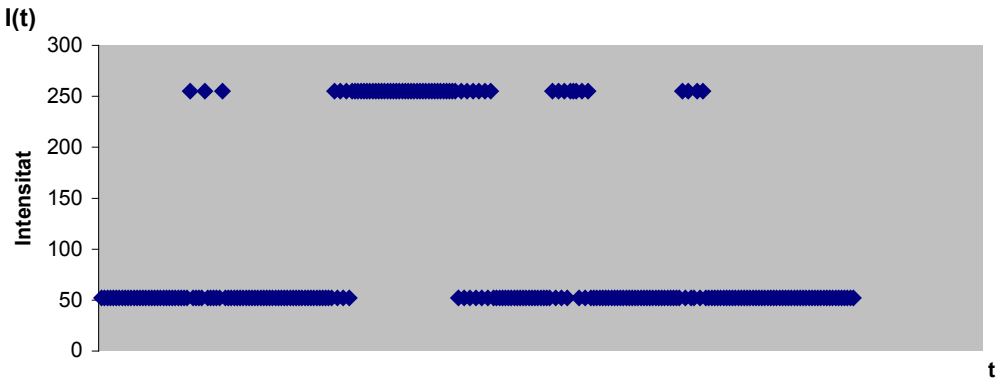


Figura 3.43a

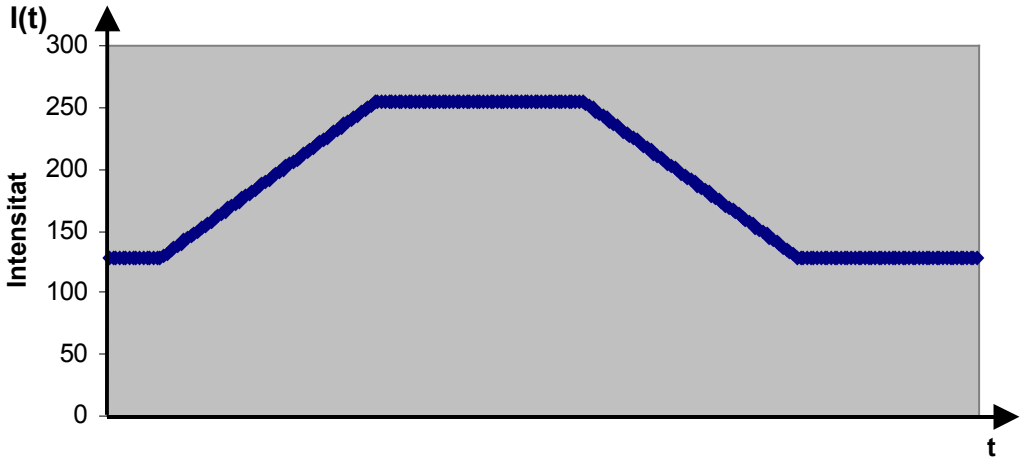
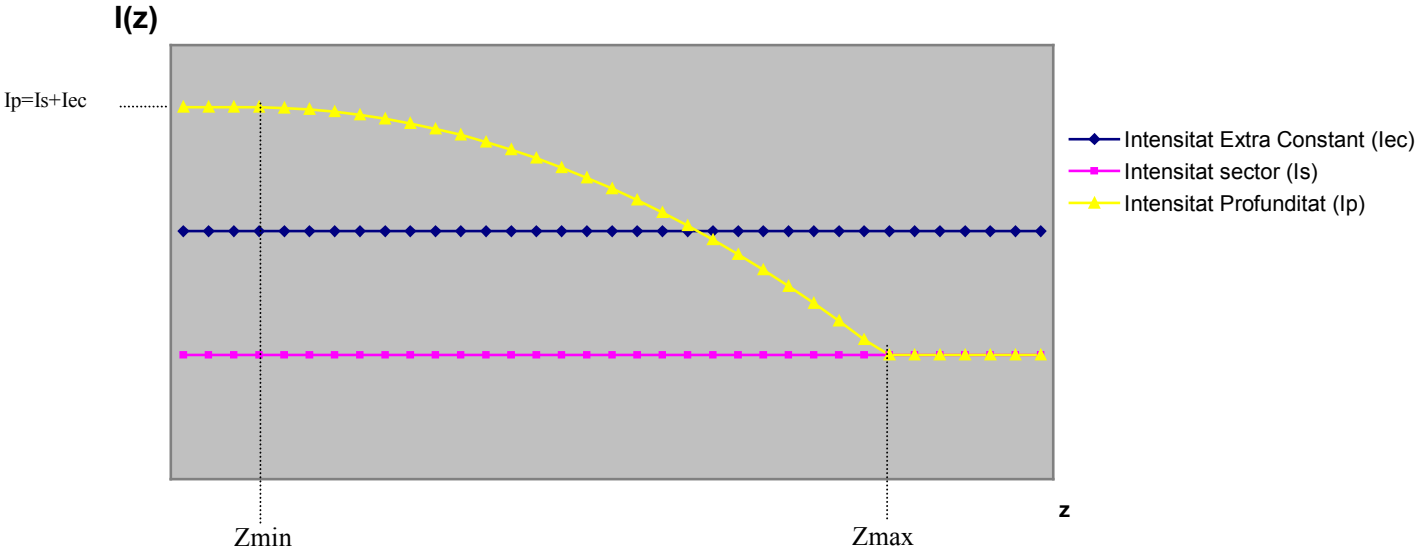


Figura 3.43b

3.5.1.3 Profunditat

Per crear una sensació de profunditat en el recorregut del nostre motor, ho farem a través d'un efecte de il·luminació. Per fer-ho, incorporarem una intensitat extra sobre la intensitat actual del sector que es mantindrà constant fins una distancia mínima (Zmin). A partir de Zmin, la intensitat començarà a decréixer linealment fins arribar a la intensitat de sector.



Gràfica 3.1 *Gràfica representant de la funció de intensitat.*

L'exemple de la Gràfica 3.1 podem observar que la intensitat de profunditat (Ip) parteix de la intensitat sector (Is) més la intensitat extra constant (Iec). A partir de Zmin, la Ip comença a decréixer fins arribar a la intensitat de sector Is.

La funció de profunditat que hem de crear es defineix com:

$$I_p(z) \begin{cases} 0 \leq z < z_{min} & I_s + I_{ec} \\ z_{min} \leq z < z_{max} & (I_s + I_{ec}) \cdot dp \\ \leq z_{max} & I_s \end{cases}$$

On:

Is : Intensitat de sector.
Iec: Intensitat extra afegit
dp: Valor que explicar el pes de decrement en profunditat. Per això cal que el seu valor ∈ [0,1]. per això, a la gràfica 3.1 hi podem observar el decrement intensitat de profunditat hiperbòlic.

A la pràctica, s'ha provat que posant els valors Zmin = 4 i dp = 0'125 donen un resultat correcte.

3.5.2 Efectes de portals

Fins ara hem vist la utilitat dels portals: **renderitzar el sector que hi connecta** (Figura 3.44). No obstant, al nostre render podem incorporar un pintat addicional després d'haver renderitzat el portal, com per exemple pintar el portal amb una textura tipus sprite.

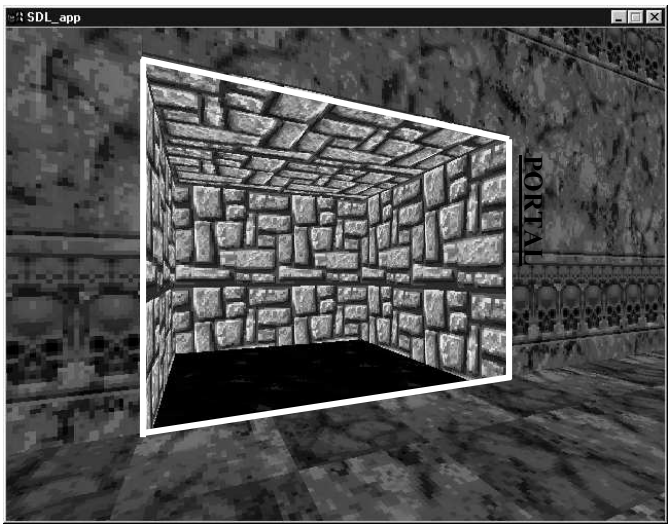
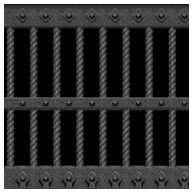


Figura 3.44 *Exemple de portal sense efecte. Només renderitza el sector que connecta.*

Els sprites són un tipus de gràfic que contenen un color especial, tal que no es pinta durant el bucle de pintat. Normalment, aquest color se'l anomena *color de transparència*. Per això, al pintar textures tipus sprite podem veure el portal renderitzat a través d'ells, com el que podem veure a la següent figura.



Textura 128x128 de tipus sprite utilitzada per el mapeig del portal. En aquest, el seu color transparent, en RGB, és el 0 (negre).

Figura 3.45 *Exemple de portal mapejat amb una textura de tipus sprite.*

3.5.3 Objectes

En aquesta secció explicarem com renderitzar objectes. Aquests objectes estaran representats per imatges degudament escalades segons la distància que els separa del jugador (o observador). Mostrem la informació mínima que cal tenir de cada objecte i que caldrà que estigui guardada prèviament en una llista,

```
TUPLA tObjecteMon
    x,y,z: real              { Punt central de l'objecte}
    Id_Imatge: tImatge       { Id. Imatge representativa del objecte}

FTUPLA
```

Un cop disposem d'aquesta informació per-objecte, podrem *renderitzar* cadascun dels objectes i, si es possible pel FOV.

El procés de renderització dels objectes es separa en dues etapes:

Etapa 1. Capturar de objectes al FOV i guardar-los en una llista. Aquesta etapa sempre es farà un cop acabat de renderitzar el sector. Per complir l'etapa cal aconseguir el següent:

Per i **desde** 0 **a** MAX_OBJECTES_MON] **fer**
Si $ObjecteMon_i \in$ Sector_actual **Llavors**

- *Projectar quadrat imatge del objecte.*

- *Si el quadrat imatge intersecta el quadrat de pantalla, llavors es renderitzable. Es guardarà la informació necessària per el posterior renderitzat de l'objecte.*

FSi
FPer

Etapa 2. Pintar la imatge del objecte

Un cop acabat de renderitzar tot l'escenari 3D, es procedirà en pintar cada objecte de la llista per-columna.

- **Projecció del quadrat de la imatge**

Senzillament, s'ha d'aconseguir projectar els punts que defineixen del quadrat a l'espai 3D. Observant la figura següent,

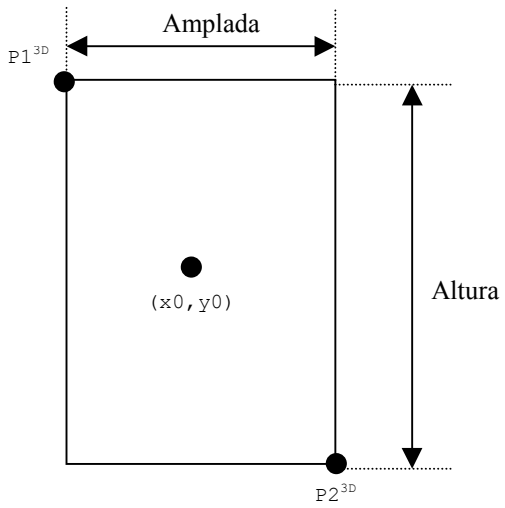


Figura 3.48

P1 i P2, de la Figura 3.48, que formen el quadrat de la imatge a l'espai 3D on els seus valors són:

$$P1^{3D} = (x0-amplada/2,\ y0-altura/2)$$
$$P2^{3D} = (x0+amplada/2,\ y0+altura/2)$$

S'hauran de transformar a l'espai d'ull,

$$P1^{3D'} = \left(\left(Rot(EixY,\ -\alpha_u)\cdot\begin{pmatrix} P1_X^{3D} - U_X^{3D} \\ P1_Z^{3D} - U_Z^{3D}\end{pmatrix}\right),\ \left(U_Y^{3D} - P_Y^{3D}\right)\right) \qquad \textbf{Equació 3.13}$$

$$P2^{3D'} = \left(\left(Rot(EixY,\ -\alpha_u)\cdot\begin{pmatrix} P2_X^{3D} - U_X^{3D} \\ P2_Z^{3D} - U_Z^{3D}\end{pmatrix}\right),\ \left(U_Y^{3D} - P2_Y^{3D}\right)\right) \qquad \textbf{Equació 3.14}$$

I finalment, segons els resultats de les equacions anteriors, aconseguim projectar els dos punts.

$$P1^{P} = \left(\left(Centre\_x^{P} + d\cdot\frac{P1_X^{3D'}}{P1_Z^{3D'}}\right),\ \left(Centre\_y^{P} + d\cdot\frac{P1_Y^{3D'}}{P1_Z^{3D'}}\right)\right)$$

$$P2^{P} = \left(\left(Centre\_x^{P} + d\cdot\frac{P2_X^{3D'}}{P2_Z^{3D'}}\right),\ \left(Centre\_y^{P} + d\cdot\frac{P2_Y^{3D'}}{P2_Z^{3D'}}\right)\right)$$

- ## Emmagatzemant la informació d'objectes a renderitzar

Si el quadrat projectat de la imatge representant de l'objecte interseca al quadrat de pantalla definit a ((0,0),(x_res-1,y_res-1)), llavors vol dir que pot ser que part de la imatge pogui ser pintada a la pantalla i, per aquest motiu, s'emmagatzemarà a una llista.

Per exemple, a la Figura 3.49 (esquerra) tenim els objectes 2,3,1 que són renderitzables perquè estan compresos al FOV de l'observador, per lo qual crearia la llista amb aquests objectes (Figura 3.49 dreta).

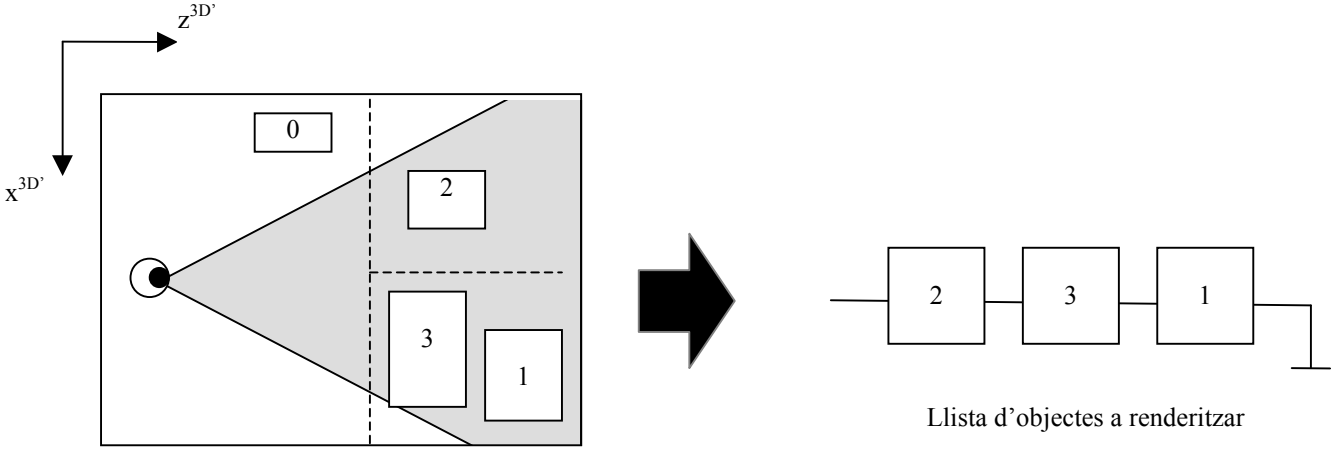


Figura 3.49

- ## Pintat d'objectes

Un cop acabada la renderització del escenari 3D es procedirà a pintar els objectes guardats a la llista, amb el següent algorisme,

```
Accio RenderitzarObjectes(entrada Llista_objectes_render: ^tObjecte_render)
Var
    Llista, Node_actual: ^tObjecte_render
FVar

    Node_actual ←PrimerNode(Llista)
    Mentre no fi(Llista) fer
        PintarImatgeObjecte(Node_actual)
        Node_actual ←SeguentNode(Llista)
    Fmentre

FAccio
```

- **L'algorisme del pintor**

Si pintéssim els objectes del llistat de la Figura 3.49 (dreta), el resultat per pantalla seria el següent.

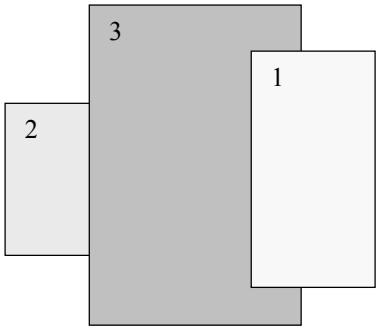
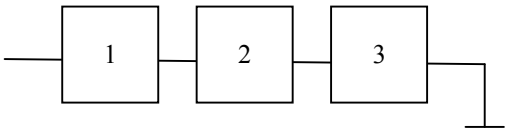


Figura 3.50

A la Figura 3.50, podem observar que l'objecte 1 solapa l'objecte 3. Això no hauria de passar ja que l'objecte 1 és més distant que l'objecte 3, doncs cal un ordre de pintat de objecte, això es soluciona amb l'algorisme del pintor. L'algorisme del pintor es basa en, pintar els objectes o de enrera a endavant (back-to-front), i així, evitar aquests solapaments. Per això cal construir una llista que guardi els objectes ordenats per distància (Z) decreixent com es veu a la següent figura.



Llista d'objectes a renderitzar ordenats en Z.

3.5.4 Estructura de mon 3D extès

El nostre motor esta basat en una estructura de mon 2D. En aquesta secció veurem d'una manera simple, com fer una estructura de món 3D estesa derivada de la estructura actual. Amb respecte a la *renderització,* veurem que apenes no caldrà introduir codi nou.

3.5.4.1 La estructura 3D

La estructura 3D de món es configura amb dos o més estructures de món 2D, solapat a l'espai XZ, però coincidents a l'espai Y. Veiem, per exemple, la creació d'un pont,

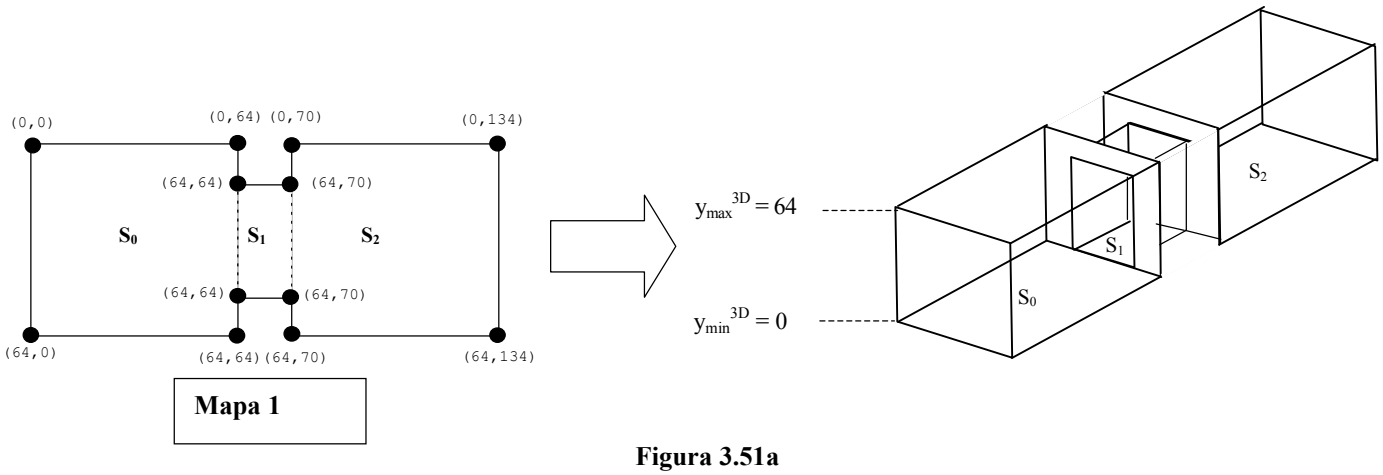


Figura 3.51a

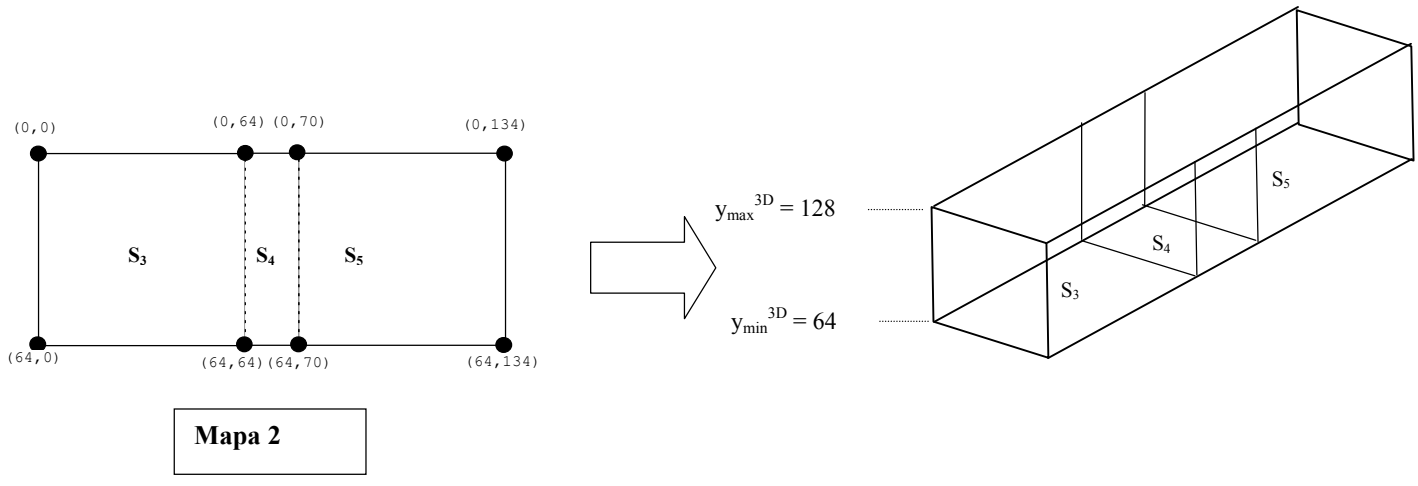
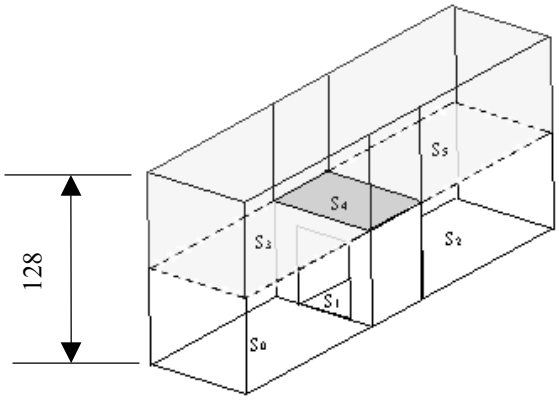


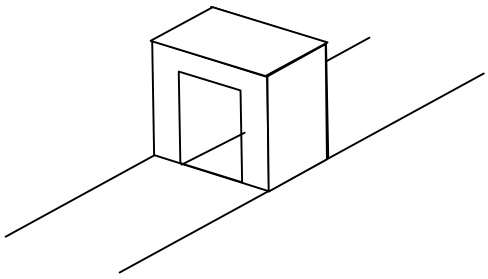
Figura 3.51b

A les figures 3.51a i 3.51b s'observen dos mapes en els quals, alguns sectors estan solapats a l'espai XZ però coincidents en Y, com per exemple el sostre del sector S_0 del mapa 1 i el terra del sector S_3 del mapa 2. Aquest mateix cas passa pels sectors S_2 i S_5. Aquest seria l'exemple de la connexió d'un portal entre dos mapes. Per contrari, els sectors S_4 i S_1 no coincideixen ni en XZ ni altura, pel que no seria connectat per cap portal.

La connexió d'ambdós mapes configuraria el mapa 3D següent,



Vista dels dos mapes connectats.



Vista dels dos mapes renderitzats

3.5.4.2 Renderització de món 3D extès

Per aconseguir la renderització del món 3D, simplement caldrà que el terra o sostre tinguin una referència sector de l'altre mapa, llavors només amb la funció *RenderitzarSector()* aconseguirem la renderització del món 3D.

Per que tingui efecte la renderització de portals terra/sostre, caldrà invocar la renderització de sector si el terra o el sostre són portal,

```
Accio RenderitzarSector(entrada xr, xf,  Observador, Sector) { Renderització sector [xr...xf] }

    LiniaActual ← LiniaInterseccio(xr, Observador, Sector) { S'Obté la línia de intersecció }

    x1 ← xr
    fi ← FALS

    mentre no fi fer {Renderització en sentit horari[1] }

            (...)

    Fmentre

    Si(Sector.Sostre = PORTAL) llavors { Renderitzem sector de mapa conectant}
            RenderitzarSector(xsostre_min, xsostre_max,  Observador, Sector)
    Fsi

    Si(Sector.Terra = PORTAL) llavors { Renderitzem sector de mapa conectant}
            RenderitzarSector(xterra_min, xterra_max,  Observador, Sector)
    Fsi

Faccio
```

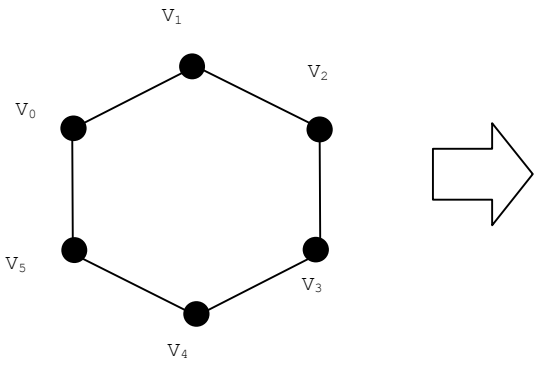
x_{min} i x_{max} (en terra i sostre) representen l'interval x mínim i màxim hàbils per la renderització de terra i sostre. Ja explicarem com aconseguir aquests valors a la secció 4.4.2, entre altres excepcions que cal tenir en compta a l'hora de fer aquest tipus de renderització.

D'aquí, l'avantatge de utilitzar portals al nostre render, és que no hem hagut d'afegir cap cosa nova, només hem reestructurat el mapa.

3.5.5 Pendents

• Estructura

Fins ara hem sigut capaços de renderitzar escenaris de terra i sostre plans. En aquesta secció s'estudiarà la manera de renderitzar els escenaris amb pendents, només afegint un valor de altura en sostre i terra per-vertex, com podem observar a la següent figura.



Taula 3.52a *Sector definit per 6 vèrtexs.*

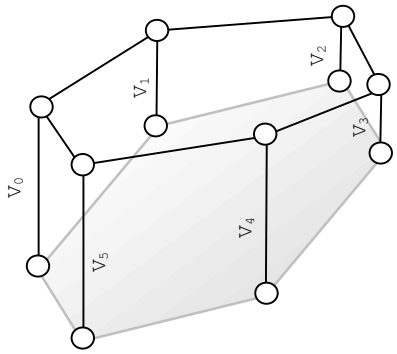


Figura 3.52b *Renderització del terra de sector, segons les altures per-vèrtex definides a la taula 3.52c*

Vèrtex	Altura superior	Altura inferior
V_0	200	0
V_1	200	10
V_2	200	20
V_3	200	20
V_4	200	10
V_5	200	0

Taula 3.52c *Taula on s'hi reflecteix la altura de terra i sostre de cada vèrtex del sector de la figura 3.52a*

• Renderització

Per aplicar aquest efecte a la renderització només caldrà modificar la fórmula de projecció Y per-vertex. Fins ara, la projecció en Y dels vèrtexs es feia en funció dels atributs d'altura de sostre-terra del sector, el qual era constant (pla).

Ara, la projecció Y es farà en funció de l'altura superior/inferior en ambdós vèrtexs de la línia. Llavors, mitjançant un DDA es rasteritzarà els extrems inferiors/superior i és renderitzarà la paret, tal com ja s'ha explicat a la secció 3.3.3.1.

Capítol 4: Excepcions i casos particulars

En aquest capítol té com objectiu nomenar alguns casos que podem trobar-nos durant processos de rendering que impliquen un procés extra que no es va tenir en compte al motor estudiat, com algunes situacions particular que caldrà tractar per evitar càlculs impossibles de realitzar, o situacions de jugador que poden fer el sistema inestable, donant les respectives solucions.

4.1 Raigs

4.1.1 Projeccions Vs raigs

Si recordem, les projeccions es una transformació de l'espai de món $\in \Re$ a l'espai de pantalla $\in Z$. Nosaltres projectem els vèrtexs de línia a l'hora de ser processada per pantalla. Val dir que si la línia és portal, més tard si llançarà un raig per la columna on s'hagi projectat **el primer vèrtex**. Per això, és molt important tenir cura a l'hora de acotar el vèrtex a l'espai Z, i **cal que el raig entri per la dreta del vèrtex projectat per assegurar la penetració dins el portal**.

Existeixen 3 tipus de arrodoniment coneguts: *ceil*, *floor* i *round*.

Sigui *x* qualsevol numero $\in \Re$ i les funcions *PartFraccional()* i *PartEntera()* que retornen la part entera i la seva part fraccional respectivament; les funcions *ceil()*, *floor()* i *round()* es defineixen per:

Ceil(x)

```
Funcio ceil(entrada x:real) retorna enter

        si(PartFraccional(x) != 0)  i (x >= 0) llavors
                ceil ← PartEntera(x)+1
         Altrament
                ceil ←PartEntera(x)
        Fsi
FFuncio
```

Floor(x)

```
Funcio floor(entrada x:real) retorna enter
        si(PartFraccional(x) != 0)  i (x < 0) llavors
                floor ← PartEntera(x)-1
         Altrament
                floor ← PartEntera(x)
        fsi
            FFuncio
```

Round(x)

```
Funcio round(entrada x:real) retorna enter

        round ← PartEntera(x+0.5)

ffuncio
```

Observem el següent exemple gràfic,

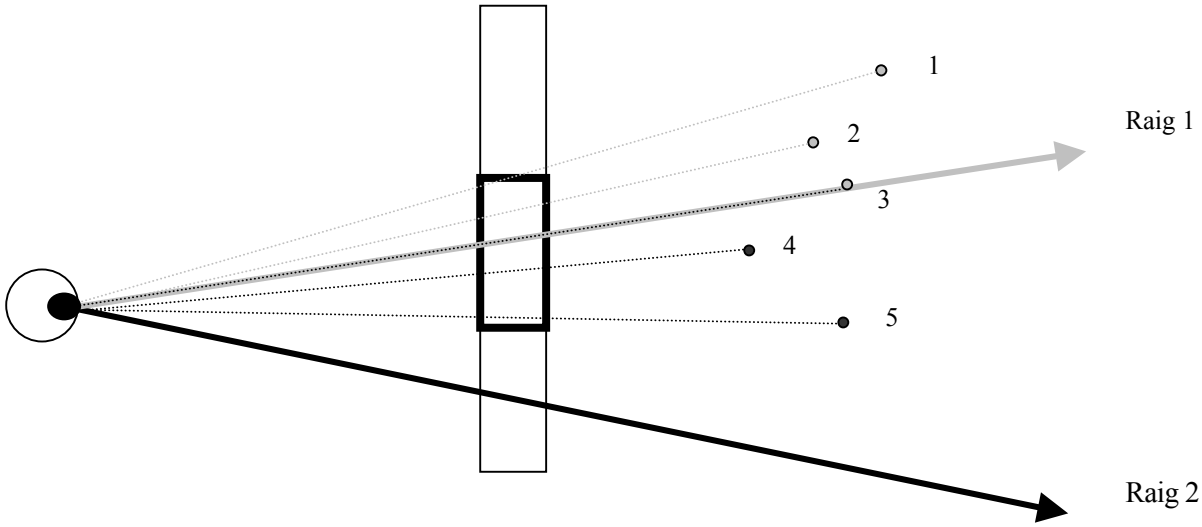


Figura 4.1

A la figura 4.1 hi tenim vèrtexs a l'espai d'ull que quedarien tots ells projectats a la columna remarcada, i el llançament de dos raigs al centre de la columna[3].

Podem observar el primer i segon vèrtex, la columna per on es llança el 1er raig seria l'adequada, perquè el raig es troba a la dreta dels vèrtexs. Per la resta caldria llançar el raig per la següent columna (raig 2).

La funció *round()* potser seria la més adequada ja que dona el resultat de la columna més pròxim i per consegüent la columna més pròxima a llançament de raig.

Però observem el raig que es llança al centre de cada columna (al vèrtex 3). Si l'aritmètica utilitzada no fos prou robusta podríem tenir problemes ja que el raig VS projecció són quasi coincidents. Per consegüent, el resultat de la funció *round()* podria donar la columna equivocada.

Per assegurar la penetració del raig, s'utilitzarà la funció d'arrodoniment *ceil()*. Segons la figura 4.1, la columna resultant serà en tota cas la columna següent que, per consegüent, resultaria el llançament del raig 2 i que, aquest, assegura estar a la dreta de tots els vèrtexs.

[3] El llançament de raig pel centre de columna ja es va acordar a la secció 2.1.4.

4.1.2 Raigs intersecta línia amb pendent

A la secció 3.4.5 vam veurem com podiem aplicar pendents als nostres escenaris evaluant la altura per-vertex en vés de la altura de sector. No obstant, quan es troba una L.I amb pendent cal trobar l'offset de altura pertinent a la intersecció raig-línia de mon (V_i^{3D}, figura 4.2).

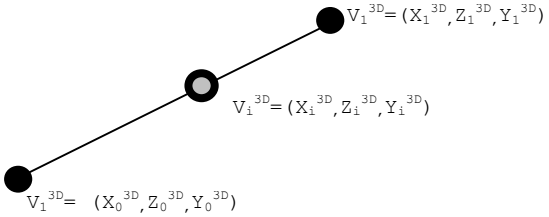


Figura 4.2

Per trobar l'offset altura inicial (Y_i , figura 4.2), cal primer tot saber els pendents altura en funció de X i Z,

$$myx = \frac{V_1^{3D}.Y - V_0^{3D}.Y}{V_1^{3D}.X - V_0^{3D}.X}$$

Equació 4a *Pendent altura en funció Z*

$$myz = \frac{V_1^{3D}.Y - V_0^{3D}.Y}{V_1^{3D}.Z - V_0^{3D}.Z}$$

Equació 4b *Pendent altura en funció X*

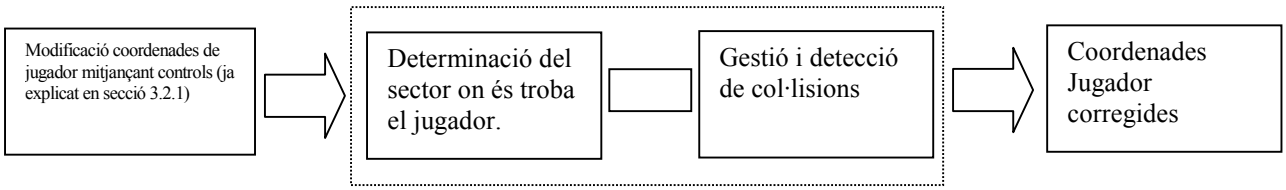
Llavors amb les ecuacions 4a, 4b aconseguim l'offset d'altura inicial,

$$Fy(X_i^{3D}, Z_i^{3D}) = V_0^{3D}.Y + myx \cdot (x_i^{3D} - V_0^{3D}.X) + myz \cdot (Z_i^{3D} - V_0^{3D}.Z)$$

4.2 El Jugador

Al anterior capítol vam introduir com aplicar moviments al jugador mitjançant els control de la maquina (secció 3.2.1)

Ara explicarem com determinar el sector on es troba el jugador (sector inicial per començar la renderització). També, cal tenir en compte que a l'hora de fer els moviments de jugador, pot ser que pugui sortir-se fàcilment del mapa, per això caldrà fer una detecció i gestió de col·lisions. I altres aspectes que s'han de considerar i que més tard s'explicaran. Un esquema representatiu del tractament que s'ha fet pel jugador és el següent.



AREA DE TRACTAMENT

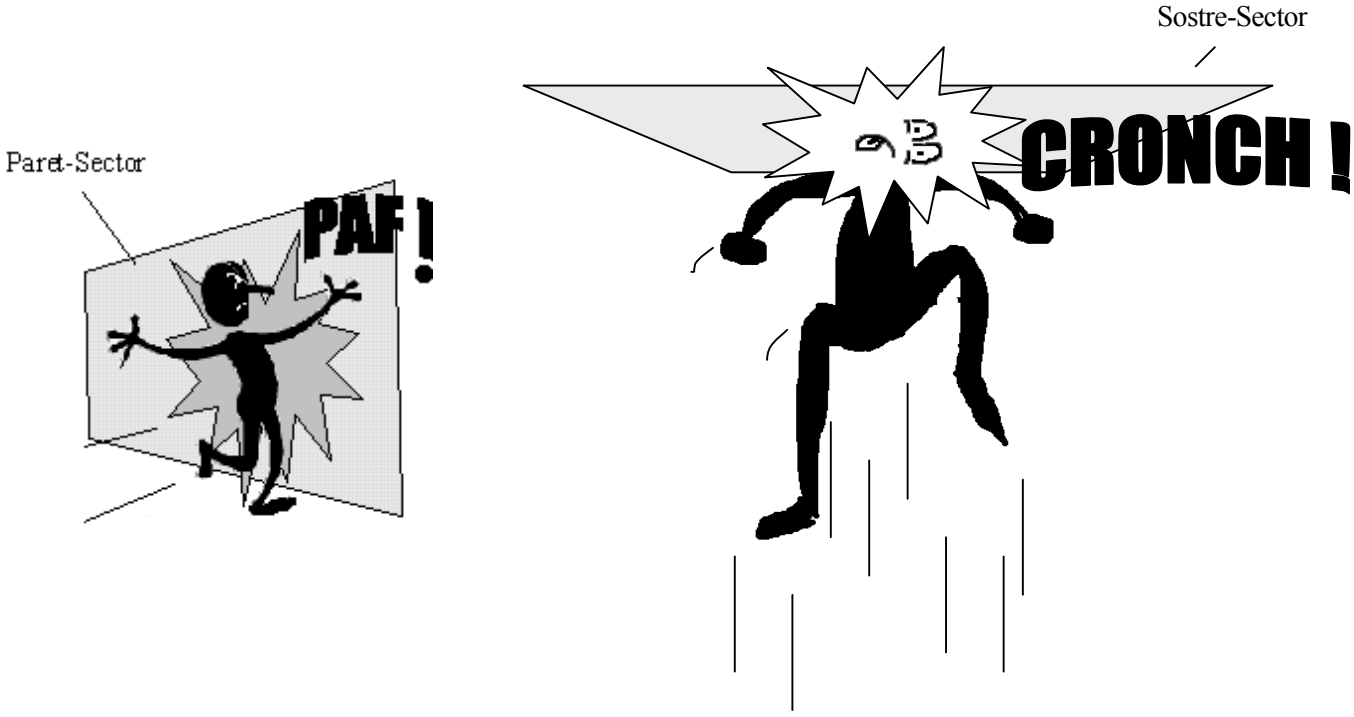
4.2.1 Determinació del sector-jugador

Tal com es va explicar a la secció 3.1, el mapa el defineix una sèrie de sectors convexos. Per tant, aplicant l'algorisme explicat a la secció 2.4.2, trobarem el sector-jugador mitjançant una cerca lineal, comprovant un a un el sector-punt jugador fins trobar-lo.

```
Funció SectorJugador(e Jugador, SectorMapa e/s SectorTrobat) retorna booleà

    trobat ← FALS
    i ←0

    Mentre i < MAX_SECTORS_MAPA-1 i no trobat fer

        Si (Punt_Pertany_Poligon_Convex(Jugador,SectorMapa[i])) llavors
                Trobat ← Cert
                SectorTrobat ← SectorMapa[i]
        Fsi

        i←i+1

    Fper

    Si(trobat) llavors SectorJugador ← Cert
    Altrament SectorJugador← FALS
    Fsi

Ffunció
```

4.2.2 Detecció i gestió de col·lisions

El nostre jugador podrà moure's lliurement pel mapa, però cal tenir en compte que el mapa es troba limitat per línies (sòlides) o pels nivells de altura. En aquesta secció s'implementarà un sistema de col·lisions perque el jugador no traspassi una paret o no superi aquest límits de mapa abans comentats.



Cada tipus de detecció de col·lisió estarà basada pel tipus de moviment que hagi efectuat el jugador, ja explicats a la secció 3.2.1 del tema anterior. Per exemple, pel moviment 2D del jugador caldria fer una comprovació de si està prop d'una línia sòlida del mapa 2D (la línia que representa paret) i llavors, si escau, fer la correcció de posició.

4.2.2.1 Detecció de col·lisions 2D

- **Detecció de col·lisió jugador amb línies sòlides de mapa**

El sistema de detecció que utilitzarem serà **aproximat**. Primerament, s'estableix una grandària al jugador que, per fer el problema senzill, aquesta grandària el definirà un quadrat englobant al punt de jugador. Així doncs, si volem aturar el jugador davant d'una paret a una distància de **aproximadament** 32 unitats, el quadrat de jugador (Q_J) serà de 64x64 unitats (figura 4.3).

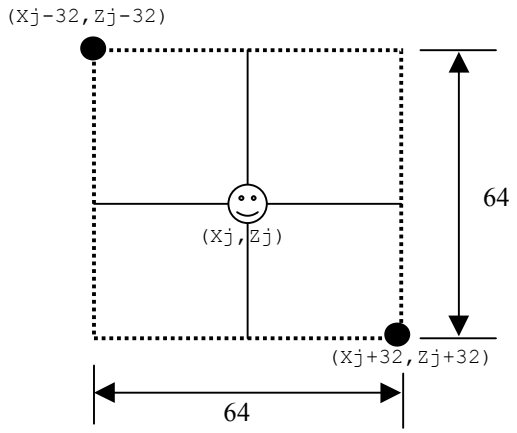


Figura 4.3

El sistema de detecció segueix els següents passos:

1. Fent referència les equacions 3.1 i 3.2 (secció 3.2.1 del capítol anterior) que explicava com obtenir els increments de moviment DeltaZ i DeltaX (explicats a la secció 3.2.1.1), segons el signe resultant d'aquests increments, comprovarem si 6 punts extrems a Q^J pertanyen al sector de jugador, a partir del problema que la pertinença d'un sector convex estudiat a la secció 2.4.2 del capítol 2.

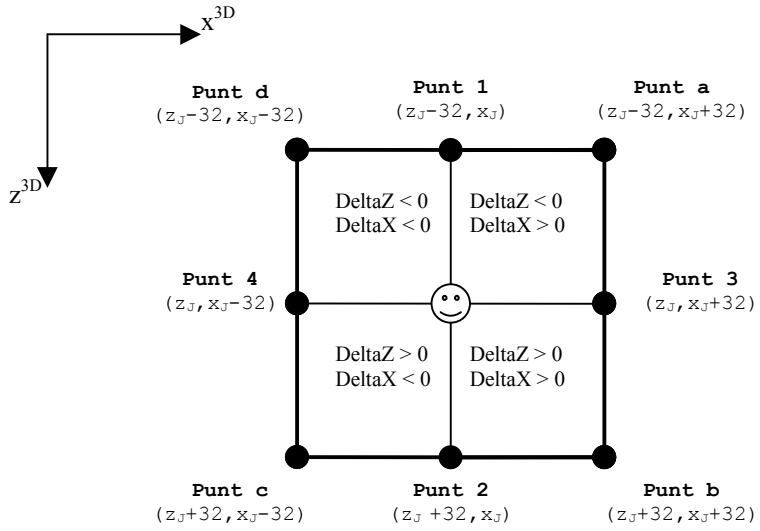


Figura 4.4

A la figura 4.4, mostrar les coordenades dels 6 punts a comprovar la seva pertinença dins el sector de jugador, segons el signe delta X delta Z. S'ha provat que tractant els punts en el següent ordre, el sistema detecta la col·lisió de manera correcte:

1.1 Segons la figura, primer caldria comprovar les direccions ortogonals es a dir els punts **1,2,3 o 4**.

1.2 Llavors, es comprovaran les direccions diagonals: **a,b,c i d**.

2. Si algun dels punts mencionats **no** pertany al sector-jugador, és llançarà un raig amb **angle definit entre l'origen del llançament**[4] (posició del jugador) i la posició del punt, amb l'objectiu de saber la distància des del jugador a la paret. Llavors, si la **intersecció** raig-línia **pertany** al $\mathbf{Q^J}$ **i** la **línia** és **sòlida**, llavors **s'haurà detectat la col·lisió**.

[4] A la figura 4.8, per exemple, per els punts 1 i 2, és llançaria un raig vertical de 270º i 90º respectivament. Pels punts 3,4 un raig horitzontal de 0º i 180º respectivament. Per la resta raigs amb angles 45º, 135º, 225º i 315º, pels punts a,b,c i d respectivament.

Agafarem com a base de la funció del traçat de raig vist al llistat 3.1a. Com ja es va explicar, aquesta funció traçar el raig fins topar amb una paret. Llavors, finalitza la recursió amb el retorn de la seva intersecció.

```
Funcio TraçarRaig_Jugador_Paret(e PosicioRaig, AngleRaig, SectorRaig) retorna punt2D

  { Obtenir línia de intersecció i intersecció raig-línia}

  si Linia és Portal llavors { En línies transparent (portals) el raig prosegueix el traçat }

        TraçarRaig_Jugador_Paret ← TraçarRaig_Jugador_Paret(Interseccio, Angle, Mapa.Sector[Linia.Portal])

  altrament { Cas trivial. El raig ha interceptat una paret, es retorna la intersecció }

        TraçarRaig_Jugador_Paret ← interseccio
  fsi

FFuncio
```

Llistat 4.4

- ## Consideració de paret per diferencia de nivells

Seguint el procediment per detectar el jugador quan es troba a una distància pròxima a una paret presentat a l'anterior punt, es important dir que el raig no només aturarà el seu traçat quan intercepti una paret (línia sòlides). Si durant en el traçat de raig es troba amb un portal que connecta amb un sector on la diferencia de l'ull respecte el nivell d'alçada terra/sostre és molt gran, és considerarà com a paret i el raig s'aturarà. Les condicions perquè es consideri paret serà quan la diferencia entre ull i sostre/terra sigui inferior a 16 unitats. Per consegüent, cal modificar la funció *TraçarRaig_Jugador_Paret()* i incloure el següent codi marcat amb un rectangle per considerar (o no) el portal com a paret.

```
Funcio TraçarRaig_Jugador_Paret(e ..., AlturaUll, ...) retorna punt2D
  {...}
  si Linia és Portal llavors { També es comprovarà la diferencia de nivells adjacents amb ull }

        si (Sector.AlturaTerra – AlturaUll) < 16 o (AlturaUll – Sector.AlturaSostre) < 16 llavors { Es considerarà paret }

              TraçarRair_Jugador_Paret ← interseccio
        altrament { El raig prosegueix el seu recorregut }
              {...}
        fsi
  altrament { El raig ha interceptat una paret, es retorna la intersecció }
        {...}
  fsi
        FFuncio
```

Llistat 4.5

4.2.2.2 Gestió de col·lisions 2D

Com s'ha explicat anteriorment, si la intersecció retornada per la funció *TraçarRaig_Jugador_Paret()* pertany al quadrat Q^J (figura 4.3), s'haurà detectat una **col·lisió 2D entre jugador i paret**. S'ha convingut que, quan es detecti esmentada col·lisió, es modifiqui la posició del jugador per tal que aquest s'estableixi a 32 unitats de la paret, segons les dimensions establertes de Q^J. Per exemple, i fent referència a la figura 4.4, en el cas que el **punt1** no pertanyés al sector el llançament de raig vertical obtingués la intersecció de coordenada Z per sota 32 unitats, només caldria actualitzar la coordenada Z per tal que s'estableixi a 32 unitats respecte la Z de intersecció (Xj queda igual).

$$Z_J \leftarrow \text{Interseccio.Z} + 32.$$

4.2.2.3 Detecció i gestió de col·lisió ull-nivell

En algunes accions com volar pot ser que l'altura d'ull rebassi els nivells de sostre i de terra. Per això, caldrà modificar l'altura l'ull perquè no sobrepassi les cotes mínimes/màximes del sostre-terra ja establerts en el sector.

```
Si (AlturaSostre- AlturaUll < 16) llavors { Detectada col·lisió ull-sostre → Gestió}
        AlturaUll ← AlturaSostre – 16
Fsi
Si (AlturaUll-AlturaTerra < 16) llavors { Detectada col·lisió ull-terra → Gestió}
        AlturaUll ← AlturaTerra + 16
Fsi
```

4.2.2.4 L'Algorisme

Podem resumir el que s'ha explicat en el següent tall de codi. En ell, dona la idea del algorisme que detectaria i gestionaria les col·lisions de jugador, tal com s'ha explicat a les seccions anteriors:

```
{ Detecció i gestió de col·lisions 2D }

{Comprovació dels punts 1,2,3,4 (si escau segon el signe deltaXZ).}

Si DeltaZ < 0 llavors {Cal comprovar el punt 1 }
        Si punt1 ∉ SectorJugador llavors { llencem raig vertical vertical cap al nord}

                AngleRaig ← 270º
                Interseccio ← TraçarRaig_Jugador_Paret(PosicioJugador,AngleRaig, AlturaUll, SectorJugador)
                Si  ZJ- Interseccio.Z < 32 llavors { S'ha detectat col·lisió en coordenada Z d'intersecció → fer gestió }

                        ZJ← Interseccio.Z+32
                Fsi
        FSi
FSi
Altrament Si DeltaZ > 0 llavors { Cal comprovar el punt  2 }
        Si punt2 ∉ SectorJugador llavors { llencem raig vertical cap al sud}
                {...}
        FSi
FSi

{De manera similar comprovem els punts 3 i 4 segons el sigen DeltaX}

Si DeltaX < 0 llavors {Cal comprovar el punt 4}
                {...}
altrament si DeltaX > 0 llavors { Cal comprovar el punt  3 }
                {...}
fsi
FSi

{Comprovació dels punts a,b,c,d (si escau segon el signe deltaXZ)}
Si DeltaZ < 0 llavors {Cal comprovar el punt d i a}
        Si punt d ∉ SectorJugador llavors { llencem raig vertical vertical cap al nord-est}
                AngleRaig ← 225º
                Interseccio ← TraçarRaig_Jugador_Paret (PosicioJugador,AngleRaig, AlturaUll, SectorJugador)
                Si ZJ - Interseccio.Z < 32 llavors {S'ha detectat col·lisió en coordenada Z d'intersecció → fer gestió }
                        ZJ← Interseccio.Z+32
                FSi
                Si XJ - Interseccio.X < 32 llavors {S'ha detectat col·lisió en coordenada X d'intersecció → fer gestió }
                        ZJ← Interseccio.Z+32
                FSi
        FSi
        Si punt a ∉ SectorJugador llavors { llencem raig vertical vertical cap al nord-oest}
                {...}
        FSi
        Altrament Si DeltaZ > 0 llavors {Cal comprovar el punt b i c}
                {...}
        FSi
FSi

 Si deltaX < 0 llavors {Cal comprovar el punt c i d}
                {...}
Altrament Si deltaX > 0 llavors {Cal comprovar el punt a i b}
                {...}
        FSi
FSi

{ Detecció i gestió de col·lisions segons l'altura d'ull amb nivells del sector actual}

Si (AlturaSostre- AlturaUll < 16) llavors { Detectada col·lisió ull-sostre → Gestió}
        AlturaUll ← AlturaSostre – 16
FSi
Si (AlturaUll-AlturaTerra < 16) llavors  { Detectada col·lisió ull-terra → Gestió}
        AlturaUll ← AlturaTerra + 16
FSi
```

4.2.3 Situacions particulars

Cal dir que el jugador pot estar en situacions del mapa on pot causar un estat de indeterminisme jugador interseca un portal.

Quan la posició del jugador interseca un portal no tenim determinat el sector on serà llançat, el raig i, per consegüent, es possible que no es trobi la L.I. I encara pitjor, pot ser que el jugador intersecti un vèrtex de portal i llavors, el grau de indeterminisme es total, ja que, aquest vèrtex podria connectar múltiples portals.

- Jugador intersecta un portal

Per solventar aquesta excepció, s'**evitarà a tot cost que el jugador intersecti el portal**. Per cercar el sector on es troba el jugador, s'ha fet servir l'algorisme que detecta si un punt pertany a un sector convex (secció 2.4.2). La cerca es simple, es va comprovant sector-per-sector fins trobar el sector on el punt de jugador que hi pertany.

El que es farà serà modificar aquest algorisme, perquè a part de determinar si un punt pertany a un sector aprofiti per dir si intersecta alguna de les línies que el defineixen. Llavors, si el jugador intersecta una línia, que suposadament ha de ser un portal, buscarem dins 8 punts unitaris que rodegen el punt de jugador (figura 4.5), fins trobar-ne un que es trobi dins el sector. Aquest serà el punt cap on serà desplaçat el jugador.

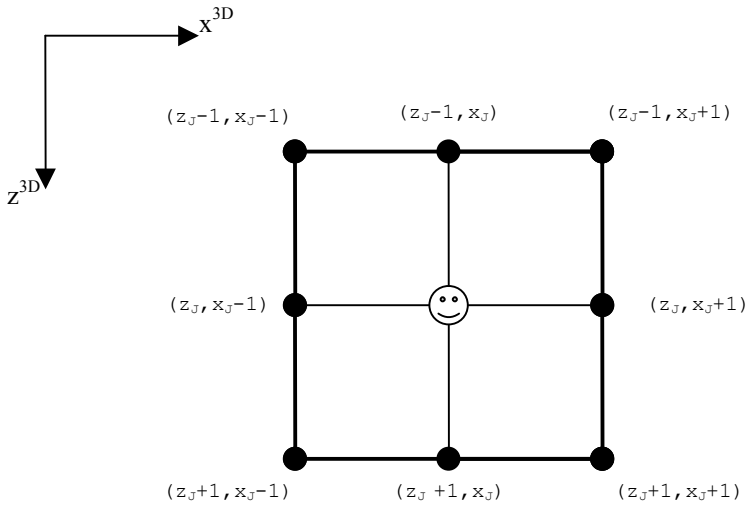


figura 4.5 *En cas que el jugador intersecti línia, es provarà trobar una posició entre les 8 vuit que rodegen el jugador per hi pertanyi però sense intersectar.*

- El jugador intersecta un vèrtex

Quan el jugador intersecta un vèrtex de portal, pot sorgir un cas de indeterminisme més gran. Un vèrtex de portal pot interconnectar **N** sectors, llavors si el jugador l'intersecta es quan sorgeix el cas de indeterminisme.

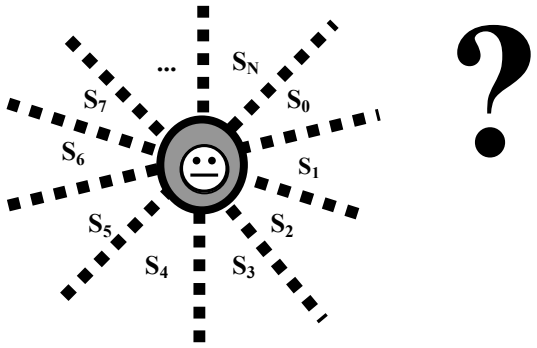


figura 4.9 *El jugador intersecta un vèrtex de portal el qual és comú a N sector. Per això, no hi tenim un sector determinat pel jugador ni pel punt de partida de raig.*

Igual que abans, la solució al problema per evitar aquest cas de indeterminisme és **evitar que el jugador mai pugui intersectar cap portal.**

Si les coordenades de jugador està definit **al domini dels reals**, pot donar-se sempre la probabilitat que el jugador intersecti un vèrtex. No obstant, si portem el jugador al domini dels reals **i es força que la seva part fraccional estigui en dècimes de 0'5, el jugador mai interceptarà cap vèrtex ja que cada vèrtex $\in Z^2$**

Per això, caldrà renovar la correcció de jugador quan hi hagi col·lisió, per què la coordenada resultant asseguri esta en dècimes de 0'5. Per exemple, i fent referència a la figura 4.4, en el cas que el **punt1** no pertanyés al sector el llançament de raig vertical obtingués la intersecció de coordenada Z per sota 32 unitats, caldria actualitzar la coordenada Z per tal que s'establís a **32.5** unitats respecte la Z de intersecció.

$Zi \leftarrow enter(interseccio.Z) + 32.5$

4.2.4 Obtenció de la altura jugador en un pendent.

Quan el jugador, es troba en un sector on existeix un pendent en el terra, degut a que no es constant, cal calcular l'alçada segons la inclinació del pla .

El procés per fer-ho es resumeix és el següent:

1. Llançarem 2 raig verticals o horitzontals per trobar dos línies de intersecció (I_1, I_2 a la figura 4.10). En en nostre cas llençarem 2 raigs verticals (angles 90º, 270º), desde la posició del observador.

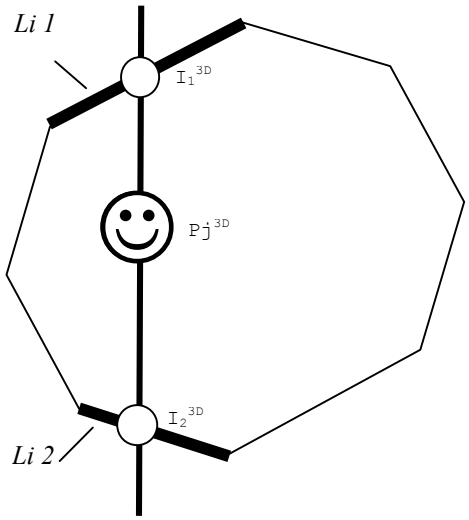


Figura 4.10

2. De les dues L.I (Li1 i Li2 a la figura 4.10) obtingudes, calculem els pendents de món per obtenir, les components d'alçada, corresponents a les interseccions, de la mateixa manera que s'ha vist a la secció 4.1.1.

$$myx_{Li1} = \frac{Li1.V_1^{3D}.Y - Li1.V_0^{3D}.Y}{LI1.V_1^{3D}.X - Li1.V_0^{3D}.X}$$

Equació 4a *Pendent altura en funció X de la L.I 1*

$$myz_{Li1} = \frac{Li1.V_1^{3D}.Y - Li1.V_0^{3D}.Y}{Li1.V_1^{3D}.Z - Li1.V_0^{3D}.Z}$$

Equació 4b *Pendent altura en funció Z de la L.I 1*

$$myx_{Li2} = \frac{Li2.V_1^{3D}.Y - Li2.V_0^{3D}.Y}{Li2.V_1^{3D}.X - Li2.V_0^{3D}.X}$$

Equació 4c *Pendent altura en funció X de la L.I 2*

$$myz_{Li2} = \frac{Li2.V_1^{3D}.Y - Li2.V_0^{3D}.Y}{Li2.V_1^{3D}.Z - Li2.V_0^{3D}.Z}$$

Equació 4d *Pendent altura en funció Z de la L.I 2*

Llavors amb les equacions anteriors[3] aconseguim les components d'alçada,

$$I_1^{3D}.Y = Li1.V_0^{3D}.Y + myx_{Li1} \cdot (I_1^{3D}.X - Li1.V_0^{3D}.X) + myz_{Li1} \cdot (I_1^{3D}.Z - Li1.V_0^{3D}.Z)$$

$$I_2^{3D}.Y = Li2.V_0^{3D}.Y + myx_{Li2} \cdot (I_2^{3D}.X - Li2.V_0^{3D}.X) + myz_{Li2} \cdot (I_2^{3D}.Z - Li2.V_0^{3D}.Z)$$

3. Finalment, obtenim les components d'alçada de la línia formada pels punts I_1, I_2. Amb aquests, obtenim la alçada de jugador.

$$myx_{i1-i2} = \frac{I_2^{3D}.Y - I_1^{3D}.Y}{I_{2}^{3D}.X - I_1^{3D}.X}$$

Equació 4e *Pendent altura en funció X de I_1 i I_2*

$$myz_{i1-i2} = \frac{I_2^{3D}.Y - I_1^{3D}.Y}{I_2^{3D}.Z - I_1^{3D}.Z}$$

Equació 4f *Pendent altura en funció Z de I_1 i I_2*

$$P_j^{3D}.Y = I_1.Y + myx_{i1-i2} \cdot (x_j^{3D} - P_j^{3D}.X) + myz_{i1-i2} \cdot (Z_j^{3D} - P_j^{3D}.Z)$$

[3] Com una observació, els valors de les equacions 4a 4b 4c i 4f poden ser precalculats.

4.3 Tires

4.3.1 retallats

Un portal sempre es una finestra a un altre sector, pel que només serà renderitzat a l'espai que el delimita, es a dir, per l'interval x i pel rasteritzat inferior/superior. Per això, abans de processar un portal cal rasteritzar els extrems superior/inferior i guardar els seus valors en un vector denominades cotes y. Aquestes cotes *y* ens serviran per evitar sobrescriure les tires de paret que prèviament ja s'havien *renderitzat*, ja que si recordem que el nostre *render* es endavant cap enrera. Les cotes inferiors/superiors els guardarem dins de dos vectors anomenats *ymin*, *ymax* (figura 4.15) respectivament d'igual amplada a la resolució x.

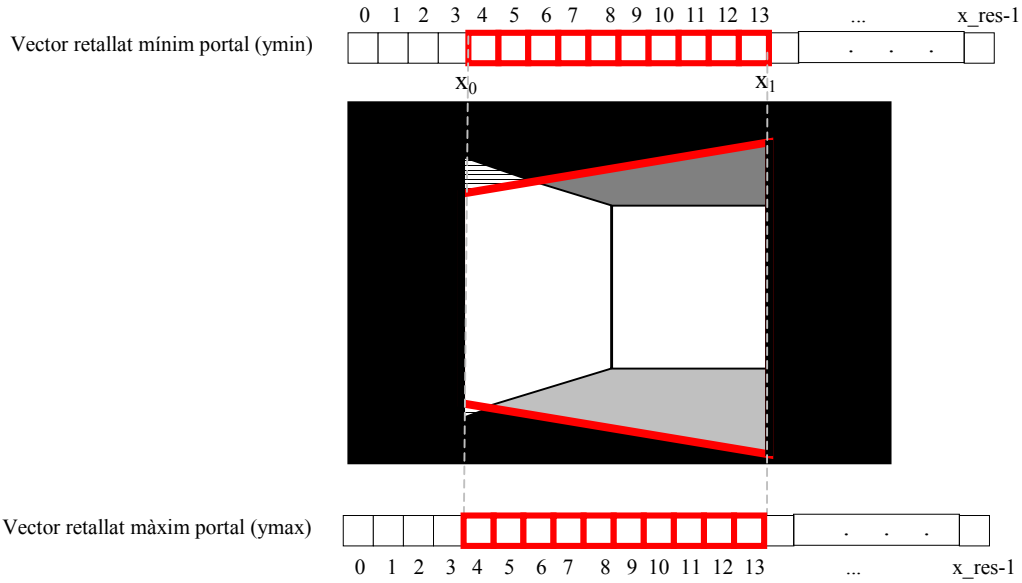


Figura 4.15 *A la figura podem observar la renderització del portal al interval de columna x_0 –x_1 i limitat pel seu rasteritzat inferior/superior*

Aquests vectors explicaran a cada columna, l'interval vàlid i que caldrà terir en compte a l'hora de pintar una tira de paret del portal.

- **Inicialització**

Podem dir que la pantalla és el primer portal que ens trobem abans de pintar res al interval de pantalla, per això aquests vectors estaran inicialitzats a la dimensió espaial de pantalla (figura 4.16).

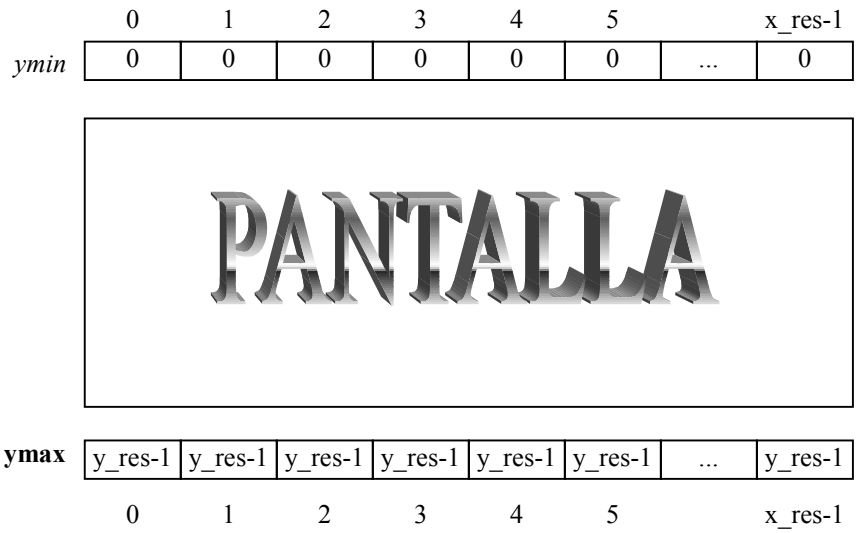


Figura 4.16

En la rasterització d'un portal s'actualitzaran aquestes cotes de retallat amb la informació provenint dels DDA del terra connectant i del sostre connectant, ja explicats al punt 2 de la secció 3.3.2.2, en cas d'haver desnivell.

Per aquest motiu, serà imprescindible fer el pas d'aquests vectors a la funció *renderitzar* sector. A mesura que s'hi vagin *renderitzant* portals, l'interval vàlid de *renderització* de tira s'anirà fent cada cop més petit fins arribar en un punt en què no sigui possible la *renderització* ja sigui perque les projeccions no ho han permès, o perquè s'ha interceptat una tira paret.

• Actualització dels vectors de retallat

L'actualització dels vectors de retallat tindrà lloc quan alguna de les rasteritzacions de portal obligui a fer la modificació del interval de retallat.

- Sense desnivell

Si el portal no té desnivell respecte el sector connectant s'haurà de capturar les cotes *y* resultants del seu rasteritzat com es va explicar a la secció 3.3.2.1, però només si la cota *y* actual ho permet. Mitjançant la següent fórmula podem actualitzar el retallat de portal de manera automàtica.

$ymin[x_r] \leftarrow max(Y1, ymin[x_r])$
$ymax[x_r] \leftarrow min(Y2, ymax[x_r])$

- Amb desnivell

Fent referència secció 3.3.2.2 del capítol 3 que explica l'exemple d'una rasterització de portal amb desnivell, observem que els resultats de la rasterització del sostre i terra connectant obligaria a actualitzar els vectors de retallat. La següent formula actualitzaria de manera automàtica els vectors de retallat, durant el rasteritzat de línia,

$ymin[x_r] \leftarrow max(y2_S^P, ymin[x_r])$
$ymax[x_r] \leftarrow min(y1_T^P, ymax[x_r])$

Però pot passar el següent, si, per exemple, la rasterització del sostre connectant es trobés per sobre del sostre actual, llavors la rasterització connectant no seria possible i, per consegüent, els vectors de retallat no es veurien modificats. Igual passaria amb el terra si el seu connectant es trobés per sota seu (figura 4.18).

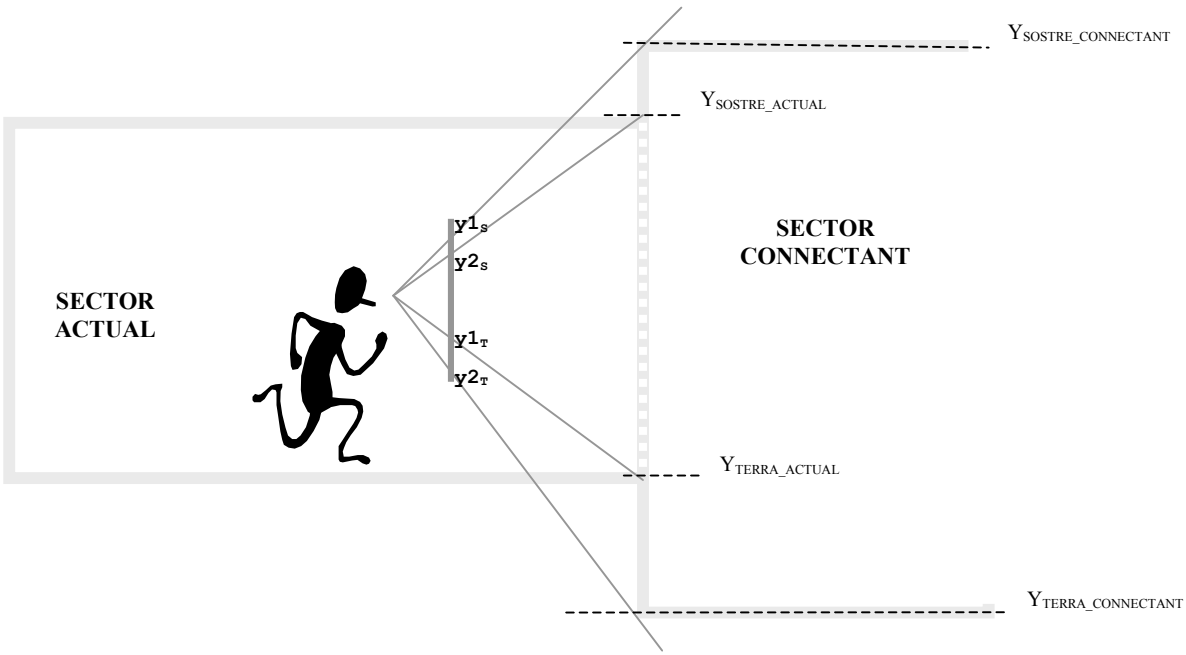


Figura 4.18

Podem observar a la figura 4.18 que **el portal està marcat per la rasterització de línia determinades per les altures del sector actual**. Per tant, per la actualitzat de les cotes de retallat cal tenir en compte la cota *y* més restrictiu. Això es pot fer de manera directa empleant una variant anterior,

$ymin[x_r] \leftarrow max(max(Y1_S, Y2_S), ymin[x_r])$
$ymax[x_r] \leftarrow min(min(Y1_I, Y2_I), ymax[x_r])$

• Procés de retallat de tira

Abans de pintar una tira vertical de paret cal tenir en compte els valors de retallat que disposem als vectors *ymin* i *ymax* ∈ **[0, ... , y_res-1]** en cada iteració x, durant el seu rasteritzat. Com ja s'ha dit, cal retallar adequadament les tires verticals per evitar sobrescriure les ja renderitzades als sectors anteriors.

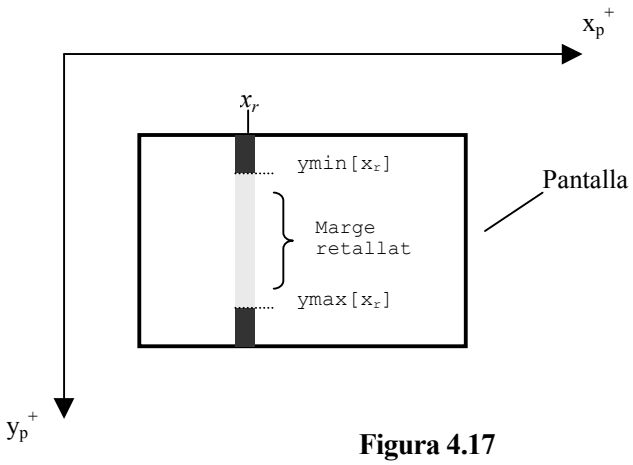
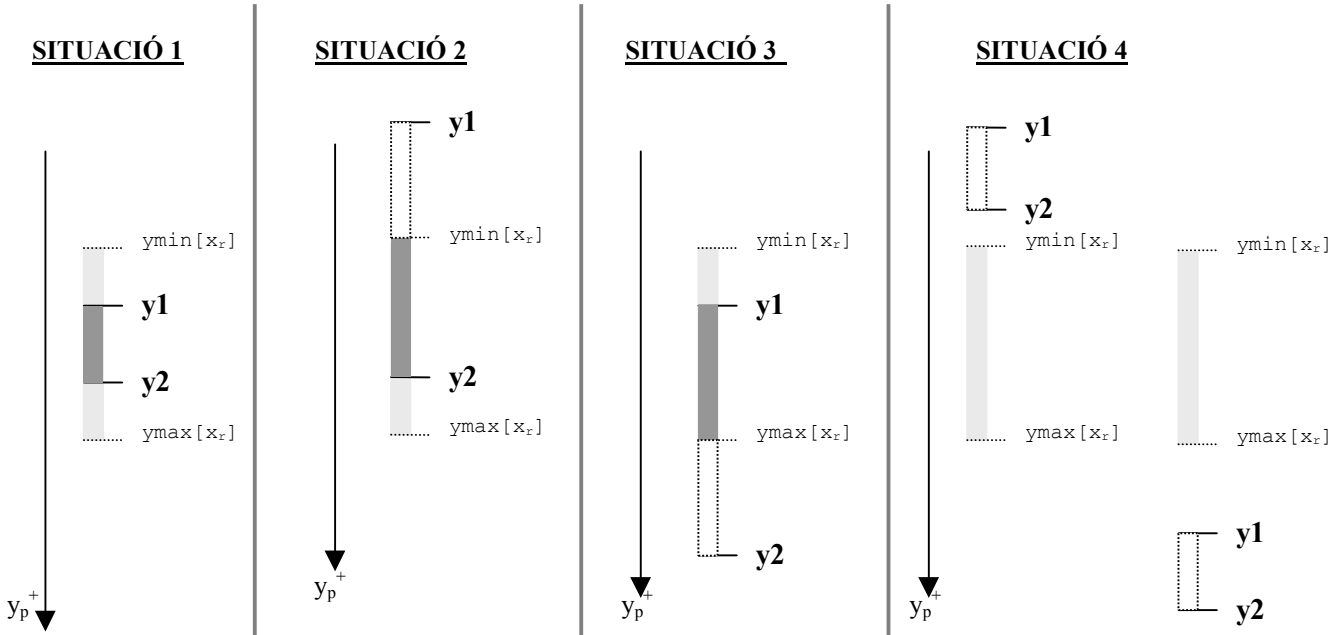


Figura 4.17

Una tira es *renderitzarà* sempre respectant els límits establerts per les cotes determinades per ymin, ymax a la columna actual (figura 4.17). El procés de retallat de tira es basa en reassignar coordenades superior inferior del rasteritzat resultant a la columna actual. Podem tenir les 4 situacions següents,



En aquesta situació *y1* i *y2* és troben dins el marge de retallat, per tant no caldrà fer cap correcció.

En aquesta situació *y1* és troba per sota de la cota *ymin*, per tant s'haurà de corregir *y1*, assignant-li *ymin*.

En aquesta situació *y2* és troba per sobre de la cota *ymax*, per tant s'haurà de corregir *y2*, assignant-li *ymax*.

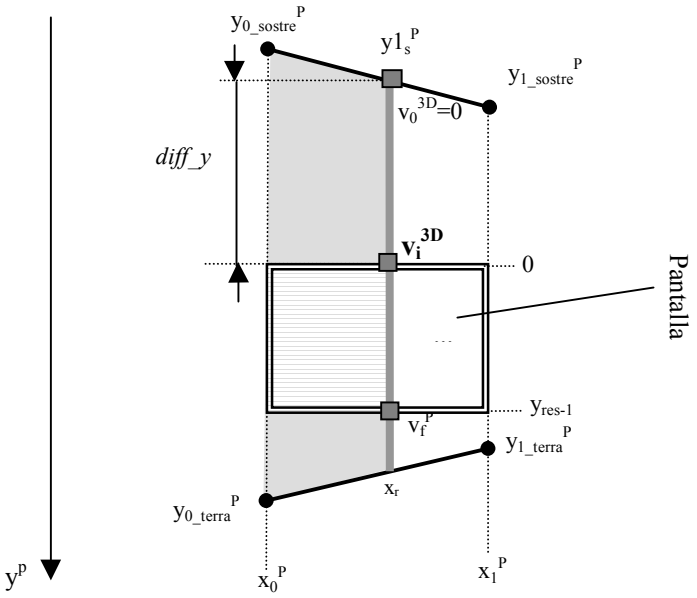
En aquest cas y1 i y2 es troben fora de retallat, per tant no es renderitzarà la tira.

Si les alguna de les coordenades pertany al marge de retallat, podrem corregir les coordenades y1 i y2 directament fent us de les funcions ***min()*** i ***max().***

```
Si no ((y2 < ymin) o (y1 > ymax)) llavors {Podem renderitzar tira}

        y1 ← max(ymin,y1)
        y2 ← min(ymax,y2)

        (...)

altrament {y1,y2 fora de retallat → No renderitzem tira !}
Fsi
```

• Coordenada inicial de pintat (v_0^{3D})

A la secció 3.3.2.3 és va explicar un exemple senzill a l'hora de fer el mapeig de paret, associant la coordenada *v* inicial (v_0^{3D}) als valors de rasterització superiors de paret. No obstant, quan una cota *y* superior de tira de paret sobresurt dels límit mínim de **retallat cal trobar un nou valor v inicial vàlid** (v_i^{3D}). A la següent figura mostra un exemple on el rasteritzat superior de paret sobresurt del retallat mínim de pantalla (0).



En aquest cas per trobar vi^{3D} cal fer l'offset respecte l'inici v_0^{3D}, multiplicant *diff_y* per el factor escala.

$$V_i^{3D} = diff\_y \cdot Escala\_v^{3D} \qquad \textit{Equació 4.1}$$

La *diff_y* representa la diferencia que hi ha entre el rasteritzat superior i la cota delimitadora mínima (ymin[x_r]), que es troba com,

$$diff\_y = ymin[x_r] - y1_s^P$$

4.4.2 Retallats a la renderització 3D extès

A la secció 3.4.4.2 es va veure que, per aconseguir la renderització del món 3D extès, només cal renderitzar el sector on el portal de terra o sostre apunten, amb la funció *RenderitzaSector()*. Però, s'ha de tenir en compte que tant el portal de sostre com el de terra **vindrà delimitat els vectors ymin i ymax i pels rasteritzats de paret** (figura 4.30).

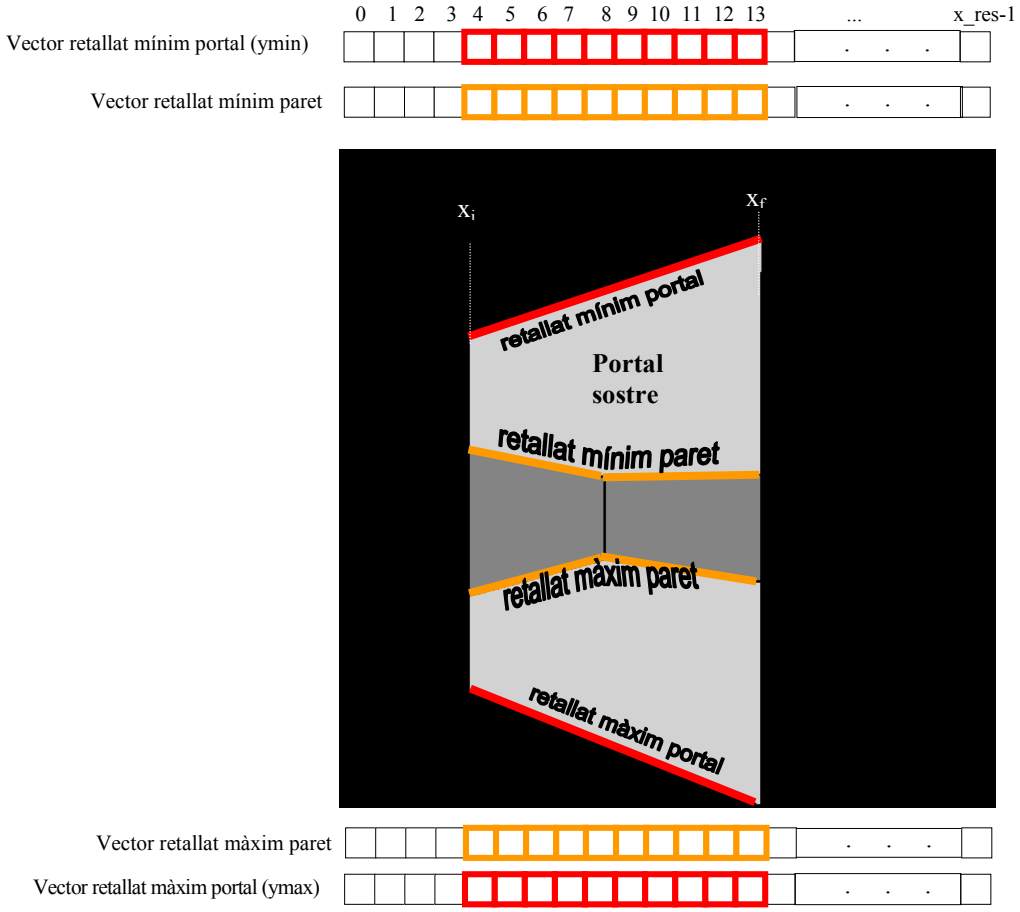


Figura 4.30

Abans de renderitzar el pla, cal obtenir tota la informació de retallat, es a dir (segons la figura 4.30):

- L'interval xi i xf.
- Obtenir els vectors Ymin/Ymax de paret segons convingui a l'interval xi-xf.
- Fer una copia dels vectors Ymin/Ymax de portal segons convingui a l'interval xi-xf.

L'obtenció dels vector retallat serà durant el bucle general de renderizat de línia de sector (funció *RenderitzarPartSector()*). Nosaltres, guardarem les cotes *y* de retallat de paret extrems de la seva renderització, i una copia de les cotes *y* de retallat de portal. També, serà important guardar l'interval x pel terra o pel sostre. Mostrem un exemple de com inicialitzaríem els valors de retallat pel sostre:

```
TUPLA tInformacioRenderitzatPla
        xi, xf: enter
        ymin, ymax: taula [0..x_res-1] de tipus enter
FTUPLA

Accio RenderitzarSector(e x1, x2, ymin, ymax, Jugador, Sector)
Var
        IRP_Sostre: tInformacioRenderitzatPla
        IRP_Terra  : tInformacioRenderitzatPla

        (...)
Fvar
                (...)

                { Inicialització de quadrat invàlid }

                 (IRP_Sostre.xi,  IRP_Sostre.xf) ← (x_res,  -1)


                (...)

                { Bucle general de renderitzat sector }
                mentre no fi fer

                        (...)

                fmentre


                si(Sector.Sostre = Portal) llavors { Portal, renderitzem sostre}
                        RenderitzarSector (IRP_Sostre.x1,
                                           IRP_Sostre.x2,
                                           IRP_Sostre.ymin,
                                           IRP_Sostre.ymin,
                                           Jugador,
                                           Sector.Sostre)
                fsi

FAccio
```

I en el següent llistat, es mostra la forma com actualitzaríem vector de retallat de sostre,

```
Accio RenderitzarPartSector(entrada ymin, ymax,  Parametres, LiniaSector, IRP_Sostre, IRP_Terra)
Var
        Y_sostre, Y_sc, Inc_Ydx_sostre, IncYdx_terra: real
        Y_terra, Y_tc, Inc_Ydx_tc, Inc_Ydx_sc: real
        y1, y2: enter
Fvar

        { Iniciatzació DDA i Càlcul dels seus increments }

          (...)

          Per x des de Parametres.x0 a Parametres.x1 fer
                si(LiniaSector és portal) llavors { Es renderitzar tires inferiors/susperiors de portal }

                        (...)

                altrament { Es renderitza paret }

                        (...)
                FSi

                { Actualizatcació inforemació sostre}


                {---------------------------- INICI ACTUALIZTACIÓ RETALLATS ----------------------------------------}

                {Actualiztació Sostre}
                si(IRP_Sostre.ymin[x] ≤ (y1-1)) llavors { Possible la renderització de pla sostre}

                        si(IRP_Sostre. xf = -1) llavors { Interval x inici/final  sostre a inicialitzar}

                                (IRP_Sostre.xi, IRP_Sostre.xi) ← (x, x)

                        altrament   { Només cal actualitza interval final }

                                IRP_Sostre.xf← x
                        fsi

                        IRP_Sostre.ymin[x]← ymin[x]
                        IRP_Sostre.ymax[x]← y1-1
                fsi

                { Actualització de terra, de manera similar...}

                (...)


                {-----------------------   FI ACTUALITZACIÓ RETALLATS   ----------------------------------}

                { Actualització DDAs}

                (...)

          Fper
FAcció
```

Capítol 5: Optimització

Havent estudiat el disseny del nostre motor, ja està preparat per ser portat a qualsevol maquina, i que, en el nostre cas, ha de ser portat a la consola de videojocs Gameboy Advance. Si penséssim portar el motor dissenyat a les plataformes PC actuals seria bastant fàcil ja que, avui en dia, tots ells disposen de CPU molt ràpides, targetes de vídeo 3D, alta capacitat de memòria RAM, etc.

Per contra, la Gameboy Advance (GBA) no té capabilitats 3D, és una màquina lenta en comparació les actuals (CPU ARM a 16'78 MHz), amb només 256 kbytes de RAM i, a més, sense coprocessador per operacions aritmètiques reals, etc (veure annex 2, per més informació sobre els aspectes genèrics d'aquesta màquina).

Figura 5.1 La GBA.

Per això, tota l'execució que comporta el procés de rendering serà efectuada per **software**. Per tant, és molt important que les tècniques estudiades als capítols anteriors siguin el més eficients possibles i, si cal, canviar la tècnica actual per d'altres equivalents més ràpides.

En aquest capítol, es començarà explicant tècniques d'aritmètiques òptimes, substitució de tècniques de render per altres de més ràpides. Finalment, adaptarem el nostre motor perquè s'executi en les òptimes condicions sota l'arquitectura de la consola GBA. Tot això per l'objectiu d'aquesta secció: aconseguir **processar les imatges del motor a més de 16 Fotogrames Per Segons (FPS).**

5.1 Optimització de operacions

La present secció té com objectiu donar alguns exemples de substitució de càlculs de arrels quadrades, divisions i funcions trigonomètriques, operacions que resulten molt costoses en temps d'execució.

5.1.1 Eliminació de la arrel quadrada

Per obtenir la distància des del primer vèrtex de la Línia de Intersecció (L.I) a la intersecció on hi intersecta el raig (coordenada u_i a la figura 5.2a), la primera operació que més senzilla que es pot aplicar és el càlcul d'una arrel quadrada (figura 5.2a).

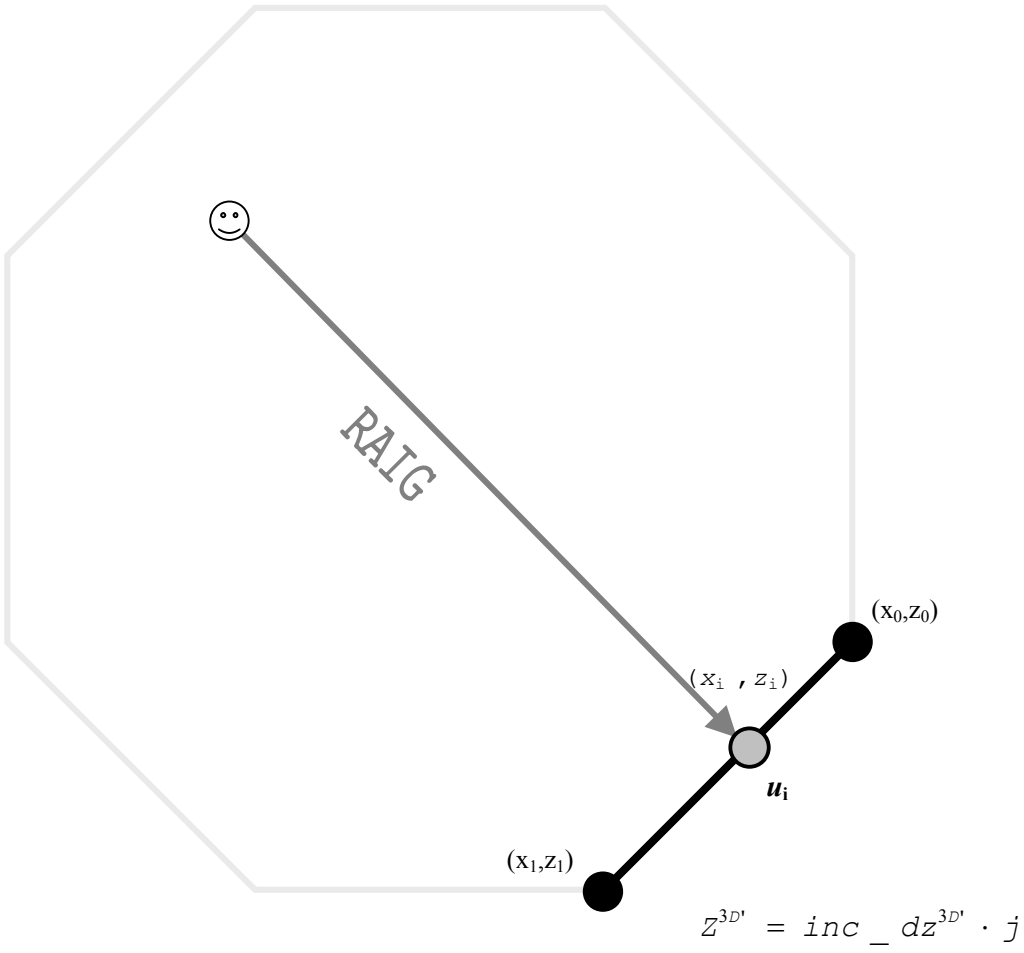


Figura 5.2a

Hi ha una altra manera més òptima de calcular la distància si coneixem l'angle de la línia (α_L).

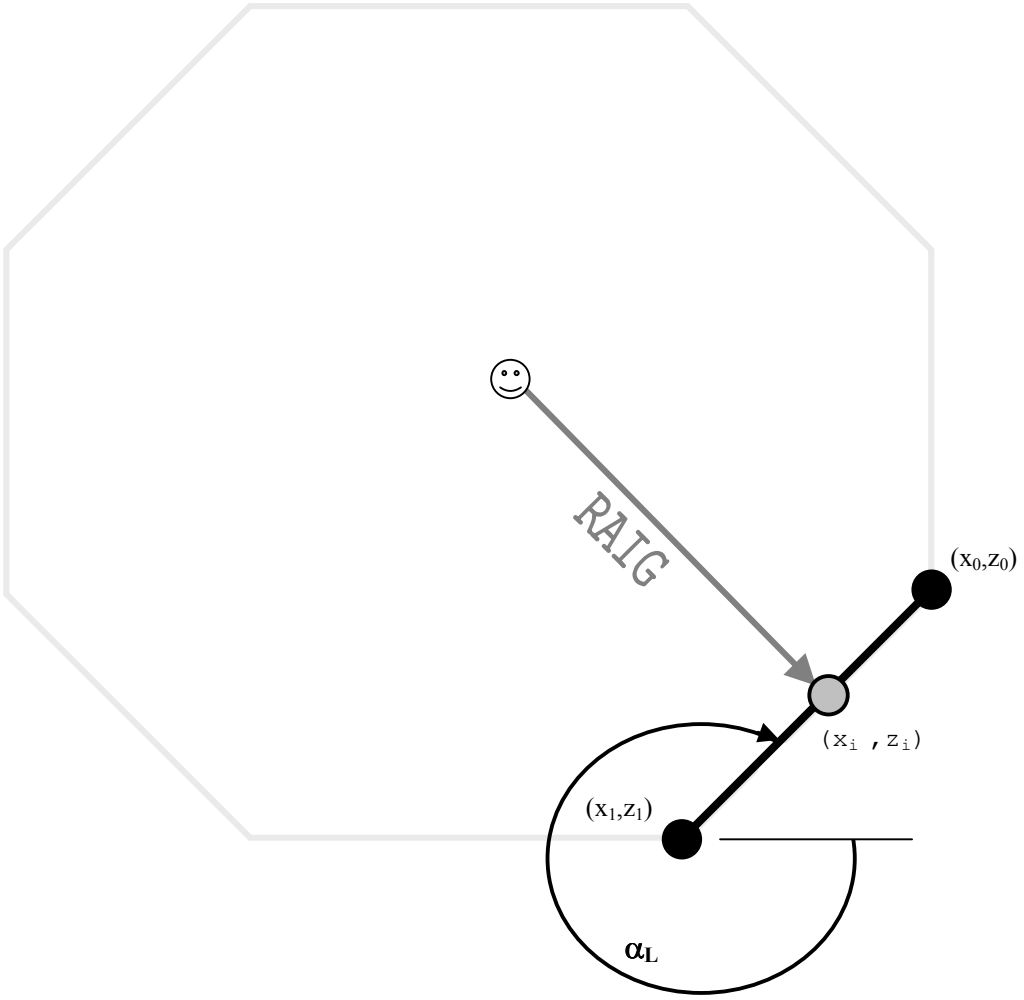


Figura 5.2b

Observant la figura 5.3b, si sabem l'angle de línia (α_L), aplicant trigonometria trobem una altra manera més òptima de calcular u_i.

$$u_i = \frac{(Z_i - Z_0)}{sin(\alpha_L)} \qquad \textit{Equació 5.1}$$

Cal observar que en la l'equació 5.1, s'hi divideix $sin(\alpha_L)$ que és una operació trigonomètrica i que el seu temps d'execució és considerable. Però com que $sin(\alpha_L)$ es constant, podem **precalcularlo** i ser guardat a l'estructura de línia, per exemple.

5.1.2 Optimització de càlculs

5.1.2.3 Eliminació de la divisió

Divisió inversa

Com al cas de l'equació 5.1, tenim una divisió però amb una peculiaritat: el denominador és constant. Podem aprofitar aquesta característica per calcular la seva inversa. Llavors l'operació que s'haurà de realitzar es simplificarà en una simple multiplicació.

$$\frac{\texttt{Numerador}}{\texttt{Constant}} = \texttt{Numerador} \cdot \frac{1}{\texttt{Constant}} = \texttt{Numerador} \cdot \texttt{Constant}'$$

Així mateix, per càlculs de DDAs i les interpolacions UZ requereix una divisió on el denominador és un enter (veure secció 3.3.2.2 del tema 3), i que, no sent més gran que l'ampla de pantalla, podem precalcular una taula divisions d'enters inverses d'aquest rang.

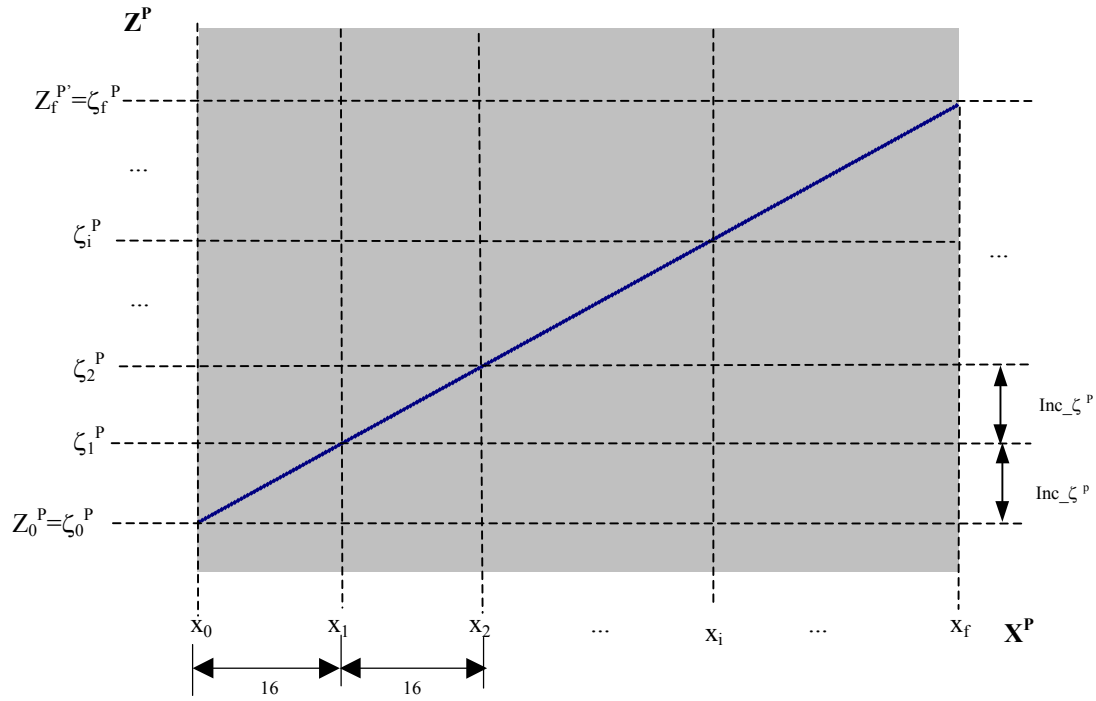
```
Divisio_inversa: taula [1,..,MAX_DIVISIO_ENTERA] de real

Divisio_inversa[1] ← 1 div 1
Divisio_inversa[2] ← 1 div 2
Divisio_inversa[3] ← 1 div 3
Divisio_inversa[4] ← 1 div 4

       ...

Divisio_inversa[MAX_DIVISIO_ENTERA] ← 1 div MAX_DIVISIO_ENTERA
```

- Interpolacions lineals parcials

Al punt 4 de la secció 2.1.5, es va estudiar l'algorisme interpolador que implica una multiplicació per atribut i una divisió a cada iteració.

[HCK97C] explica la manera de substituir aquest mètode per un de aproximat més ràpid i amb una pèrdua de qualitat inapreciable. Aquest mètode consisteix en partir la corba de perspectiva en interpolacions lineals cada *n* columnes. Normalment, són de longitud de 8 o 16 columnes. Amb la incorporació d'aquest mètode, només caldrà aplicar una suma per iteració durant les *n* columnes.

A les següents figures, mostrem un exemple de la partició de la corba en perspectiva de l'atribut Z (distància) en interpolacions lineals de **16 columnes** (figura 5.3b), també representant-lo a l'espai de pantalla equivalent (figura 5.3a).



$$i \in \left[1, \frac{x_f - x_0}{16}\right]$$

Figura 5.3a *Interpolació atribut Z a l'espai de pantalla entre x_0 a x_f, en particions de longitud 16.*

Observant la figura 5.3a, l'atribut ζ_i^P l'obtenim del increment constant inc_ζ^P, que es calcula com,

$$inc\_\zeta^P = \frac{z_f^P - z_0^P}{x_f - x_0} \cdot 16$$

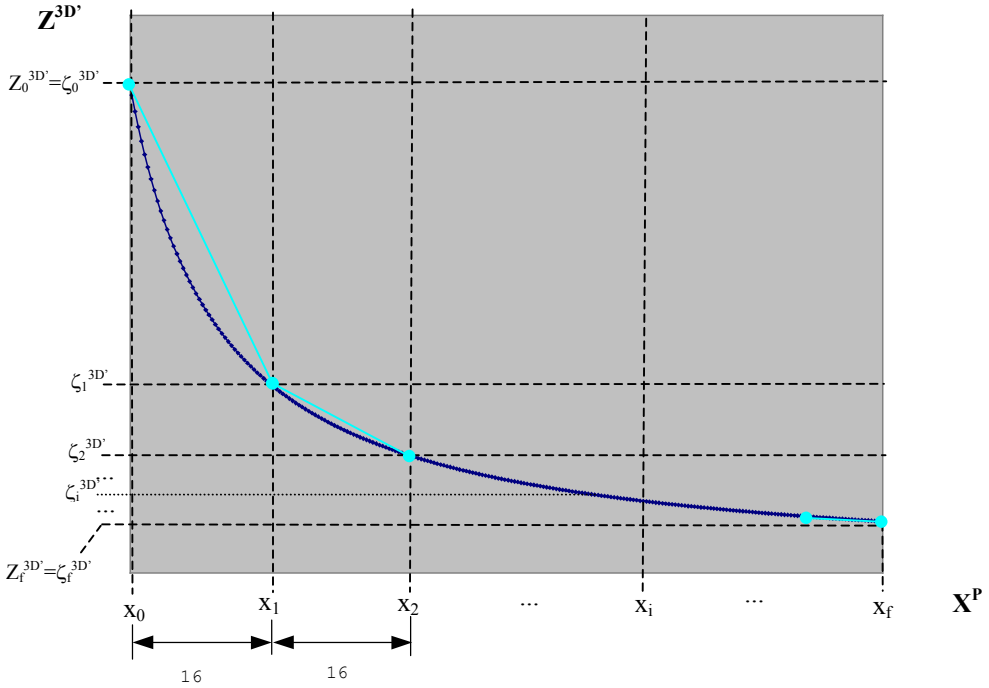


Figura 5.3b *Interpolació lineal de l'atribut Z a l'espai de món entre X0 a Xn, en particions de longitud 16.*

Observant la figura 5.3b, durant l'interval de 16 columnes, tot atribut $Z^{3D'}$ (línia blava) s'aconseguirà de la següent manera,

$$Z^{3D'} = \zeta_{i-1}^{3D'} + j \cdot \left(\frac{\zeta_i^{3D'} - \zeta_{i-1}^{3D'}}{16} \right)$$

$$i \in \left[1, \frac{x_f - x_0}{16}\right]$$

$$j \in [0, 16]$$

$$\zeta_i^{3D'} = \frac{1}{\zeta_i^{P'}}$$

Si $(x_f - x_0) \bmod 16 \neq 0$, voldrà dir que tindrem un resta iteracions no múltiples de 16 cap al final, fet que passarà quan $(x_f - x_{i-1}) < 16$. Durant aquest interval, $Z^{3D'}$ s'aconsegueix com,

$$Z^{3D'} = \zeta_{i-1}^{3D'} + j \cdot \left(\frac{\zeta_f^{3D'} - \zeta_{i-1}^{3D'}}{(x_f - x_0) \bmod 16} \right)$$

$$j \in [0, (x_f - x_0) \bmod 16]$$

Detallem l'ampliació de l'algorisme interpolador, ja explicat a la secció 2.1.5 (apartat 4), per aconseguir el proposat en aquest apartat,

```
Algorisme Interpolador_Amb_Correccio_Perspectiva_Particionat

    { Obtenció de les coordenades 3D (x0^3D',z0^3D'), (xf^3D',zf^3D') i els valors d'atributs (I0^3D',If^3D') }

    x0^P ← centre_X+d·(x0^3D'/z0^3D')
    xf^P ← centre_X+d·(xf^3D'/zf^3D')

    z0^P ← 1/z0^3D'
    zf^P ← 1/zf^3D'

    I0^P ← I0^3D'/z0^3D'
    If^P ← If^3D'/zf^3D'

    { derivades de la interpolació lineal del valors-Z i valors-I}

    inc_dzdX ← (zf^P – z0^P)/ (xf^P – x0^P)
    inc_dIdX ← (If^P – I0^P)/ (xf^P – x0^P)

    { Obtenció increments linials amb partició de 16 unitats }

    inc_dzdX_Part16 ← inc_dzdX mul 16
    inc_dIdZ_Part16 ← inc_dIdX mul 16

    { inicialització components esquerra (inicials) d'interpolació}

    (dz_esquerra, dI_esquerra) ← (z0^P , I0^P )
    (Z_esquerra , I_esquerra) ← (z0^3D' , I0^3D' )


    { inicialització components dreta d'interpolació}

    (dz_dreta, dI_dreta) ← (dz_esquerra + inc_dzdX_Part16, dI_esquerra + inc_dIdX_Part16)

  {Numero subsivisions de 16 unitats i el seu resta}
  Subdivisions ← (xf-x0+1) div 16
  RestaIteracions ← (xf-x0+1) mod 16

    x← x0^P
    fi ← FALS
    { Bucle principal de la interpolació lineal}
    mentre no fi fer

        { Calcul interpolacio lineal }
        si Subdivisions > 0 llavors
            Z_Dret ← 1/(dz_dret)
            Z ← Z_Esquerra

            I_Dret ← Z_Dret*(dI_dret)
            I ← I_Esquerra

            inc_dIdX_Lineal ← (I_Esquerra - I_Dret)/16
            inc_dzdX_Lineal ← (Z_esquerra- Z_dret)/16
            NIteracions ← 16
        altrament
            si RestaDivisions > 0 llavors
                Z_Dret ← 1/( zf^P)
                Z ← Z_Esquerra

                I_Dret ← Z_Dret*( If^P)
                I ← I_Esquerra

                inc_dIdX_Lineal ← (I_Esquerra -I_Dret)/ RestaDivisions
                inc_dzdX_Lineal ← (Z_esquerra-Z_dret)/ RestaDivisions
                NIteracions ← RestaIteracions

            altrament { No hi ha més iteracions }
                NIteracions ← 0
            Fsi
    Fsi

    Si (NIteracions ≠0) llavors

        per j des de 0 a NIteracions-1 fer

            { Processar info}

            I ← I + inc_dIdX_Lineal
            Z← Z + inc_dzdX_Lineal
            x ← x + 1
        Fper

        Z_Esquerra ← Z_Dret
        I_Esquerra ← I_Dret
        dz_dret ←dz_dret + inc_dzdX_Part16
        dI_dret ← dI_dret + inc_dIdX_Part16
     fsi

    si (Subdivisions ≤ 0) llavors

        Fi ← CERT
    Altrament
        Subdivisions ← Subdivisions – 1
    Fsi
fmentre
falgorisme
```

- Creació de LUTs

Les funcions trigonomètriques (***sin()***, ***cos(),tan()***) són operacions molt lentes en temps d'execució. Com totes les que s'han utilitzat fins ara van en funció de l'angle de jugador, i aquest, està enumerat de [0..360) en increments de 1º, podem crear les taules precalculades (Look Up Tables –LUTs-) respectius resultats taules de longitud 360.

```
Const
       MAX_LUT = 360
fConst


Sin: Taula [1,..,MAX_LUT] de real
Cos: Taula [1,..,MAX_LUT] de real
Tan: Taula [1,..,MAX_LUT] de real


Accio CrearLUT_SinCos()

       Per i des de 1 a MAX_LUT fer
              Sin[i] ← sin(i)
              Cos[i] ← cos(i)

              Si i = 90 o i = 270 llavors   { Evitem excepció }
                     Tan[i] ← 0
              Altrament                     { Podem precalcular}
                     Tan[i] ← tan(i)
              FSi
       Fper
FAccio
```

Llistat 5.2

També, podem crear la LUT dels pendents de raig comentada a la secció 2.1.4 (equació 2.4),

```
Pendent: Taula [0,..,x_res-1] de real


Accio CrearLUT_Pendent()

       Per i des de 0 a x_res-1 fer
              Pendent[i] ← (i+0.5-x_centre)/ d
       Fper
FAccio
```

Llistat 5.3

5.1.3 Cerca més òptima de la L.I

Si recordem, a la secció 3.3.2 és va explicar el procés per determinar la línia de intersecció (L.I) i la respectiva intersecció.

No obstant, hem de observar que s'ha de fer el costós càlcul de t per cada línia vàlida, determinat pel producte vectorial. Existeix una manera més òptima de trobar la L.I.

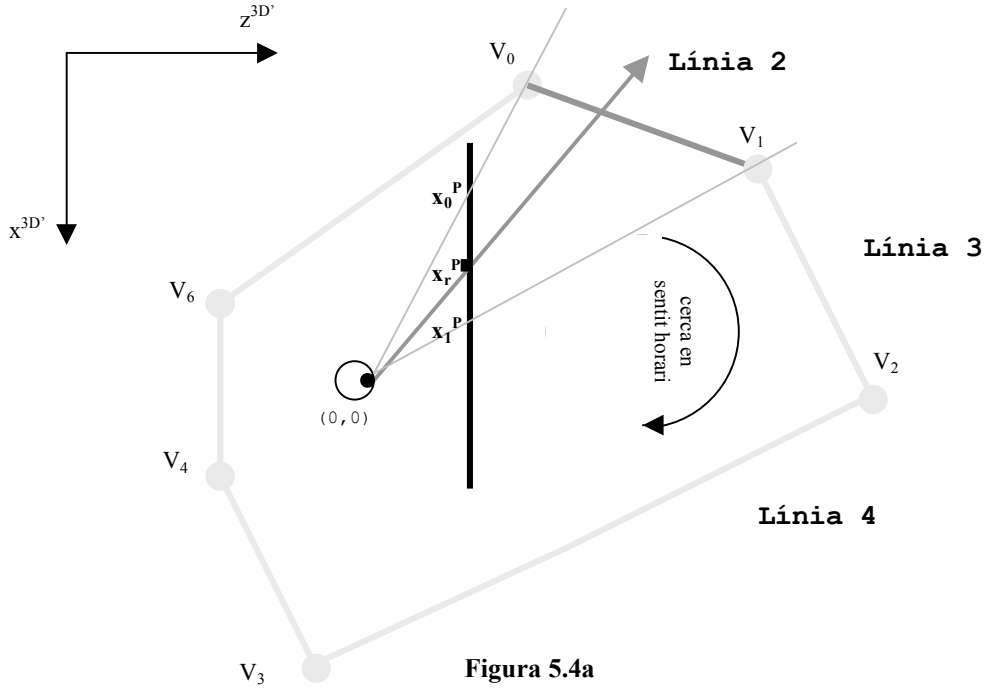


Figura 5.4a

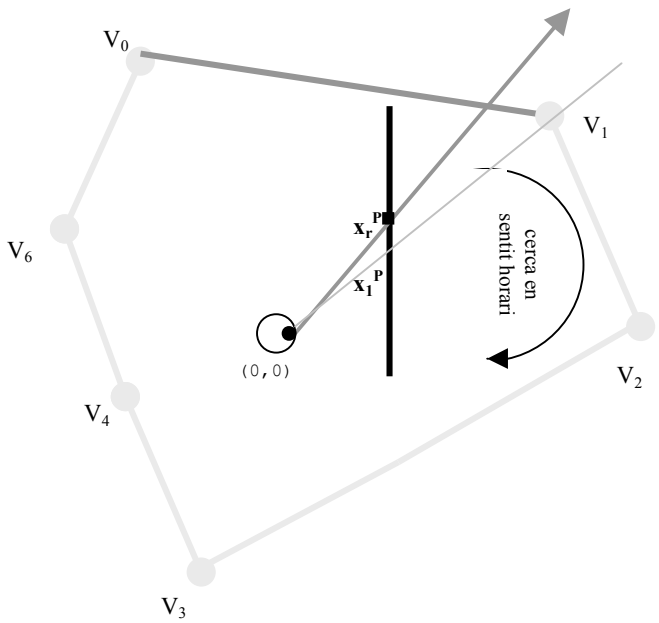


Figura 5.4b

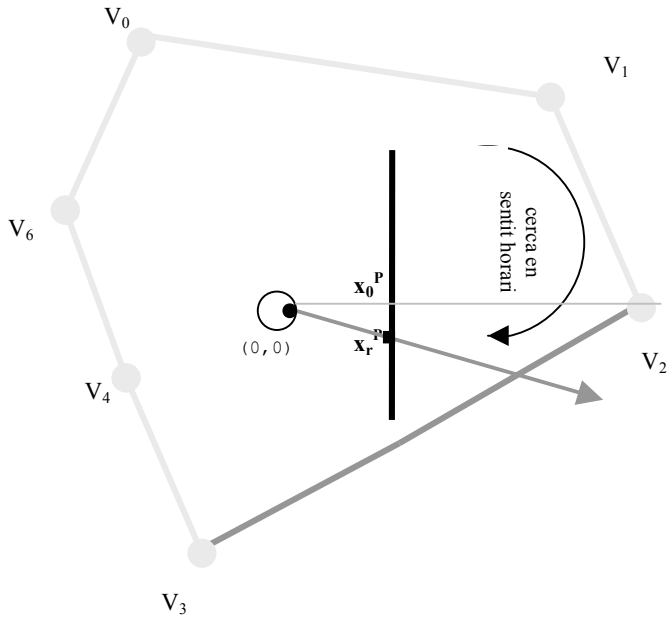


Figura 5.4c

Observant a la figura 5.4a, si és llança un raig per la columna x_r^P la línia de intersecció (**línia 2**) la trobem quan $x_0^P \leq x_r^P$ i $x_r^P < x_1^P$. Així doncs, si la orientació de línia és vàlida només caldrà fer una projecció per iteració.

Veiem el codi modificat respecte l'algorisme L.I del llistat 3.2 (secció 3.3.2), per observar la millora explicada en aquesta secció,

```
Algorisme LiniaInterseccio

        interseccio ← FALS
        x0p         ← ProjectarVertex(V0)
        i           ← 0

        mentre (i < n_vertexs) i no (interseccio) fer
                Si ( ( ξ̄i  x  d⃗ ) ≤ 0) llavors

                        x1p ←ProjectarVertex(Vi+1)

                        Si (x0p ≤ xrp < x1p){ S'ha trobat línia intersecció}

                                interseccio ← CERT

                        Fsi


                Fsi

                i ← i+1
                x0p ← x1p

        Fmentre
    Falgorisme
```

Llistat 5.4

- **Excepcions**

Cal mencionar dos casos que cal tenir en compte a l'hora de fer la nova cerca de L.I.

A la figura 5.4b, hi ha una L.I correcte però el primer vèrtex (V_0) no és projectable perquè es troba darrera del observador (Z < 0). De manera semblant, a la figura 5.4c, tenim la L.I on el segon vèrtex (V_3) es troba també darrera de l'observador. Doncs, senzillament caldrà modificar la funció del llistat 5.4a per comprovar si és L.I quan algun dels seus vèrtexs es trobi darrera del observador. A continuació mostrem la modificació necessària respecte el llistat 5.4.

```
Algorisme LiniaInterseccio

        interseccio ← FALS
        i           ← 0

        mentre (i < n_vertexs) i no (interseccio) fer
                Si ( ( ξ_i  x  d ) ≤ 0) llavors
                        X0_esquerra_raig ← V0_esta_a_la_esquerra_raig (Vi,xr)
                        X1_dreta_raig    ← V1_esta_a_la_dreta_raig (Vi+1,xr)

                        Si (X0_esquerra_raig i X1_dreta_raig){ S'ha trobat línia intersecció}

                                interseccio ← CERT

                        Fsi
                Fsi
                i ← i+1
        Fmentre
    Falgorisme
```

A continuació mostrem la funció que comprova si el vèrtex es troba projectat a l'esquerra del raig llançat per la columna x_r,

```
funció V0_esta_a_la_esquerra_raig(entrada V0: tVertex; xr^P:enter) retorna boolea
{ PRE: V0 esta transformat a l'espai d'ull }

    si(V0.Z > 0) llavors { Vèrtex projectable }

        x0^P ← Projeccio(V0)

        Si(x0^P ≤ xr^P) llavors   V0_esta_a_la_esquerra_raig ← CERT
        Altrament                 V0_esta_a_la_esquerra_raig ← FALS

    altrament { Vèrtex 0  no projectable, però assumim que es projecta a l'esquerra}
            V0_esta_a_la_esquerra_raig ← CERT
    fsi
ffunció
```

A continuació mostrem la funció que comprova si el vèrtex es troba projectat a la dreta del raig llançat per la columna x_r,

```
funció V1_esta_a_la_dreta_raig(entrada V1: tVertex; xr^P:enter) retorna boolea
{ PRE: V1 esta transformat a l'espai d'ull }

    si(V1.Z > 0) llavors { Vèrtex projectable }

        x1^P ← CentreX + d*(V1.X/V1.Z)

        Si(x1^P < xr^P) llavors  V1_esta_a_la_esquerra_raig ← CERT
        Altrament                V1_esta_a_la_esquerra_raig ← FALS

    altrament { Vèrtex 1  no projectable, però assumim que es projecta a dreta}
            V1_esta_a_la_dreta_raig ← CERT
    fsi
ffunció
```

- **Cerca encara més òptima**

Segons el l'algorisme de cerca de L.I vist al llistat 5.5a, es requereix fer dos projeccions per-iteració. Recordem que el càlcul de projecció X implica el cost d'una divisió (veure equació 2.2, secció 2.1.3). No obstant, nosaltres ens interessa avaluar la igualtat $\mathbf{x_0^P \leq x_r^P < x_1^P}$ per saber si és L.I.

Si desenvolupem la igualtat,

$$x_0^P \quad \leq \quad x_r^P \quad < \quad x_1^P$$

$$\left(\text{Centre\_x}^P + d \cdot \frac{x_0^{3D'}}{z_0^{3D'}}\right) \leq \left(\text{Centre\_x}^P + d \cdot \frac{x_r^{3D'}}{z_r^{3D'}}\right) < \left(\text{Centre\_x}^P + d \cdot \frac{x_1^{3D'}}{z_1^{3D'}}\right)$$

Simplificat queda com,

$$\left(\frac{x_0^{3D'}}{z_0^{3D'}} \leq \frac{x_r^{3D'}}{z_r^{3D'}} < \frac{x_1^{3D'}}{z_1^{3D'}}\right) \qquad \textit{Equació 5.2}$$

Vist a la secció 2.1.4 (equació 2.4), sabem que el pendent de raig és equivalent a,

$$\frac{x_r^{3D'}}{z_r^{3D'}} = \frac{(x_r^P + 0.5) - x_{centre}}{d} \qquad \textit{Equació 5.3}$$

Com ja s'ha vist a la secció anterior (creació de LUTs) el podem obtenir precalculat (Pendent[xr]). Pel que l'equació 5.3 es equivalent a,

$$\frac{(x_r^P + 0.5) - x_{centre}}{d} = \text{Pendent}\left[x_r^P\right] \qquad \textit{Equació 5.4}$$

Substituint queda l'equació 5.4 per 5.2 queda,

$$\left(\frac{x_0^{3D'}}{z_0^{3D'}} \leq \texttt{Pendent}[x_r] < \frac{x_1^{3D'}}{z_1^{3D'}} \right)$$

Finalment, per fer el test de L.I cal avaluar,

$$\boxed{\left(x_0^{3D'} \leq \texttt{Pendent}[x_r] \cdot z_0^{3D'}\right) \wedge \left(\texttt{Pendent}[x_r] \cdot z_1^{3D'} < x_1^{3D'}\right)} \quad \textit{Equació 5.5}$$

Observem que hem reduït el càlculs a una multiplicació.

5.2 Renderització per franges horitzontals

A l'annex 3 s'explica l'estudi del *mode 7*, un mode gràfic configurat que es capaç de suportar la consola GBA. Aquest mode gràfic, **es capaç de renderitzar plans per hardware**, un gran avantatge si es pogués combinar aquesta tècnica al nostre motor. Si ho aconseguíssim, no caldria fer cap càlcul per software, ja que s'encarregaria de fer-ho la maquinaria de la consola.

Però, tal com s'ha estudiat a l'annex 3, el *mode 7* treballa sota un mode gràfic rajola transformables (modes 1 o 2) i el nostre render treballa sota un mode gràfic de mapa de bits (pintat per-pixel), per lo que no ha estat possible la seva adaptació. Per això, s'ha tingut de fer el procés de renderització de plans per software.

Si recordem al llistat de codi 3.3 (secció 3.3.4) vam veure que, el cost per-pixel important que presenta el bucle general de **renderitzat de pla per franges verticals**, que es resumeix en **4 multiplicacions i 2 divisions per-pixel, un important cost de temps**. No obstant, al llistat a3.1 de la secció a3.4.5 (annex 3), es va estudiar que **si el renderitzat d'un pla es processa per franges horitzontals el cost per-pixel és de només 2 sumes**.

Es per això, en aquesta secció s'ha fet un estudi per compatibilitzar la renderització del sostre i terra a franges horitzontals.

Ho aconseguirem a partir dels següents passos:

1. Obtenció de retallats de portal i paret.
2. Assignat de cotes *x*.
3. Mapeig sostre i terra.
4. Excepció, algorisme de preprocessat

1. Obtenció retallats de portals i paret

Sabent els retallat superior/ inferior de portal i el retallat superior/inferior de paret un cop acabada la renderització del sector (figura 5.6).

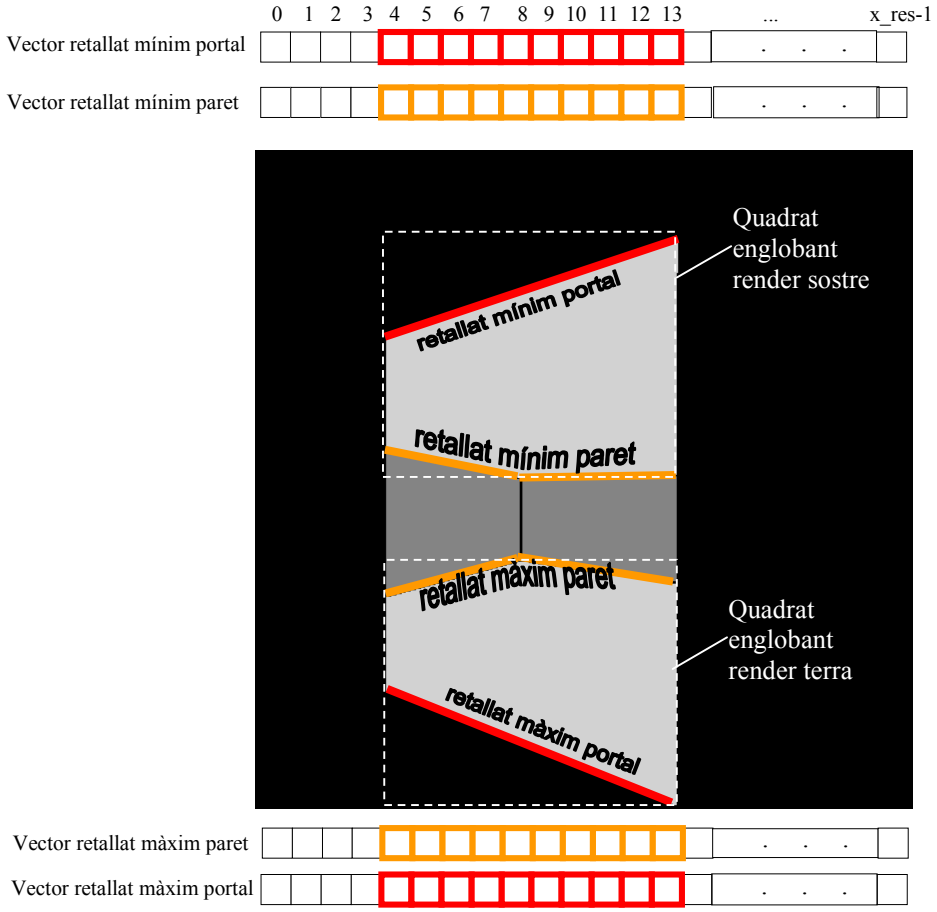


Figura 5.6

L'obtenció dels vector retallat serà durant el bucle general de renderizat de línia de sector (funció *RenderitzarPartSector()*). Nosaltres, guardarem les cotes *y* de retallat de paret extrems de la seva renderització, i les cotes *y* de retallat del portal actual. També serà important guardar els quadrats englobant pel terra i sostre que, ja veurem, seran necessaris per la posterior renderització del pla. L'estructura, inicialització i el recull de informació de retallat, és mostra al següent llistat:

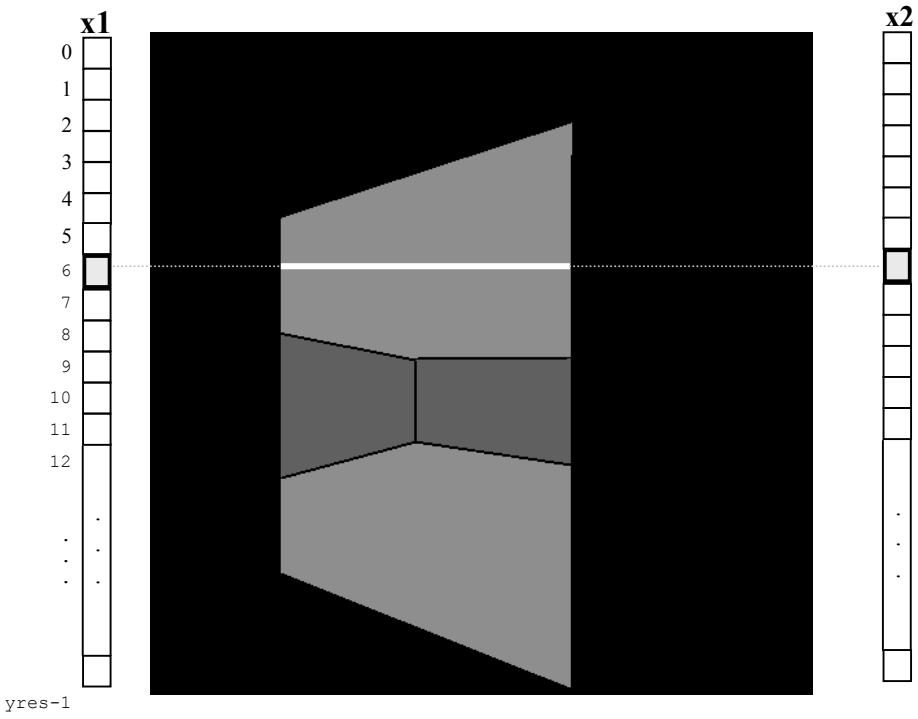
```
TUPLA tInformacioRenderitzatPla
        Quadrat_Sostre: tQuadrat
        ymin_retallat_sostre, ymax_retallat_sostre: taula [0..x_res-1] de tipus enter

        Quadrat_Terra: tQuadrat
        ymin_retallat_terra, ymax_retallat_terra: taula [0..x_res-1] de tipus enter
FTUPLA

Accio RenderitzarSector()
Var
        IRP: TinformaciorenderitzatPla
        (...)
Fvar
                (...)

                { Inicialització de quadrat invàlid }

                (IRP.Quadrat_Sostre.x1,  IRP. Quadrat_Terra.x1) ← (x_res,  x_res)
                (IRP.Quadrat_Sostre.x2,  IRP. Quadrat_Terra.x2) ← (-1, -1)
                (IRP.Quadrat_Sostre.y1,  IRP. Quadrat_Terra.y1) ← (y_res,  y_res)
                (IRP.Quadrat_Sostre.y2,  IRP. Quadrat_Terra.y2) ←  (-1 , -1)


                (...)

                { Bucle general de renderitzat sector }
                mentre no fi fer

                        (...)

                fmentre

                { Finalment, havent acabada la renderització de sector, toca renderirzar el pla de terra i sostre, per franges horitzontals}
                RenderitzacioPla(IRP)

FAccio
```

I en el següent llistat, es mostra la forma com actualitzaríem els 4 vectors de retallat,

```
Accio RenderitzarPartSector(entrada Parametres, LiniaSector, IRP)
Var
        Y_sostre, Y_sc, Inc_Ydx_sostre, IncYdx_terra: real
        Y_terra, Y_tc, Inc_Ydx_tc, Inc_Ydx_sc: real
        y1, y2: enter
Fvar

        { Iniciatzació DDA i Càlcul dels seus increments }

          (...)

          Per x des de Parametres.x0 a Parametres.x1 fer
                si(LiniaSector és portal) llavors { Es renderitzar tires inferiors/susperiors de portal }

                        (...)

                altrament { Es renderitza paret }

                        (...)
                FSi

                { Actualizatcació inforemació sostre}


                {---------------------------- INICI ACTUALIZTACIÓ RETALLATS ----------------------------------------}

                {Actualiztació Sostre}
                si(y_retallat_min_portal[x] ≤ (y_sostre-1)) llavors { Possible la renderització de pla sostre}

                        si(IRP. Quadrat_Sostre.x2 = -1) llavors { Interval x inici/final  sostre a inicialitzar}

                                (IRP.Quadrat_Sostre.x1, IRP.Quadrat_Sostre.x2) ← (x, x)

                        altrament  { Només cal actualitza interval final }

                                IRP.Quadrat_Sostre.x2← x
                        fsi

                        IRP.ymin_retallat_sostre[x]← y_retallat_min_portal[x]
                        IRP.ymax_retallat_sostre[x]← y_sostre-1
                        IRP.Quadrat_Sostre.y1  ← min(IRP.Quadrat_Sostre.y1, y_retallat_min_portal[x] )
                        IRP.Quadrat_Sostre.y2  ← max(IRP.Quadrat_Sostre.y2, (y_sostre-1))
                fsi

                {Actualiztació Terra}
                si((y_terra+1) ≤ y_max_retallat_portal[x]) llavors { Possible la renderització de pla terra}

                        si(IRP.Quadrat_Terra.x2 = -1) llavors { Interval x inici/final  terra a inicialitzar}

                                (IRP.Quadrat_Terra.x2, IRP.Quadrat_Terra.x2) ← (x, x)

                        altrament  { Només cal actualitza interval final }

                                IRP.Quadrat_Terra.x2 ← x
                        fsi

                        IRP.ymin_retallat_ terra[x]← y_terra +1
                        IRP.ymax_retallat_terra[x]← y_retallat_max_portal[x]
                        IRP.Quadrat_Sostre.y1  ← min(IRP.Quadrat_Sostre.y1, (y_terra+1))
                        IRP.Quadrat_Sostre.y2  ← max(IRP.Quadrat_Sostre.y2, y_retallat_max_portal[x] )

                fsi

                {----------------------  FI ACTUALITZACIÓ RETALLATS  ----------------------------------}

                { Actualització DDAs}

                (...)

          Fper
FAcció
```

2. Assignat cotes x

Un cop ja disposem dels 4 vectors de retallat, ja podem a explicar l'algorisme que s'encarregarà de assignar 2 vectors més, indispensables abans de fer el mapeig de sostre o terra.



Els 2 vectors, que els anomenarem x1 i x2 respectivament, hi contindrà la informació de inici/fi de cada franja de pla a renderitzar (figura 5.7). Nosaltres, hem de crear un algorisme que sigui capaç de omplir degudament aquests vectors *xs* a partir dels vectors de retallat obtinguts durant el renderitzat de sector.

La tècnica per assignar els valors *xs* de cada vector es basa coneixent la diferència de altures entre la cota *y* retallat actual i cota *y* retallat anterior. A les següents figures mostrem tots els possibles casos que ens podem trobar i l'interval de inicialització que caldria fer al vector adequat.

Figura 5.7

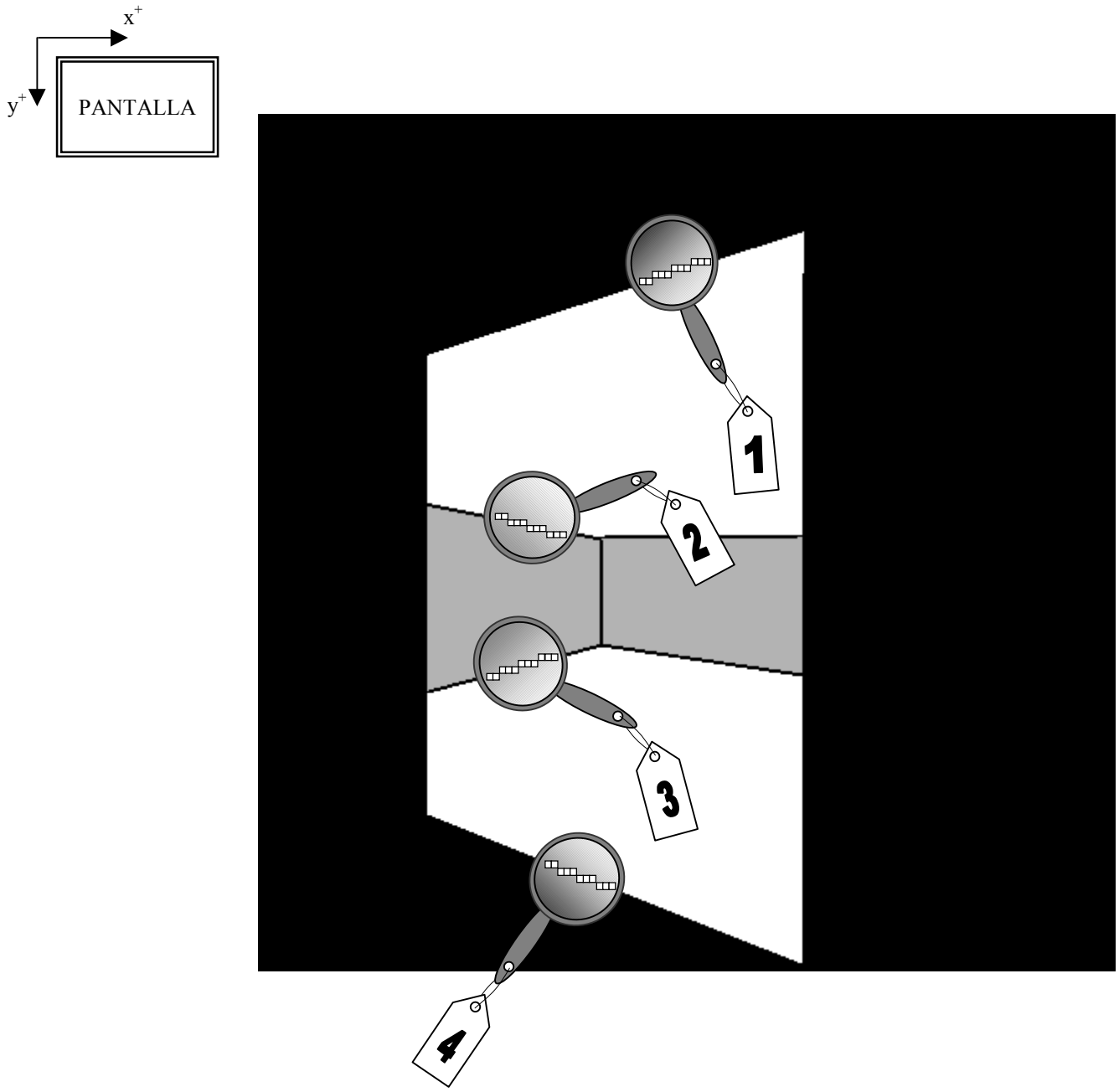
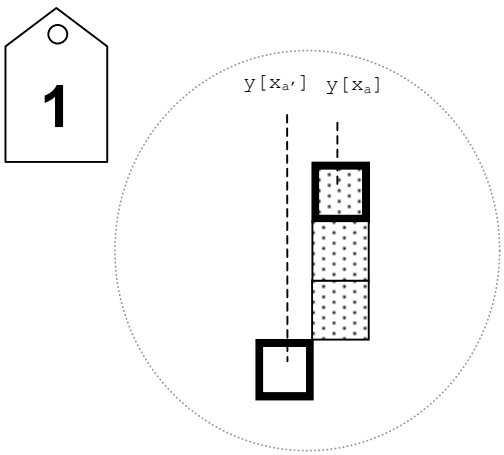
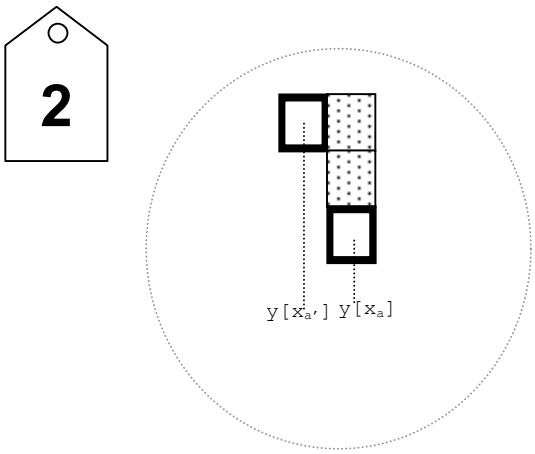


Figura 5.8 *A la figura, mostra l'exemple de tots els possibles casos en que cal assignar cotes x del vector x1, l'interval d'inicialització estarà en funció de la diferencia de cotes y dels vectors de retallat. Per informació més detallada, hem marcat amb etiquetes numerades del 1 al 4, que s'expliquen a la pàgina següent.*

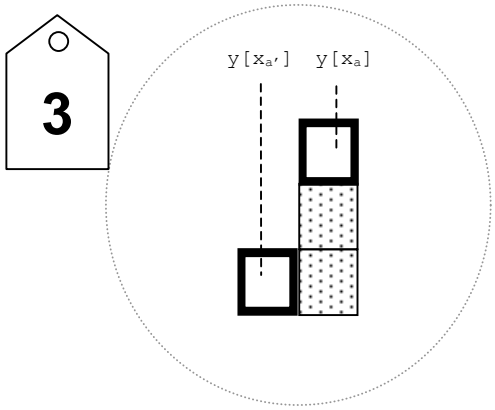
A la figura 5.8 mostra els 4 rasteritzats: 2 als extrems de paret i 2 més als extrems de portal. Segons el seu l'increment presentat en el increment del rasteritzat, **tots 4 hauran d'inicialitzar les cotes x1.** Tot seguit expliquem amb detall els tipus de rasteritzat i el procés de inicialització segons l'etiqueta de la lupa vist a la figura 5.8.



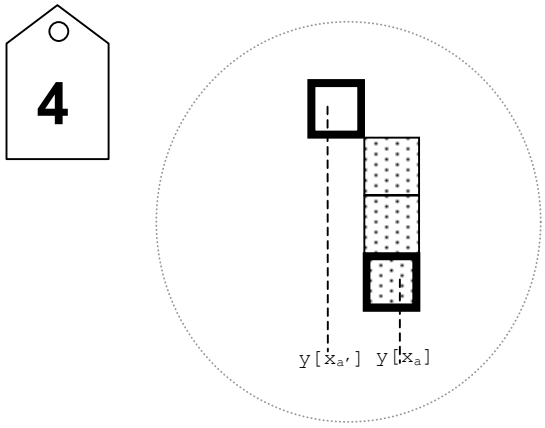
Aquest cas, el rasteritzat mínim de portal (retallat mínim de sostre) és decreixent per el que cal assignar les cotes x1. El procediment és el següent: quan la cota *y* de la columna x anterior (y[$x_{a'}$]) sigui més gran que l'actual(y[x_a]), assignarem el valor x_a al vector **x1** a l'interval ∈ [y[$x_{a'}$]-1,y[x_a]]



Aquest cas, el rasteritzat mínim de paret (retallat màxim de sostre) és creixent per el que cal assignar les cotes x1. El procediment és el següent: quan la cota *y* de la columna x anterior (y[$x_{a'}$]) sigui més petit que l'actual(y[x_a]), assignarem el valor x_a al vector **x1** a l'interval ∈ [y[$x_{a'}$],y[x_a]-1]



Aquest cas, el rasteritzat màxim de paret (retallat mínim en terra) és decreixent per el que cal assignar les cotes x1. El procediment és el següent: quan la cota *y* de la columna x anterior (y[$x_{a'}$]) sigui més gran que l'actual (y[x_a]), assignarem el valor x_a en el vector **x1** a l'interval ∈ [y[$x_{a'}$]-1,y[x_a]+1]



Aquest cas, el rasteritzat màxim de portal (retallat màxim en terra) és decreixent per el que cal assignar les cotes x1. El procediment és el següent: quan la cota *y* de la columna x anterior (y[$x_{a'}$]) sigui més petit que l'actual (y[x_a]), assignarem la columna x_a al vector **x1** a l'interval ∈ [y[$x_{a'}$]+1,y[x_a]]


Assignat cota *x* inicial


Lectura cota *y* vector retallat.

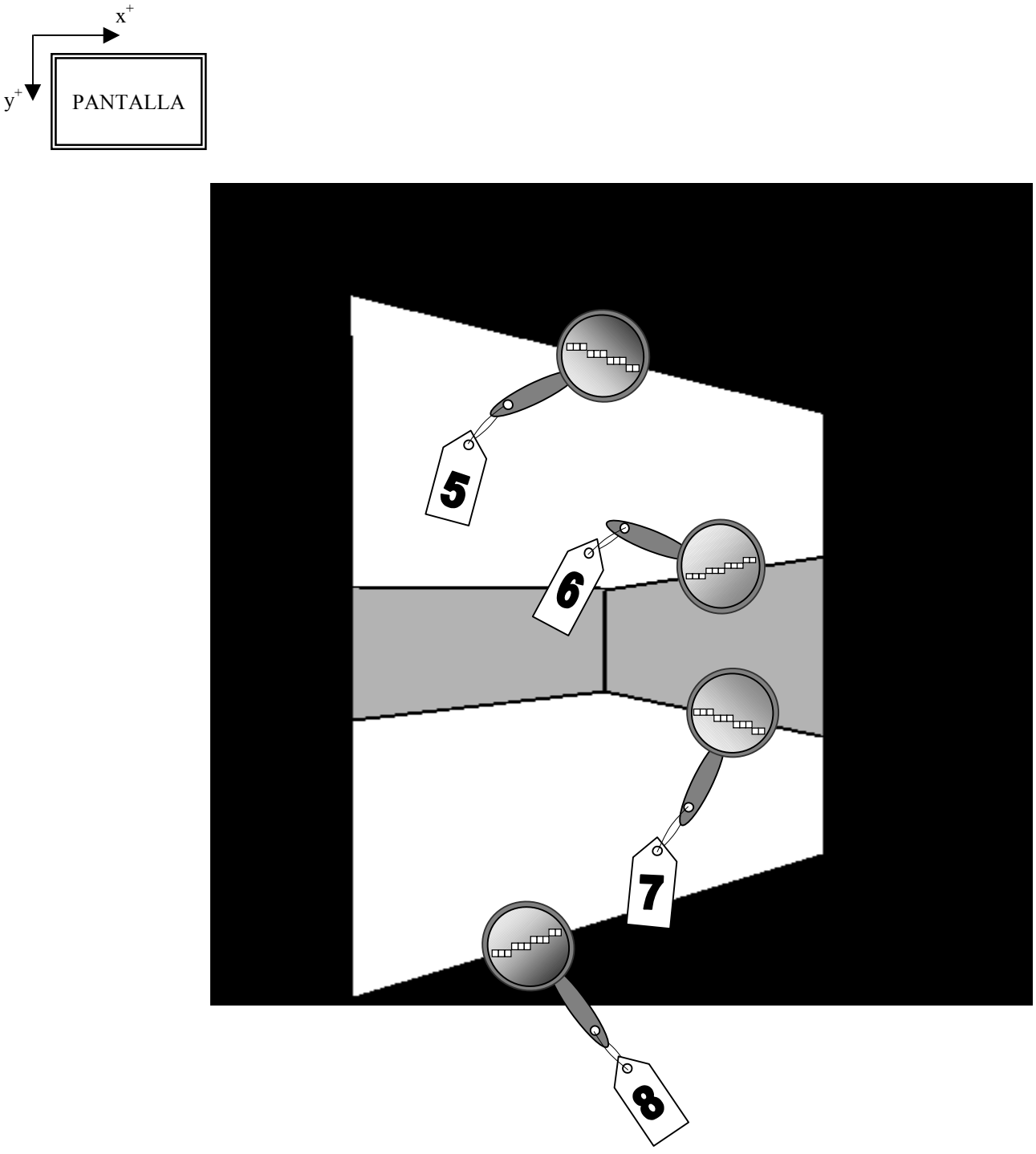
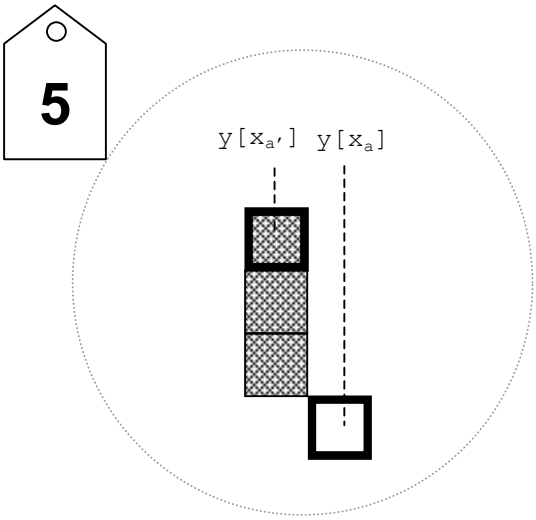
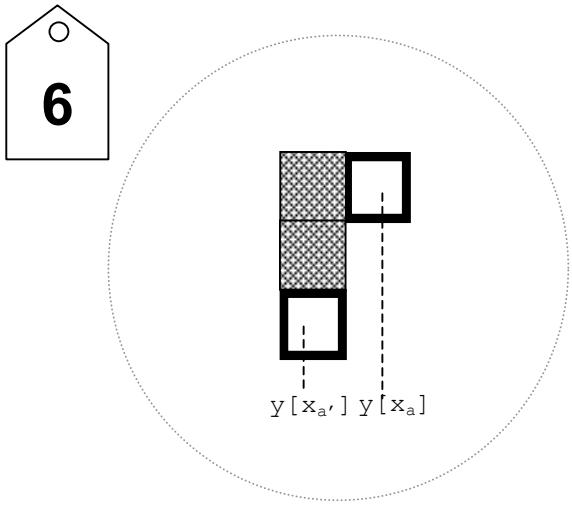


Figura 5.9 *A la figura, mostra l'exemple de tots els possibles casos en que cal assignar cotes x del vector x2, l'interval d'inicialització estarà en funció de la diferencia de cotes y dels vectors de retallat. Per informació més detallada, hem marcat amb etiquetes numerades del 5 al 7, que s'expliquen a la pàgina següent.*

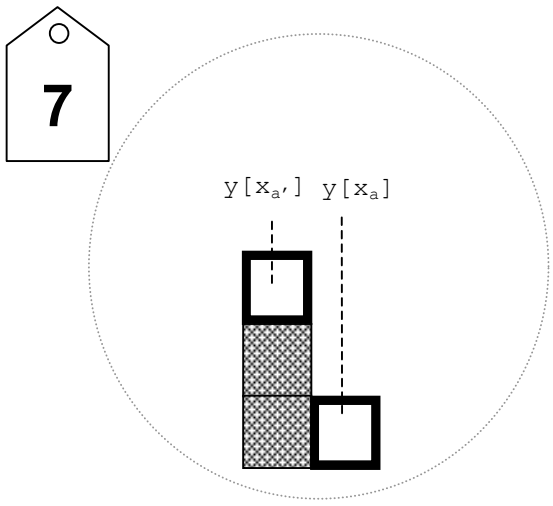
A la figura 5.9 mostra els 4 rasteritzats: 2 als extrems de paret i 2 més als extrems de portal. Segons el seu l'increment presentat en el increment del rasteritzat, **tots 4 hauran d'inicialitzar les cotes x2.** Tot seguit expliquem amb detall els tipus de rasteritzat i el procés de inicialització segons l'etiqueta de la lupa vist a la figura 5.9.



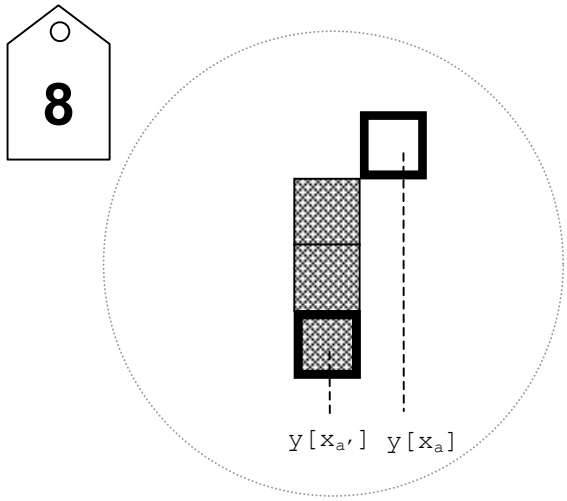
Aquest cas, el rasteritzat mínim de portal (retallat mínim en sostre) és creixent per el que cal assignar les cotes x2. El procediment és el següent: quan la cota *y* de la columna x anterior (y[$x_{a'}$]) sigui més petita que l'actual(y[x_a]), assignarem el valor $x_{a'}$ al vector **x2** a l'interval ∈ [y[$x_{a'}$],y[x_a]-1]



Aquest cas, el rasteritzat màxim de paret (retallat màxim de sostre) és decreixent per el que cal assignar les cotes x2. El procediment és el següent: quan la cota *y* de la columna x anterior (y[$x_{a'}$]) sigui més gran que l'actual(y[x_a]), assignarem el valor $x_{a'}$ al vector **x2** a l'interval ∈ [y[$x_{a'}$]-1,y[x_a]]



Pels valors de retallat màxim de paret (retallat mínim de terra) és creixent per el que cal assignar les cotes x2. El procediment és el següent: quan la cota *y* de la columna x anterior (y[$x_{a'}$]) sigui més petita que l'actual(y[x_a]), assignarem el valor $x_{a'}$ al vector **x2** a l'interval ∈ [y[$x_{a'}$]+1,y[x_a]]



Aquest cas, el rasteritzat màxim de portal (retallat màxim de terra) és decreixent per el que cal assignar les cotes x2. El procediment és el següent: quan la cota *y* de la columna x anterior (y[$x_{a'}$]) sigui més gran que l'actual(y[x_a]), assignarem el valor $x_{a'}$ al vector **x2** a l'interval ∈ [y[$x_{a'}$],y[x_a]-1]

Assignat cota *x* final

Lectura cota *y* vector retallat.

L'algorisme de assignat, començaria a processar a partir de l'interval x+1 del quadrat de render obtingut durant el bucle de renderitzat (veure figura 5.13). Podem observar a les figures 5.15a i 5.15b que els intervals de omplerts són idèntics, pels 4 tipus que hi ha. Així doncs, mostrem l'algorisme equivalent per l'obtenció de cotes *x* del sostre.

```
y1_sostre_anterior ← IRP_Sostre.y_retallat_min[IRP_Sostre.Quadrat.x1]
y2_sostre_anterior ← IRP_Sostre.y_retallat_max[IRP_Sostre.Quadrat.x2]


Per x des de IRP_Sostre.Quadrat.x1+1 a IRP_Sostre. Quadrat .x2 fer

        y1_sostre_actual ← IRP_Sostre.y_retallat_min[x]
        y2_sostre_actual ← IRP_Sostre. y_retallat_max [x]

        si y1_sostre_actual ≤ y2_sostre_actual  llavors {retallat vàlid}

                si(y1_sostre_actual < y1_sostre_anterior) llavors {assignació cotes x1 }

                        per y des de y1_sostre_actual a y1_sostre_anterior-1 fer
                                x1[y] ← x_actual
                        fper

                altrament {assignació cotes x2 }


                        per y des de y1_sostre_anterior a y1_sostre_actual fer {assignació cotes x2}
                                x2[y] ← x
                        fper

                fsi

                si(y2_sostre_anterior < y2_sostre_actual) llavors {assignació cotes x1 }

                        per y des de y2_sostre_anterior a y2_sostre_actual-1 fer
                                x1[y] ← x
                        fper

                altrament {assignació cotes x2 }


                        per y des de y1_sostre_actual a y2_sostre_anterior fer {assignació cotes x2 }
                                x2[y] ← x
                        fsi

                fsi

        fsi

        y1_sostre_anterior ← y1_sostre_actual
        y2_sostre_anterior ← y2_sostre_actual

         { Assignació de cotes x1, x2, segons el retallat de terra...}

        (...)

Fper
```

Llistat 5.10

3. Mapeig de pla per franges horitzontals

Un cop guardats els vectors x1, x2 ja podrem mapejar el sostre i terra per franges verticals de manera similar com s'explica al llistat s'explica en el llistat a3.1 (annex 3). Mostrem l'algorisme de pintat pel sostre,

```
Per y desde Quadrat_Sostre.y1 a Quadrat_Sostre.y2 fer

        { Calcular components pla (x_mapa, dx_mapa, ect }
        Components ← CalculComponentsPla(y,Jugador)

        { renderitzar franja}
        PintaTiraHoritzontal(x1[y], x2[y], Components)

FPer
```

4. Excepció, algorisme de processat

Hem pogut veure la manera de inicialitzar les cotes *x*. Aquest procés vist, requereix ser convex per pintar el sostre-terra per franges horitzontals a la forma del seu retallat.

Pot donar-se la situació de trobar-nos amb uns quants sectors transparents, tal com podem observar a la figura 5.10, els quals trenquen la convexitat del sostre.

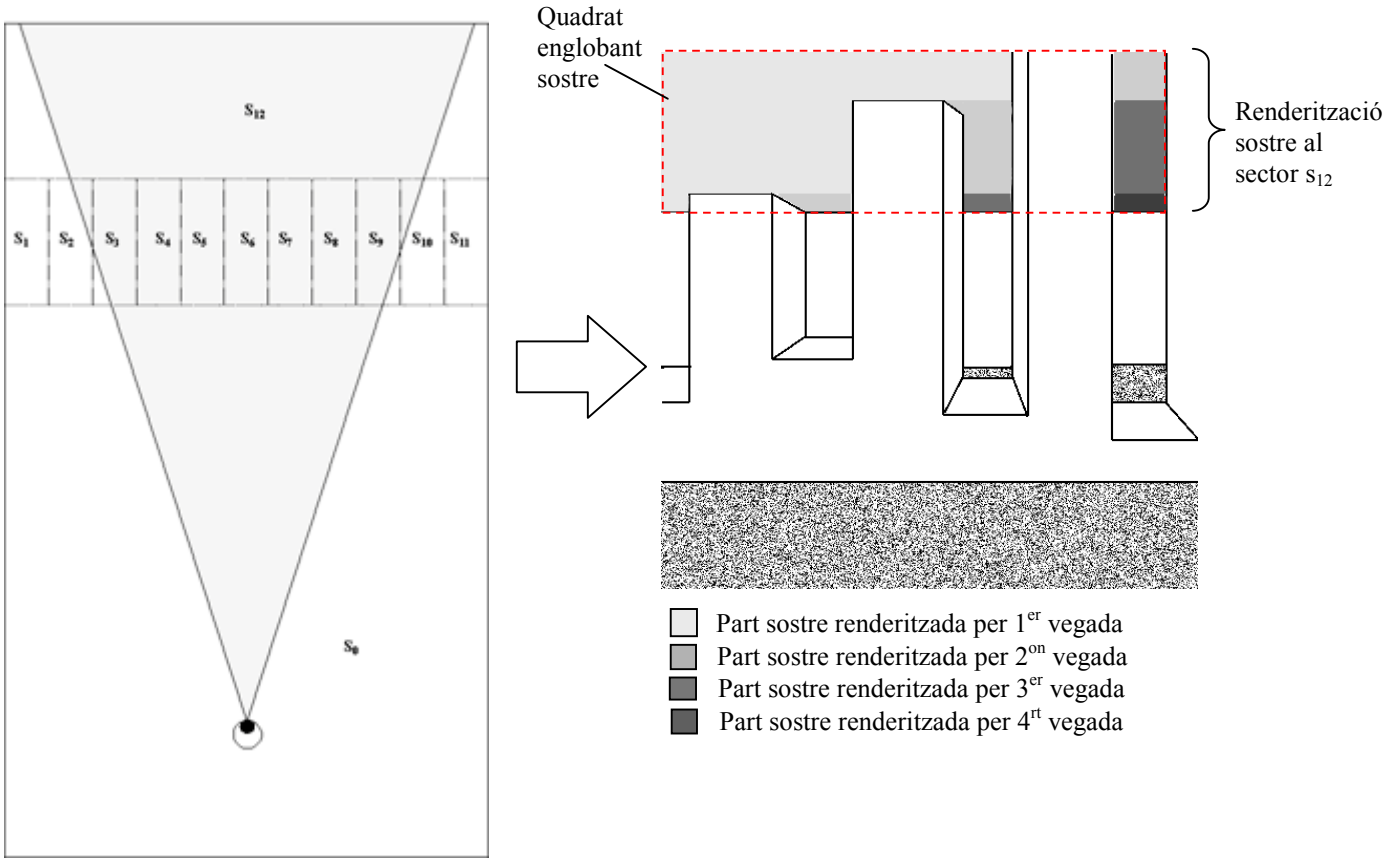


Figura 5.10

Fixem-nos en l'exemple de la figura 5.10. Els sectors transparents, provoquen la renderització parcial del sostre del sector s_{12}, fins una 4a vegada. Partint de l'algorisme de obtenció de cotes estudiat al llistat 5.10, hauríem de modificar-lo per tal de solventar aquesta excepció.

Tal com es té l'algorisme del llistat, s'assigna cotes x1 i les cotes x2 segons quan troba els intervals *y* actuals/anteriors. Quan s'assigna les cotes x2, vol dir que s'ha acabat de rasteritzar la franja i esta a punt per ser pintada, per consegüent, podem pintar directament en ves de assignar la cota del vector x2. Amb això volem dir que hem trobat una manera de rasteritzar tot el sector sense apenes modificar el llistat 5.10, i estalviar-nos temps i espai (no cal el vector x2). Quan s'hagi pintat la franja a una cota *y*, si més endavant hi ha de tornar a pintar-ne una altre la es tornaria a fer la inicialització de *x1* i acabant un altre cop segons els intervals y actuals/anteriors. En canvi, cal parar especial, atenció quant el sector transparent talla per complert el sostre, per el qual es tindrà que tornar a inicialitzar els vectors x1 quan tornem a tenir un interval de retallat vàlid.

Veiem la modificació introduïda al llistat 5.10, segons el que hem explicat,

```
y1_sostre_anterior ← IRP_Sostre.y_retallat_min[IRP_Sostre.Quadrat.x1]
y2_sostre_anterior ← IRP_Sostre.y_retallat_max[IRP_Sostre.Quadrat.x2]

Y_retallat_anterior_valid ← y1_sostre_anterior <= y2_sostre_anterior

Per x des de IRP_Sostre.Quadrat.x1+1 a IRP_Sostre.Quadrat.x2 fer

        y1_sostre_actual ← IRP_Sostre.y_retallat_min[x]
        y2_sostre_actual ← IRP_Sostre. y_retallat_max [x]

        si(y1_sostre_actual < y2_sostre_actual)llavor Y_retallat_actual_valid ← CERT
        altrament Y_retallat_actual_valid ← FALS

        si(actual_valid) i no (anterior_valid)llavors { Cal inicialitzar cotes x1...}

                per y desde y1_actual a y2_actual+1 fer
                        x1[y] ← x_actual
                fper

       altrament { Actualitzem o mapejem...}

           si no(actual_valid) i (anterior_valid) llavors { Cal fer el mapeig...}

               per y desde y1_anterior fins y ≤ y2_anterior fer { Pintar tires...}

                   Components ← CalculComponentsPla(y,Jugador)
                   PintarTiraHoritzontal(x1[y], x_actual-1,y, Components)
               Fper

           altrament { ok, avaluem si i només si, l'actual és vàlid...}

                si y1_sostre_actual ≤ y2_sostre_actual  llavors {retallat vàlid}

                        si(y1_sostre_actual < y1_sostre_anterior) llavors {assignació cotes x1 }

                                per y des de y1_sostre_actual a y1_sostre_anterior-1 fer
                                        x1[y] ← x_actual
                                fper

                        altrament { pintar...}

                                per y des de y1_sostre_anterior a y1_sostre_actual fer {pinta tira}
                                        Components ← CalculComponentsPla(y,Jugador)
                                        PintarTiraHoritzontal (x1[y], x_actual-1, Components)
                                fper
                fsi

                si(y2_sostre_anterior < y2_sostre_actual) llavors {assignació cotes x1 }

                        per y des de y2_sostre_anterior a y2_sostre_actual-1 fer
                                x1[y] ← x_actual
                        fper

                altrament { pintat }

                        per y des de y1_sostre_actual a y2_sostre_anterior fer {assignació cotes x2 }
                                Components ← CalculComponentsPla(y,Jugador)
                                PintarTiraHoritzontal (x1[y], x2[y], Components)
                        fsi
                fsi
        altrament {pintat}

                per y des de y1_sostre_anterior a y2_sostre_anterior fer {assignació cotes x2 + pintat}
                        Components ← CalculComponentsPla(y,Jugador)
                        PintarTiraHoritzontal (x1[y], x2[y], Components)
                fper
        fsi

        y1_sostre_anterior ← y1_sostre_actual
        y2_sostre_anterior ← y2_sostre_actual

         { Assignació de cotes x1, x2, segons el retallat de terra...}

        (...)

Fper
```

Llistat 5.11

5.3 Optimitzacions per a la GBA

Arribats aquest i últim punt d'aquest treball, només queda optimitzar el procés de render per la consola GameBoy Advance (GBA). Moltes de les tècniques que a continuació s'estudien estan optimitzades per tal d'executar el render a la màxima velocitat, sota les restriccions que imposa l'arquitectura d'aquesta consola.

5.3.1 Iniciació a la arquitectura de la GBA

• Els modes de processador GBA

Tal com s'explica a la secció a1.2.3 del annex 1, el processador de la GBA pot operar en 4 tipus de estat dels quals 2 són propis als programes GBA que són: *ARM i THUMB*. Segons els avantatges i desavantatges de cadascun que anomentats en la secció a1.2.3, el mode *THUMB* és molt efectiu per memòria de codi (16 bits) que el mode *ARM*, ja que és molt més ràpid (aproximadament 160 % respecte ARM) i es menys dens (fins un 60% respecte l'ARM).

Es per això que el que es recomana es compilar les nostres aplicacions en format THUMB.

• A memòria interna més ràpid

La GBA disposa de memòria interna, la qual un programa opera dins d'ella, s'executa a més velocitat ja que és una memòria integrada de 32 bits, 1 cicle en vers 3 cicles que és requereix, quan el codi de programa es troba a memòria externa (veure a2.3). Donat que no podem col·locar tot el nostre codi de programa a memòria interna (només disposem de 32 kbytes), hem avaluat les funcions de més cost per col·locar-les a memòria interna, mitjançant una eina de temps i la formula explicades a l'annex a2.1.4. La llibreria HAM conté la directiva de compilador *FUNC_IN_IWRAM* per incloure funcions de codi C a memòria interna. Caldria ser declarada com es veu al llistat C següent,

```
FUNC_IN_IWRAM void function(void)
{
        ...
}
```

• Estats del processador

Com ja hem comentat el codi generat en estat THUMB és el més òptim que podem executar a la GBA en una memòria d'un bus de dades de 16 bits, i encara més si l'executem a memòria interna. No obstant, un codi ARM dins de la memòria interna executa una rutina **a la màxima velocitat** d'aquesta consola. Per això, el més recomanable és que les rutines que tinguin major cost, estiguin en format ARM i a memòria interna. La GBA pot operar amb ambdós modes, no simultanis, però si combinats mitjançant un salt especial de ensamblador (veure annex a0.4.3).

• Programació ensamblador

Per l'objectiu de optimitzar alguns processos, moltes parts del codi les haurem de codificar en ensamblador i, pel motiu que s'ha descrit anteriorment, en format ARM i a memòria interna perquè és la manera de fer corre un procés a la màxima velocitat sota la maquina GBA.

Però, ens podríem preguntar, si el nostre compilador no optimitzarà de manera adequada el nostre codi C. La resposta es si, però el compilador només pot genera tot el codi en THUMB o codi en ARM, no es pot personalitzar el mode de processador de cada mòdul C. Segons el recomanat, THUMB es el mode òptim pel compilador. Sabem que THUMB només disposa 7 registres a part, de que no s'opera amb prou rapidesa (16 bits), per això algunes de les nostres tasques crítiques a temps de execució, hauran de estar programades en ensamblador ARM.

5.3.1 Tècniques de aritmètica entera

5.3.1.1 Substitució de l'operació mòdul.

L'operació complementaria de la divisió (mòdul), es pot reduir a una simple operació aritmètica lògica si el divisor és potència de dos.

Per exemple, per seleccionar l'angle de jugador dins l'interval dels 360º. Després d'un increment d'angle caldria fer una costosa divisió,

Angle ← (Angle + increment_angle) **modul** 360

Però, si l'angle el codifiquem en un interval enumerat de 256 posicions ($0xFF_{|16}$), només caldria fer la operació lògica AND.

Angle ← (Angle + increment_angle) **AND** 0xFF

5.3.1.2 Eliminació de multiplicacions

La multiplicació entera la podem simplificar en simples desplaçaments aritmètics si el seu operant és potencia de dos. No obstant, **tota multiplicació per enters** la podem reduir en simples desplaçaments aritmètics combinat d'unes quantes restes o sumes.

Aquí tenim uns quants exemples,

```
x*1  → x
x*2  → x << 1
x*3  → x*(2+1) → x*2 + x   → (x << 1) + x
x*4  → x << 2
x*5  → x*(4+1) → x*4 + x   → (x << 2) + x
x*6  → x*(4+2) → x*4 + 2*x → (x << 2) + (x << 1)
x*7  → x*(4+3) → x*4 + 3*x → (x << 2) + (x << 1) + x
x*8  → x << 3
x*9  → x*(8+1) → x*8 + x   → (x << 3) + x
x*10 → x*(8+2) → x*8 + 2*x → (x << 3) + (x << 1)
x*11 → x*(8+3) → x*8 + 3*x → (x << 3) + (x << 1) + x
```

(...)

El símbols '<< ' representa l'operació en codi C de desplaçament aritmètic cap a l'esquerra.

5.3.1.3 Aritmètica Punt Fix

La GBA no disposa de coprocessador per treballar amb aritmètica real, per la qual cosa totes les variables i operacions amb reals **són emulades per software en temps d'execució**. Per una aplicació que requereixi interactivitat a la GBA, com el nostre cas, no es recomanable fer ús d'aquesta emulació d'aritmètica real. D'altra banda, si codifiquem l'aritmètica real en aritmètica entera podem accelerar el molt el procés. Aquest tipus de codificació entera s'anomena Aritmètica de Punt Fix (APF).

- **Representació APF 24.8**

Tal com s'explica a l'annex 2, la GBA disposa d'un processador de 32 bits per tant el numero real quedaria codificat en aquesta grandària. Nosaltres, partirem d'una APF 24.8, molt utilitzada en molts programes gràfics. Tal com ve definida, la APF 24.8 disposa de 24 bits per representa els nombres enters (-1bit reservat pel signe) i els 8 restants per representar els decimals.

Així doncs el rang enter d'un numero en APF 16.8 serà,

$$-2^{23}{}_{[24.8]} \leq \text{rang\_enter} \leq 2^{23}\text{-}1_{[24.8]}$$

i la mínima representació fraccional (o resolució fraccional) seria de,

$$resolució\_fraccional = \frac{1}{2^8}$$

- **Codificació de real a APF 24.8**

Per codificar qualsevol valor real en APF 24.8, només cal multiplicar-ho per 2^8, es a dir, multiplicat pel tamany que ocup en els bits de la part fraccional.

$$\begin{array}{ccc} f & \Re & \longrightarrow & \Re_{24.8} \\ & x & & x \cdot (2^8) \end{array}$$

- **Operacions APF 24.8**

Per suma i restar dos nombres APF tant sols cal aplicar la operació suma/resta entera.

$$A_{[24.8]} = B_{[24.8]} + C_{[24.8]}$$

$$A_{[24.8]} = B_{[24.8]} - C_{[24.8]}$$

Per la multiplicació, cal fer una rectificació després d'haver fet la multiplicació APF, per tornar a tenir un valor APF 24.8 vàlid. Posem l'exemple senzill de multiplicar dos 1s en format APF 24.8,

$$1_{[24.8]} * 1_{[24.8]} = (1 * 256) * (1 * 256) = \frac{(1 * 256) * (1 * 256)}{256} = 1 * 1 * 256 = 1 * 256$$

Per evitar la amplificació de la part fraccional, cal dividir el resultat entre 256

De manera similar passa pel cas de la divisió.

$$\frac{1_{[24.8]}}{1_{[24.8]}} = \frac{(1 * 256)}{(1 * 256)} = \frac{(1 * 256) * 256}{(1 * 256)} = \frac{1 * 256}{1} = 1 * 256$$

Per evitar simplificar la part fraccional, cal premultiplicar el numerador per 256.

- **Robustesa de la APF**

Fer us d'una APF dins un programa té l'avantatge de ser molt ràpida en temps d'execució, però el desavantatge de que **és una aritmètica no robusta**. Mostrem alguns problemes que hem tingut durant el desenvolupament d'aquest projecte.

Overflow

La APF 24.8, té bastant rang enter $\in$ [-16777216, 16777215] però, si no tenim cura de les operacions que es fan, podem tenir un error de overflow (per una multiplicació o una divisió, per exemple). Tal com s'explica a la secció A4.1.4, podem evitar aquests problemes senzillament canviant l'ordre de les operacions. És a dir si tenim una operació amb una multiplicació i una divisió, primer operant la divisió i llavors la multiplicació.

Insuficient resolució fraccional

- Mapeig pla

Fer us la codificació en APF 24.8, hi ha problema quan volem tenir més precisió fraccional, ja que per la configuració d'aquesta aritmètica no s'arriba a representar valors fraccionals per sota de $1/2^8 = 1/256 = 0{,}00390625$. Presentem un dels problemes que ens hem trobat molt durant el desenvolupament d'aquest treball, durant l'adaptació de la APF 24.8:

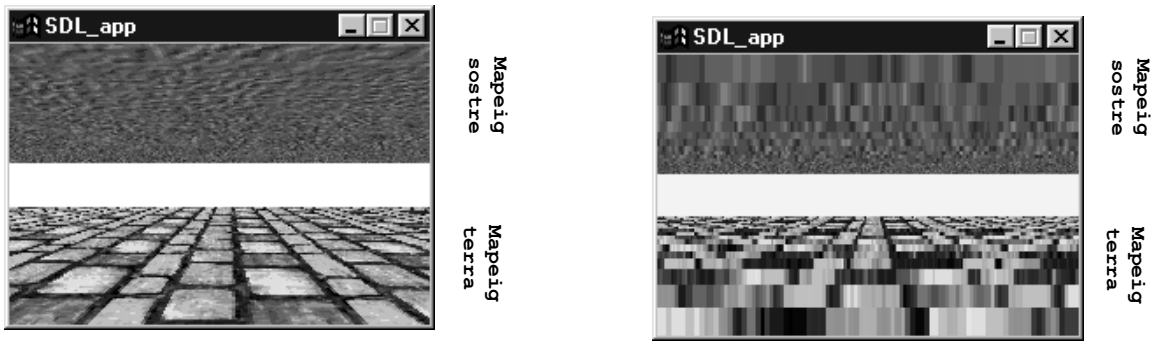
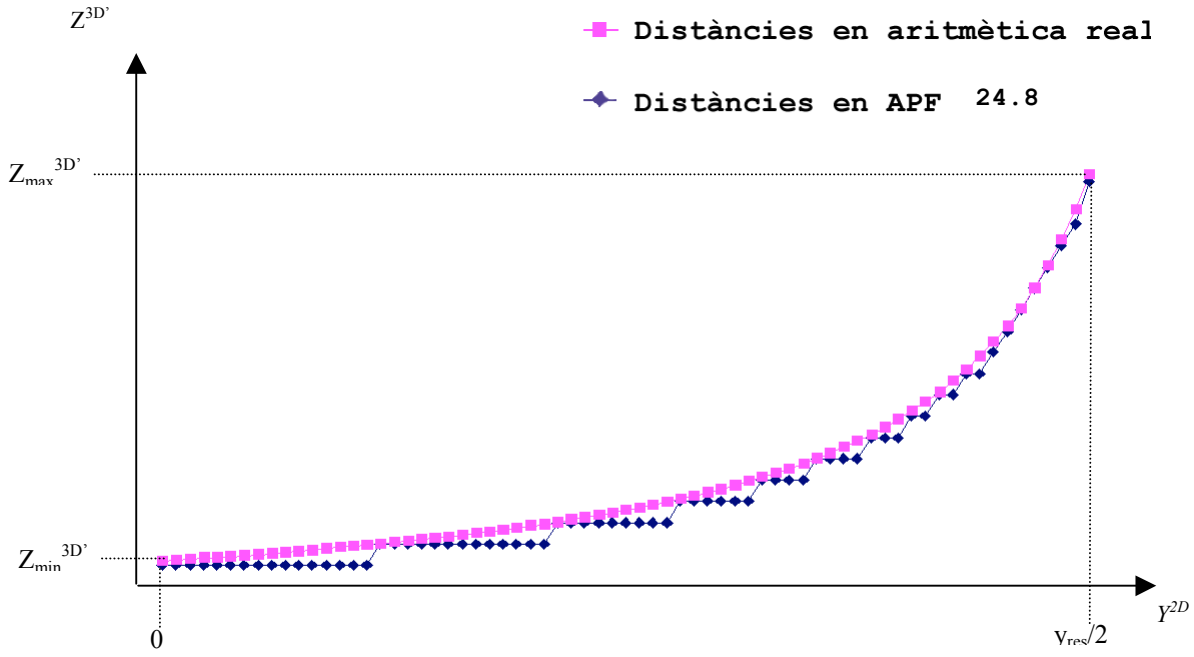


Figura 5.17 *Mapeig de sostre i terra sense distorsió (esquerra) i amb distorsió (dreta)*

A la figura 5.17 podem apreciar la distorsió del pla (tant en terra com en sostre) quan les franges que representen el pla estan a prop del observador però, quan es troben a una certa llunyania ja no hi ha distorsió. En aquesta mateixa situació de la figura 5.17, es van analitzar els càlculs implicats per en el mapeig de pla i vam veure que el valor distància ($z^{3D'}$) tenia problemes per arribar als valors reals. Mostrem la gràfica 5.1 de comparació amb dels valors real contra els valors APF 24.8.



Gràfica 5.1 *Comparació dels valors de distància pel mapeig del sostre real contra APF 24.8. Els valors s'han obtingut de la situació de la figura 5.17*

A la gràfica podem observar que valors de distància pel mapeig de sostre en $y^{P'} \in [0, y_{res}/2)$. Podem veure que la distància prop de l'observador, els càlculs obtinguts en APF 24.8 es manté, en gran part, invariant fins al proper valor representable. A mesura que es calculen distàncies més grans, la APF 24.8 comença a representa els valors igual que els reals.

El problema, com ja havíem comentat, es degut a la resolució que disposa la APF 24.8. Per solucionar-ho, cal ampliar la resolució fraccional, però a canvi de sacrificar bits de la part entera. Recordant a la equació 2.2,

$$Z_i^{3D'} = d \cdot \frac{Y^{3D'}}{Y^{P'}} \qquad \textit{Equació 5.5}$$

Podem observar en el càlcul de la distància ($Z_i^{3D'}$, equació 5.5), el denominador és enter i $\in [y_{res/2}, 0]$, per el cas del sostre. Com que $y^{P'}$ és enter, el podem multiplicar per la seva divisió inversa, tal com s'ha explicat a la secció 5.1.2.3, en format APF 16.16.

Com d és constant i $y^{3D'}$ no requereix part fraccional, pot estar representat per un enter i, així, fer una multiplicació entera en vés d'una operació APF.

$$z_i^{3D}=d_{[32.0]} \cdot y^{3D'}{}_{[32.0]} \cdot \texttt{divisio\_inversa}[y^{P'}{}_{[32.0]}]_{[16.16]} \qquad \textit{Equació 5.6}$$

Nota: subíndex *[32.0]* serveix per denotar que la variable tractada és un enter (no APF), per això no cal cap operació de tipus APF, ja que no altera el resultat final.

- Interpolacions Z

Les distancies durant la renderització de la part de sector les aconseguim indirectament a través d'una interpolació lineal a l'espai de pantalla (z^p), més ràpid que no pas llançar un raig per-columna (veure secció 2.1.5 del capítol 2). El problema de la interpolació z^p, és similar al cas del problema 2 (P2), es a dir, la insuficient disposició de resolució fraccional no ens permet aconseguir els valors reals. Sabem que, per interpolar Z a l'espai de pantalla cal fer la seva inversa (equació 5.7).

$$Z^P = \frac{1}{Z^{3D'}} \qquad \textit{Equació 5.7}$$

Com que $Z^P \in [0.5, MAX\_Z]$[5] , $Z^P \in [1/0.5, 1/MAX\_Z]$, es a dir, $Z^P \in [2, 0)$, per tant, podem representar-lo en una APF 8.24 és a dir, 8 bits per la part entera i 24 bits per la part fraccional, que assegurarà aconseguir una representació quasi igual a la real.

Per convertir Z^P en APF 8.24, només cal desplaçar 16 bits a l'esquerra, equivalent a multiplicar per 2^{16} (16384) el numerador, es a dir, el $1_{[24.8]}$ (equació 5.8).

$$Z_{[8.24]}{}^P = \frac{1_{[24.8]} \cdot 16384}{Z_{[24.8]}{}^{3D'}} \qquad \textit{Equació 5.8}$$

[5] Recordem que el mínim és 0.5 tal com es va determinar a la secció 4.2.3, que explica la gestió de les col·lisions entre observador (jugador) i línies.

MAX_Z, es el màxim establert per les dimensions de mapa, que en principi queda límitat per la capacitat entera de la APF. Com que la APF que s'esta utilitzant és al 24.8, llavors, MAX_Z seria de 2^{24}.

Operacions crítiques

Com anem dient des de el principi una APF no és tan robusta com una aritmètica real. Cal posar molta atenció a les operacions crítiques com, per exemple, la funció de projecció en X. Sabem que llançarem el raig per la columna, resultant de la projecció X, segons la fórmula que ja es va estudiar a la secció 2.1.3. La fórmula estudiada passada a APF 16.8 seria la següent,

$$x\_proj \leftarrow Centre\_x + CeilFX\_16\_8(DivFX\_16\_8(z^{3D'}_{[16.8]}\ ,\ x^{3D'}_{[16.8]}))$$

Una petita alteració del seu resultat, podria donar la columna de projecció errònia i, per consegüent, la no penetració del raig pel portal, no trobar la L.I i provocar la no renderització de sector (figura 5.19)

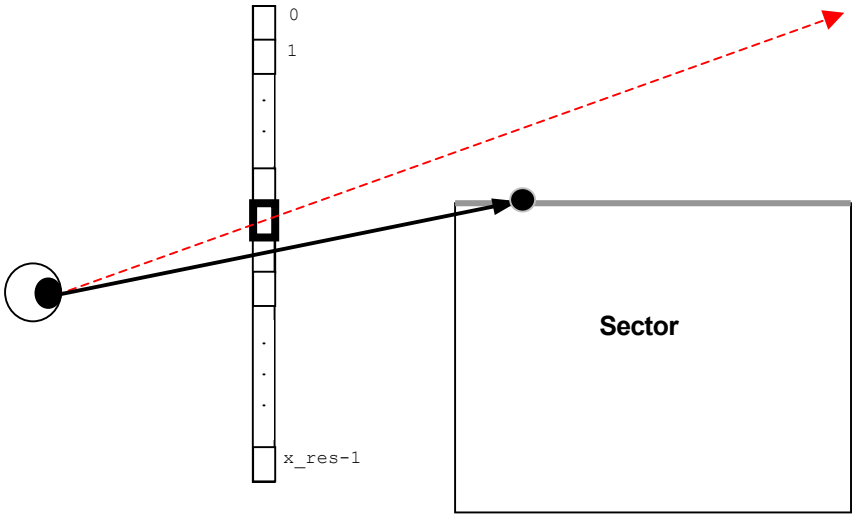


Figura 5.18 *Degut a la poca robustesa de la APF, s'ha donat la projecció equivocada (columna remarcada), pel qual el raig no intersectarà mai cap línia del sector (raig discontinuo). La bona, seria la marcada el raig negre (línia contínua).*

El problema principal està en que perdem precisió a la part fraccional a la divisió que es realitza. Tant el numerador, com el denominador estan en format APF, per tant no podem fer una operació entera com passava a l'apartat anterior. Com que, en aquest cas, volem aconseguir precisió a tot cost, caldria configurar la APF per què treballi en més bits fraccionals. A la pràctica, s'ha provat que l'us de la APF 16.16 (16 bits per la part entera i 16 per la part fraccional), aconsegueix la projecció correcta.

$$x\_proj \leftarrow Centre\_x + CeilFX\_16\_16(DivFX\_16\_16(z^{3D'}_{[16.8]} \cdot 2^8 \ , \ x^{3D'}_{[16.8]} \cdot 2^8))$$

La funció *divfx_16_16* representa la operació de punt fixa 16.16, per això a les entrades cal multiplicar les coordenades $XZ^{3D'}$ a 2^8, per tal ser convertides a format 16.16. La operació *ceilfx_16_16*, arrodoneix i entrega l'enter d'una variable en format 16.16, segons l'algorisme explicat a la secció 4.3 del capítol 4.

Distorisó de tira

També, durant el recorregut ens vam trobar amb el següent problema visual,

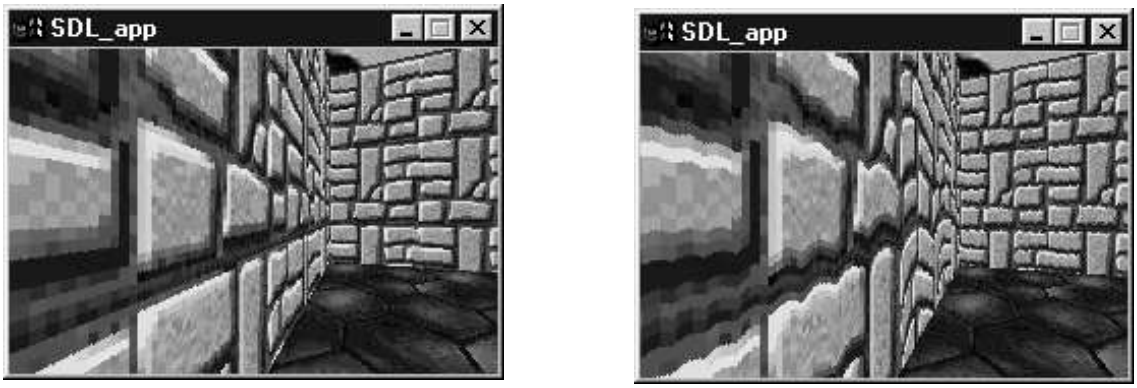


Figura 5.19a *Mapeig de paret sense distorció (esquerra) i amb distorsió (dreta)*

Podem observar a la imatge distorsionada (figura 5.19a dreta), que la paret presenta unes ones en el mapeig. El problema es troba al fer el càlcul inicial de V (v_i^P, figura 5.19b).

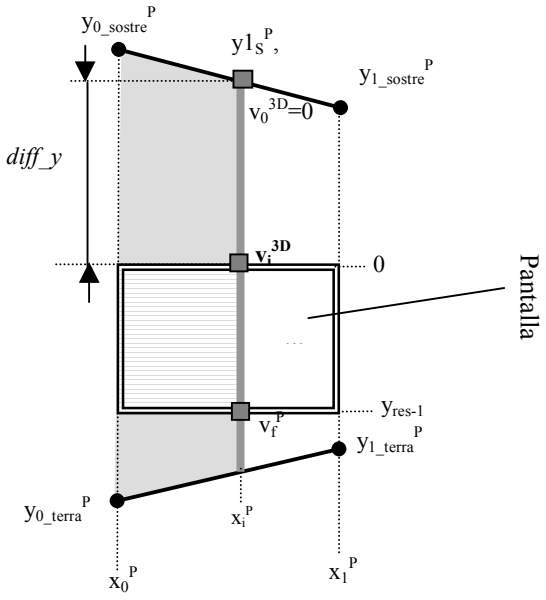


Figura 5.19b

Sabem que la coordenada inicial v (v_0^{3D}=0) pel pintat s'associa al rasteritzat superior de paret, pel que si el rasteritzat sobresurt del límit inferior del retallat, cal multiplicar l'escala per diff_y i, així, obtenir la v inicial (v_i^{3D})

$$v_i^{3D} \leftarrow diff\_y * Escalat\_v^{3D}$$

on el valor *escalat_v*^{3D} deriva de la distancia,

$$Escalat\_v^{3D} = \frac{Distancia}{d}$$

Segons les interpolacions lineals de perspectiva ja explicades a la secció 5.1.2.3 d'aquest capítol, que realitza la interpolació lineal UZ lineal en 16 columnes de renderització, Z^{3D} l'obtenim a partir d'aquestes interpolacions. Aquí tenim el problema, perquè obtenim l'atribut $Z^{3D'}$ amb un petit error i *diff_y* el que fa es amplificar-lo, provocant aquest onatge en el mapeig. Per això, com més gran sigui *diff_y*, més es notarà aquest onatge.

Una solució podria ser associar v^{3D} a $y_m^{3D'}$(centre y^{3D} respecte l'espai d'ull, secció 2.1.3), així, es calcularà v_i^{3D} des del centre de projecció, per consegüent, el valor màxim *diff_y* serà centre_y^P, tal com podem observar a la figura 5.19c.

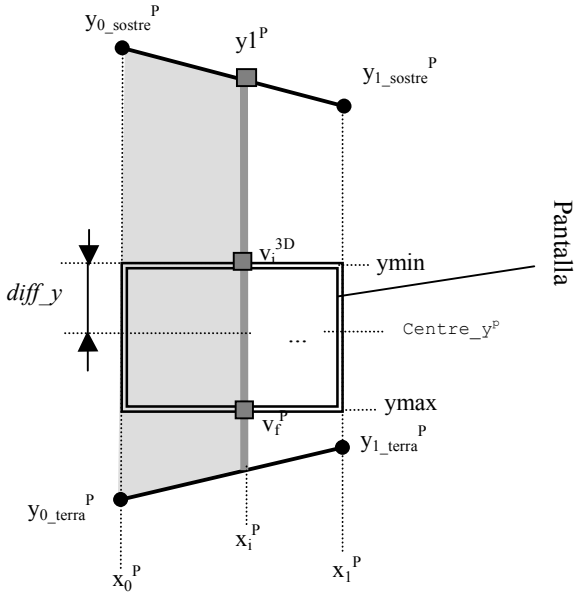


Figura 5.19c

Recordem la seva fórmula per obtenir $y_m^{3D'}$,

$$y_m^{3D'} = y_m^{3D} - y_u^{3D}$$

On y_m^{3D} podria ser l'altura de terra o de sostre. Nosaltres ho farem respecte el sostre per el que $y_m^{3D'}$o v_{centre}^{3D} serà,

$$V_{centre}^{3D} \leftarrow Y_{sostre}^{3D} - Y_u^{3D}$$

Ara, *diff_y* serà la diferencia entre el retallat del vector ymin[x_i^p] i respecte del centre de pantalla.

$$diff\_y = (ymin[x_i^p] - centre\_y^p)$$

Llavors, *diff_y* es multiplicaria per la escala, aconseguint la nova manera de calcular v_i^{3D} sense d'onatge,

$$V_i^{3D} = v_{centre}^{3D} + escala * diff\_y$$

5.3.2 Optimització de les rutines de pintat.

En tots els algorismes de rendering, el processos de més cost en temps es troben a les rutines de pintat (aproximadament al 95%). L'objectiu d'aquesta secció serà, un cop determinat el mode en que es treballarà, explicar la rutina de pintat de pixel bàsica i òptima. Llavors, integrarem aquesta funció a les funcions de mapeig verticals i horitzontals. Per últim, simplificarem al màxim aquestes rutines per tal aconseguir la eficiència òptima.

• Modes gràfics a la GBA

A la GBA hi podem trobar-hi 3 modes els quals expliquem a l'annex A1.4.1.2 i tenen les següents característiques:

-Mode 3: Sistema de vídeo RGB de 16 bits i amb una resolució de 240x160.
-Mode 4: Sistema de vídeo paleta (8 bits) amb doble-bufer i una resolució de 240x160.
-Mode 5: Sistema de vídeo RGB de 16 bits, amb doble-bufer i una resolució de 160x128.

Cada mode gràfic té els seus avantatges i els seus inconvenients. El mode 3, per exemple, disposa d'un sistema de vídeo de màxima qualitat a la GBA, ja que utilitza la màxima resolució de la GBA (240x160 píxels) i a més una profunditat de color de 16 bits. Per contra, tenim el desavantatge que no disposa de doble-bufer, pel que és necessari reservar un bufer de 240x160 de 16 bits, a memòria externa, per fer un bolcat de informació ràpid pel motiu de evitar parpelleigs de pantalla. Per aquest fet, aquest bufer causaria la ocupació del 30% del total disponible en memòria (256 kbytes), apart del cost de temps del bolcat de bufer a memòria de vídeo. Per aquest motiu aquest mode gràfic no seria l'adequat pels nostres propòsits. En canvi, els modes 4 i 5 disposen de doble-bufer. S'anomena doble-bufer un sistema de vídeo que disposa de dos bufers que, un es l'actiu el qual mostra el frame per pantalla mentre que en l'altre, el búfer de treball (no visible al usuari), s'escriuen els pixels **directament** per mostrar-se al pròxim frame.

Dels dos modes s'ha decidit utilitzar el mode 4 ja que es el mode més econòmic (només tracta informació de 8 bits). Anem a explicar la implementació necessària per escriure un pixel a la GBA, per aquest mode en concret.

- **Funcio PutPixel**

Sabem que tenim 2 bufers un ubicat a la posició 0x0600000 i l'altre a la posició 0x0600A000 i un és el bufer actiu i l'altre el bufer de treball. Ens interessa saber quin és el bufer de treball en tot moment, que serà on s'hi escriurà el pixel. El registre de control REG_CNT conté aquesta informació.

Si el bit 4é del registre REG_CNT (ubicat a la posició 0x400000) està a 1 vol dir que la memòria de treball és la 0x0600A000 altrament, si es troba a 0, la memòria de treball es troba a la posició 0x06000000.

Per fer la seva lectura, en codi C caldria fer,

```
char *ptr_video;

if(((* (u16*)0x4000000) & 0x10) == 1)
{
        ptr_video = (char *)0x0600A000
}
else
{
        ptr_video = (char *)0x06000000
}
```

Un cop disposem de l'adreça de treball cal fer l'offset XY per escriure-hi el pixel. El frame bufer de treball es unidimensional, però podem fer l'equivalent accés bidimensional mitjançant un offset lineal,

```
Ptr_video[y*240 + x] = color
```

O millor, la versió òptima,

```
Ptr_video[(y << 8) - (y >> 4) + x] = color
```

Una altre cosa a tenir en compte es que hi ha una greu limitació en el mode 4. Encara que el mode 4 té 8 bits per píxel, degut al disseny del hardware de la memòria de vídeo, només pot ser accedida de 16 en 16 bits. Això vol dir que al escriure en el mode 4 caldria de llegir els 2 píxels adjacents de la memòria, emmascarar el píxel que es vol canviar i escriure els dos píxels de nou a la memòria (llistat 5.10).

```
short  *video_ptr ; // (short *) equival a un punter de valors de 16 bits  o 2
pixels en mode 4.

if(((* (u16*)0x4000000) & 0x10) == 1)
{
       ptr_video = (short *)0x0600A000
}
else
{
       ptr_video =(short *) 0x06000000
}

// llegim els 2 pixels en x. Cal adonar-se que l'ampla del frame bufer
// s'ha reduit a la meitat ja que llegim 2 bytes en ves d'un.
// color_video = video_ptr[y*(240)/2 + x/2];

    if(x&0x1) // Volem inserir un pixel a la columna senar, cal salvaguardar
pixel parell (8 bits de menys pes)
    {
        color_video = ((color_video & 0xf) | color << 8);
    }
    else // Volem inserir un pixel a la columna parell, cal salvaguardar pixel
senar (8 bits de més pes)
    {
        color_video = ((color_video & (0xf << 8)) | color);
    }
     // ... i tornem a escriure els dos pixels.
    video_ptr[y*(240)/2 + x/2] = color_video;
```

Llistat 5.10

Degut a aquesta restricció imposada per l'arquitectura de vídeo, escriurem 2 píxels per iteració (2 bytes), ja que és que es més ràpid. Així la funció final quedaria,

```
void GBA_PutPixel(int x, int y, char color)
{
//PRE:      x       ∈[0,..120)
//          y       ∈[0,..160)
//        color     ∈[0,..,256)

      short *ptr_video;

      // Establint punter al bufer de treball...
      if((((* (short*)0x4000000) & 0x10) == 1)
      {
            ptr_video = (short *)(0x0600A000);
      }
      else
      {
            ptr_video = (short *)0x06000000;
      }

      // Offset (y*120 + x) i escritura dels dos pixels
      ptr_video[(y << 7) - (y << 3)+ x ] = (color | (color << 8));

}
```

- **Rutina de mapeig vertical**

A la secció 3.3.2.3, es va estudiar la rutina de mapeig vertical per les parets (llistat 3.10). que tot mostrem l'equivalent implementat en C,

```
void GBA_PutSlice(int x, int y1, int y2, int ID_Textura, FIXED u3d, FIXED v3d, FIXED escale_v3d)
{
//PRE:  x          ∈[0,..120)
//      y1, y2     ∈[0,..160) ∧ y1 ≤ y2
//      v3d, u3d inici de coordenades XY 3D segons x i y1, associades a les coordenades UV
//      de mapeig, en APF 16.16
//      escale_v3d, valor increment v3d en APF 16.16

    char  *textura_ptr, color;
    short color_video, offset_v;
    short amplada_textura,altura_textura;
    short y2d;

    // Lectura dades imprescindibles de la textura...
    textura_ptr          = Textura[ID_Textura].ptr_texels;
    amplada_textura      = Textura[ID_Textura].amplada_textura;
    altura_textura       = Textura[ID_Textura].altura_textura;
    y2d = y2 - y1 + 1;

    if(y2d > 0)
    {

        // Bucle principal de pintat de pixel...
        do{

            offset_v = (v3d >> 16) % altura_textura;

            color = textura_ptr[offset_v*amplada_textura + (u3d >> 16)];

           GBA_PutPixel(x,y,color);

           v3d = v3d + escale_v3d;
            y2d--;
        }while(y2d > 0);
    }
}
```

Tot seguit, anirem explicant les optimitzacions que se li pot fer aquesta rutina de mapeig fins arribar al seu punt de simplificació màxim. Ens hem de conscienciar que qualsevol petita reducció del cost del bucle principal de pintat de pixel accelerarà molt el procés en general, ja que com hem dit es una funció que es fa servir bastant (fins a 120x160 = 19200 iteracions).

1era Optimització: Evitant crides.

A la rutina de mapeig es crida la funció *GBA_PutPixel()*, si posem el seu codi en línia, no caldrà fer el cost implicat en la crida de la funció. A més, podem preparar el punter de vídeo abans de entrar al bucle principal.

```
void GBA_PutSlice(int x, int y1, int y2, int ID_Textura, FIXED u3d, FIXED v3d, FIXED escale_v3d)
{
//PRE:  x          ∈[0,..120)
//      y1, y2     ∈[0,..160) ∧ y1 ≤ y2
//      v3d, u3d inici de coordenades XY 3D segons x i y1, associades a les coordenades UV
//      de mapeig, en APF 16.16
//      escale_v3d, valor increment v3d en APF 16.16


    short *ptr_video;
    char  *textura_ptr, color;
    short amplada_textura,altura_textura, offset_v;
    short y2d;


    // Lectura dades imprescindibles de la textura...
    textura_ptr          = Textura[ID_Textura].ptr_texels;
    amplada_textura = Textura[ID_Textura].amplada_textura;
    altura_textura     = Textura[ID_Textura].altura_textura;
    y2d = y2 - y1 + 1;

    // Establint punter al bufer de treball...
    if((((* (short*)0x4000000) & 0x10) == 1)
    {
        ptr_video = (short *)(0x0600A000);
    }
    else
    {
        ptr_video = (short *)0x06000000;
    }

    // Bucle principal de pintat de pixel...
    if(y2d > 0)
    {
        do{

            offset_v = (v3d >> 16) % altura_textura;
            color = textura_ptr[offset_v* amplada_textura + (u3d >> 16)];

            // Offset (y*120+x) + escritura dels dos pixels
            ptr_video[(y << 7) - (y << 3)+ x ] = (color | (color << 8);

           v3d = v3d + escale_v3d;
            y2d --;

        }while(y2d > 0)

    }
```

2ª Optimització: Preindexat de offsets lineals

Si fem el preindexat de punter de vídeo inicial (y1*120 + x), i llavors, dins el bucle principal, anem sumant l'amplada efectiva (120), evitarem fer el càlcul de l'offset a cada iteració. Igualment per el punter de textura, si preindexem l'offset u, no cal fer la seva suma en el càlcul de l'offset UV per l'obtenció del texel.

```
void GBA_PutSlice(int x, int y1, int y2, int ID_Textura, FIXED u3d, FIXED v3d, FIXED escale_v3d)
{
//PRE:   x          ∈[0,..120)
//       y1, y2     ∈[0,..160) ∧ y1 ≤ y2
//       v3d, u3d inici de coordenades XY 3D segons x i y1, associades a les coordenades UV
//       de mapeig, en APF 16.16
//       escale_v3d, valor increment v3d en APF 16.16


    short *ptr_video;
    char  *textura_ptr, color;
    short color_video;
    short amplada_textura,altura_textura, x2d;


    // Lectura dades imprescindibles de la textura...
    textura_ptr          = Textura[ID_Textura].ptr_texels;
    amplada_textura = Textura[ID_Textura].amplada_textura;
    altura_textura     = Textura[ID_Textura].altura_textura;
    y2d = y2- y1 +1;

  // Establint punter al bufer de treball...
    if((((* (short*)0x4000000) & 0x10) == 1)
    {
                ptr_video = ((short *)(0x0600A000 + (y1 << 7) - (y1 << 3)+ x);
    }
    else
    {
                ptr_video = ((short *)(0x06000000 + (y1 << 7) - (y1 << 3)+ x);
    }

    textura_ptr = textura_ptr + (u3d >> 16);

    // Bucle principal de pintat de pixel...
   if(y2d > 0)
   {
         do{

           offset_v = (v3d >> 16) % altura_textura;
           color = textura_ptr[offset_v*amplada_textura];

          // Offset (y*120+x) + escritura dels dos pixels
          *ptr_video = (color | (color << 8);

          ptr_video +=120;

          v3d = v3d + escale_v3d;

         y2d --
    }while(y2d > 0);
  }
}
```

3era Optimització: Textures potencies de dos

Veiem que al bucle tenim un mòdul i sabem que les divisions són lentes en ser processades. El mòdul, el fem servir per filtrar la coordenada V perquè no es surti del rang de altura de textura que té establert. No obstant, si la textura té la dimensió potència de 2, llavors el rang $V \in [0,..,2^{\log2(altura\_textura)}-1]$. Per exemple, si la altura de la textura fos de 256 texels, llavors $V \in [0, 2^{\log2(256)}-1] = [0, 2^8-1] = [0, 256-1] = [0,255] = [0,0xFF_{|16|}]$. Així doncs, si per aquest exemple, enmascarem V amb 0xFF mitjançant una AND lògica, a cada iteració assegurem que qualsevol valor v^{3D} estarà al interval de altura vàlid.

A més, si la amplada de textura també és potencia de 2, en ves de multiplicar V emmascarada per l'amplada de textura, només caldrà fer un desplaçament a la esquerra (<< en C) $\log_2$(amplada_textura) vegades.

```
void GBA_PutSlice(int x, int y1, int y2, int ID_Textura, FIXED u3d, FIXED v3d, FIXED escale_v3d)
{
//PRE:   x          ∈[0,..120)
//       y1, y2     ∈[0,..160) ∧ y1 ≤ y2
//       v3d, u3d inici de coordenades XY 3D segons x i y1, associades a les coordenades UV
//       de mapeig, en APF 16.16
//       escale_v3d, valor increment v3d en APF 16.16


    (...)

    log2_amplada_textura = Textura[ID_Textura].log2_amplada_textura;
    mascara_altura = altura_textura -1;

    // Bucle principal de pintat de pixel...
    if(y2d > 0)
    {

     do{

         offset_v = (v3d >> 16) & mascara_altura;

          color = textura_ptr[offset_v << log2_amplada_textura]

         // Offset (y*120+x) + escritura dels dos pixels
         *ptr_video = (color | (color << 8);

         ptr_video +=120;
          v3d = v3d + escale_v3d;
         y2d --;

    }while(y2d > 0);
  }
}
```

4a Optimització: Codificació del bucle principal a ensamblador ARM

Una de les etapes que més significatives és portar la acció GBA_PutSlice() es la codificació del bucle principal a en ensamblador ARM i a memòria interna, ja que com s'ha dit al principi, la seva execució serà la més ràpida a la màquina GBA.

Per dir que el compilador porti el codi a memòria interna cal declarar a l'inici del mòdul ensamblador `.SECTION .iwram`. Per ultim, el pas de paràmetres hi tindrem el punter destí (video) i font (textura) amb el preindex de offset, v inicial, increment v, mascara_v,$\log_2$(amplada_u) i el numero de iteracions (x2d). Com que hi tenim més de 4 paràmetres de entrada, s'ha definit la llista de paràmetres en una estructura C (llistat 5.20) i s'ha passat el seu punter dins la rutina per evitar el pas a través de la pila, segons el procediment estàndard de crides (veure a0.5.3, annex 0).

```
typedef struct{
        short *ptr_video;
        char  *ptr_textura;
        FIXED   v3d;
        FIXED   inc_dv3d;
        int     mascara_altura; // En APF 16.16
        int     log2_amplada_u;
        int     y2d;
}tParametresBuclePrincipal;


extern void ARM_BuclePrincipal(tParametresBuclePrincipal *PBP);


void GBA_PutSlice(int x, int y1, int y2, int ID_Textura, FIXED u3d, FIXED v3d, FIXED escale_v3d)
{
//PRE:  x          ∈[0,..120)
//      y1, y2     ∈[0,..160) ∧ y1 ≤ y2
//      v3d, u3d inici de coordenades XY 3D segons x i y1, associades a les coordenades UV
//      de mapeig, en APF 16.16
//      escale_v3d, valor increment v3d en APF 16.16

    short *ptr_video;
    short amplada_textura,altura_textura, x2d;
    tParametresBuclePrincipal PBP;



    // Lectura dades imprescindibles de la textura...
    PBP.ptr_textura           = Textura[ID_Textura].ptr_texels + (u3d >> 16);
    PBP.log2_amplada_u        = Textura[ID_Textura].log2_amplada_u;
    PBP.mascara_textura       = Textura[ID_Textura].altura_textura - 1;
    PBP.y2d                    = y2- y1 +1;
    PBP.v3d                    = v3d;
    PBP.escale_v3d             = escale_v3d;

  // Establint punter al bufer de treball...
    if((((* (short*)0x4000000) & 0x10) == 1)
    {
                PBP.ptr_video = ((short *)(0x0600A000 + (y1 << 7) - (y1 << 3)+ x);
    }
    else
    {
                PBP.ptr_video = ((short *)(0x06000000 + (y1 << 7) - (y1 << 3)+ x);
    }

    // Bucle principal de pintat de pixel...
    if(y2d > 0)
    {
        ARM_BuclePrincipal(&PBP);
    }
}
```

Llistat 5.20

La codificació del bucle principal a ensamblador ARM, i configurat el més òptim (estudiat) seria el següent llistat de codi,

```
.SECTION .iwram                 @ Li diem al compilador que portir el codi d'aquest mòdul a
                                @ memòria interna

.ARM                            @ Li diem al compilador que el codi esta en format ARM (32 bits)

.ALIGN                          @ Codi alineat a 32 bits

.GLOBL ARM_BuclePrincipal       @ Per enllaçar la funció per un modul extern

.TEXT                           @ A partir d'aquí comença el codi

ARM_BuclePrincipal:

        STMFD   sp!,{r4-r8}             @ Salvem registres r4-r9 ja que, seran sobrescrits.

        LDMIA   r0,{r0-r8}              @ Carreguem paràmetres..

        @ GAS permet el nombrament de registres. Ho fem tot seguit, per determinar amb més claredat cada
        @ funció d'aquests dins la rutina.

        r0  .req        ptr_video       @ r0 conté el punter video més offset y*120+x.
        r1  .req        ptr_textura     @ r1 conté el punter textura més offset u.
        r2  .req        v3d             @ r2 conté v3d.
        r3  .req        escale_v3d      @ r3 conté inc_v3d.
        r4  .req        mascara_altura  @ r4 conté mascara_altura
        r5  .req        log2_amplada_u  @ r5 conté log2_amplada_u
        r6  .req        y2d             @ r6 conté y2d
        r7  .req        offset_v        @ r7 conté el valor de offset_v
        r8  .req        color           @ r8 conté color


        MOV     mascara_altura, mascara_altura, lsl #16   @ Convertim mascara_altura en APF 16.16, perquè
                                                          @ més endavant veurem que, per la selecció v només
                                                          @ caldrà una sola instrucció.

        @ Bucle principal de pintat de pixel...

Continua_Bucle:                                 @ do{

        @ En ves de fer offset_v = (v3d >> 16) & mascara_altura,  fem offset_v = (v3d & mascara_altura) >> 16,
        ja @ que es pot fer en una sola instrucció ARM. Per això mascara_altura es passa en APF 16.16.

        AND     offset_v, mascara_altura, v3d, #lsr 16

        @ color = ptr_textura[offset_v, offset_v << log2_amplada_textura];

        LDR     color, [ptr_textura, offset_v, lsl log2_amplada_u]  @ ARM permet una carrega offset_v amb
                                                                    @ un predesplaçament constant o per
                                                                    @ registre
        @ color = color | (color << 8)

        ORR     color, color, color, lsr #8

        @ *ptr_video  = color
        @ ptr_video  += 120

        STRH    color, [ptr_video], #+120       @ Guarda 16 bits. Suma +120 un cop feta la escriptura.

        @ v3d = v3d + escale_v3d
        ADD     v3d, v3d, escale_v3d

        @ y2d--
        SUBS    y2d, y2d, #1                    @ Resta (SUB) amb modificació del registre CPSR (S)
        }while(y2d != 0)
        BNE     Continua_Bucle                  @ BNE, salta si no està activat el flag de zero.

Acabar_Bucle:

        LDMFD   sp!,{r4-r8}
        BX      lr
```

Llistat 5.21

Finalment, podem observar que el bucle principal ens ha quedat en només 7 operacions aritmètiques a codi ensamblador.

- **Rutina de mapeig de pla**

Igual que la rutina de mapeig vertical, la rutina de mapeig de pla també és una de les que més itera i el més convenient es que s'optimitzi tant com sigui possible, principalment, el bucle principal. Veiem el codi de la acció de pintat horitzontal (ja estudiada a la secció A3.4.3), optimitzada als punts anteriors del mapeig vertical i traspassada a code C.

```
void RenderitzaScanline(int x1, int x2,int y, FIXED u3d, FIXED v3d, FIXED du3d, FIXED dv3d, int ID_Textura)
{

        int     Npixels, x;
        FIXED   u, v, x_mapa, z_mapa;

        (...)

        Selecció memòria de video més offset y*120 + x

        (...)

        Npixels = x2 - x1 + 1;
        x_mapa  = x0;
        z_mapa  = z0;
        mascara_amplada = Textura[ID_Textura].amplada-1;
        mascara_altura  = Textura[ID_Textura].altura -1;
        log2_amplada_u  = Textura[ID_Textura].log2_amplada_u;


        if (Npixels > 0) //  Es renderitza scanline.
        {

                do{
                        u = (x_mapa >> 16) & mascara_amplada;
                        v = (z_mapa >> 16) & mascara_altura;

                        offset_uv = (v << log2_amplada_u) + u;

                        color = ptr_textura[offset_uv];

                        ((short *)ptr_video++) = color | (color << 8);

                        x_mapa = x_mapa + dx_mapa;
                        z_mapa = z_mapa + dz_mapa;

                }while(--Npixels);
        }
}
```

1era Optimització: Aprofitant la linealitat del video

El mapeig horitzontal a l'espai de pantalla té un gran avantatge, perquè podem aprofitar la linealitat de memòria de vídeo per escriure 32 bits (paraules) en ves de 16 bits com es feia en el mapeig vertical.
L'únic inconvenient es que les escriptures de 32 bits han esta alineades en posicions de memòria múltiples de 4 bytes (0x006A000, 0x006A004, 0x006A008, 0x006A00C, etc), per això 1er caldrà comprovar en la posició que estem escrivim i escriure mitja paraula (16 bits) si la posició de memòria no és de múltiple de 4. Tot seguit es calcula i s'escriu la longitud de paraules que es pot escriure. Per últim, s'haurà de escriure la mitja paraula restant, si cal.

```
if (Npixels > 0) //  Es renderitza scanline.
{
        if((ptr_video & 0x3) != 0) // No es multiple de 4, cal escriure mitja paraula...
        {
                u = (x_mapa >> 16) & mascara_amplada;
                v = (z_mapa >> 16) & mascara_altura;

                color = ptr_textura[(v << log2_amplada_u) + u];

                ((short *)ptr_video++) = color | (color << 8);

                x_mapa = x_mapa + dx_mapa;
                z_mapa = z_mapa + dz_mapa;

                Npixels--;
        }

        // Calculem el numero de paraules que podem escriure...

        hi_ha_resta =  Npixels & 0x1;
        n_paraules  = (Npixels >> 1);

        if(n_paraules > 0) // escriptura ràpida (32 bits)...
        {

            do{
                u = (x_mapa >> 16) & mascara_amplada;
                v = (z_mapa >> 16) & mascara_altura;

                color1 = ptr_textura[(v << log2_amplada_u) + u];
                color1 = color1 | (color1 << 8);

                x_mapa = x_mapa + dx_mapa;
                z_mapa = z_mapa + dz_mapa;

                u = (x_mapa >> 16) & mascara_amplada;
                v = (z_mapa >> 16) & mascara_altura;

                color2 = ptr_textura[(v << log2_amplada_u) + u];
                color2 = color2 | (color2 << 8);

                x_mapa = x_mapa + dx_mapa;
                z_mapa = z_mapa + dz_mapa;

                ((int *)ptr_video++) = color1 | (color2 << 16); // little endian

            }while(n_paraules--);
        }

        if(hi_ha_resta) // Cal la ultima mitja paraula...
        {
                u = (x_mapa >> 16) & mascara_amplada;
                v = (z_mapa >> 16) & mascara_altura;

                color = ptr_textura[(v << log2_amplada_u) + u];

                ((short *)ptr_video) = color | (color << 8);

                x_mapa = x_mapa + dx_mapa;
                z_mapa = z_mapa + dz_mapa;
        }

}
```

2ona Optimització: El bucle principal a ensamblador

Per últim, per optimitzar al màxim el nostre pintat horitzontal s'hauria de passar el bucle principal de pintat a ensamblador.

```
ARM_BuclePrincipalHoritzontal:

        STMFD   sp!,{r4-r12,r14}        @ Salvem registres r4-r12 i r14 ja que, seran sobrescrits.

        LDMIA   r0,{r0-r9}              @ Carreguem paràmetres..

        @ GAS permet el nombrament de registres. Ho fem tot seguit, per determinar amb més claredat cada
        @ funció d'aquests dins la rutina.

        r0   .req       ptr_video       @ r0  conté el punter video més offset y*120+x.
        r1   .req       ptr_textura     @ r1  conté el punter textura més offset u.
        r2   .req       x_mapa          @ r2  conté z_mapa
        r3   .req       dx_mapa         @ r3  conté dz_mapa
        r4   .req       z_mapa          @ r4  conté z_mapa
        r5   .req       dz_mapa         @ r5  conté dz_mapa
        r6   .req       mascara_amplada @ r6  conté mascara_altura
        r7   .req       mascara_altura  @ r7  conté mascara_altura
        r8   .req       log2_amplada_u  @ r8  conté log2_amplada_u
        r9   .req       x2d             @ r9  conté x2d
        r10  .req       u               @ r10 conté el valor de u
        r11  .req       v               @ r10 conté el valor de v
        r12  .req       color1,color    @ r11 conté color1 i color
        r14  .req       color2          @ r12 conté color2



        MOV     mascara_altura, mascara_altura, lsl #16   @ Convertim mascara_altura en APF 16.16, perquè

        MOV     mascara_amplada, mascara_amplada, lsl #16 @ Convertim mascara_amplada en APF 16.16

        @ Bucle principal de pintat horitzontal...

Continua_Bucle_Horitzontal:                     @ do{

        AND     u, mascara_altura,  x_mapa, #lsr 16       @ u = (x_mapa & mascara_amplada) >> 16
        AND     v, mascara_amplada, z_mapa, #lsr 16       @ v = (z_mapa & mascara_altura) >> 16

        ADD     offset_uv, u, v, log2_amplada_u           @ offset_uv = v << log2_amplada_u + u

        LDR     color1, [ptr_textura, offset_uv]          @ color1 = ptr_textura[offset_uv];

        ORR     color1, color1, color1, lsr #8            @ color = color | (color << 8)

        ADD     x_mapa, x_mapa, dx_mapa                   @ x_mapa = x_mapa + dx_mapa
        ADD     z_mapa, z_mapa, dx_mapa                   @ z_mapa = z_mapa + dz_mapa

        AND     u, mascara_altura,  x_mapa, #lsr 16       @ u = (x_mapa & mascara_amplada) >> 16
        AND     v, mascara_amplada, z_mapa, #lsr 16       @ v = (z_mapa & mascara_altura) >> 16

        ADD     offset_uv, u, v, log2_amplada_u           @ offset_uv = v << log2_amplada_u + u

        LDR     color2, [ptr_textura, offset_uv]          @ color2 = ptr_textura[offset_uv];

        ORR     color2, color2, color2, lsr #8            @ color = color | (color << 8)

        ADD x_mapa, x_mapa, dx_mapa                       @ x_mapa = x_mapa + dx_mapa
        ADD z_mapa, z_mapa, dx_mapa                       @ z_mapa = z_mapa + dz_mapa

        EOR     color, color1, color2, lsl #16            @  color = color | color2 << 16

        STR     color, [ptr_video], #+4                   @ ((int *)ptr_video++) = color

        SUBS    x2d, x2d, #1                              @ x2d--

        BNE     Continua_Bucle_Horitzontal                @ }while(y2d != 0)

        LDMFD   sp!,{r4-r12,r14}
        BX      lr
```

Llistat 5.25

5.3.3 Escalat vertical per hardware

La GBA disposa de escalat per hardware en el mode 4. Els registres de rotat per aquest mode són BG2PB i BG2PD, (també anomenats R_BG2ROTDMX i R_BG2ROTDMY) ubicats a les adreces 0x04000022 i 0x04000024 respectivament. Aquests contenen el factor escala/rotat de amplada i altura respectivament, codificats a una APF 16.8. Quan la pantalla inicialment no està rotada i el factor escala és 1, els registres estan inicialitzats com,

```
R_BG2ROTDMX = 0;
R_BG2ROTDMY = 0x100;          // 0x100, representa 1 en APF 16.8
```

Aquí tenim avantatge que si escalem al doble la pantalla només caldrà pintar la meitat de la pantalla (80 files). Per fer-ho s'ha de assignar el valor 0.5 en APF al registre BG2ROTDMY a,

```
R_BG2ROTDMY = 0x80;           // 0x80, representa 0,5 en APF 16.8
```

Per més informació sobre la funcionalitat d'aquests registres, consultar a3.4.2 de l'annex 3.

6. Resultats i conclusions

Han estat molts objectius abans de arribar al final d'aquest treball. Fa tres anys, es va començar el disseny del nostre motor: un ray-casting amb llançament de raig per-columna i basat en un mapa de cel·les (figura 6.1). Un cop acabat el seu estudi, es va muntar un prototipus per PC amb la llibreria SDL i per la GBA optimitzat al màxim a quan a pintat per-pixel (resolució 120x160 i ensamblador) i aritmèticament (APF 16.16).

Figura 6.1 *Execució del primer motor sobre la GBA, un ray-casting amb mapa de cel·les a la GBA (dreta) i a l'emulador de GBA (esquerra).*

Més tard, es va modificar el nostre motor perquè fos capaç de renderitzar un mapa basat en portals i sectors. També, vam implementar la manera de renderitzar desnivells ortogonals simplement amb una informació addicional d'altura de sostre i terra en el sector. Com que el motor modificat necessitava fer més operacions per-raig, pel que ens vam veure obligats a estudiar el canvi de tècnica perquè renderitzès més ràpidament una escena, però mantenint l'arquitectura base de motor, es a dir, el ray-casting. Es va aconseguir que el motor fos capaç de renderitzar un sector amb només del llançament d'un raig.

La següent part va ser l'estudi per introduir el pintat de sostre i terra al nostre motor. Es sabia que la GBA era capaç de pintar plans per hardware, la qual cosa era un gran avantatge si es pogués adaptar a la renderització del nostre motor, ja no que no suposaria cap cost en temps. Per això, es va estudiar i construir un prototipus "mode 7" per la GBA.

Figura 6.2 *Execució del prototipus "mode 7" a la GBA (dreta) i a l'emulador de GBA (esquerra)*

Malauradament, no va ser possible l'adaptació del "mode 7" perquè només estava disponible per mode gràfics basats en cel·les i el nostre motor treballava en un mode basat en mapa de bits. Però bé, es va continuar l'estudi del pintat de plans i és va fer tal com teníem configurat el pintat de tires de paret, és a dir pintat de per franges verticals. Pel el pintat de franges verticals calia aplicar 6 multiplicacions per-pixel, pel qual va causa també va tenir una caiguda important en velocitat. Era necessari modificar la tècnica de pintat de pla. Es va estudiar el pintat d'un pla per franges horitzontals, i es va comprovar que només calia 2 sumes per-pixel, per això, vam adaptar la tècnica de pintat de pla per franges horitzontals.

• Resultats

Mostrem la següent taula històrica dels FPS de mitjana aconseguits a la GBA, per les optimitzacions introduïdes, partint del motor dissenyat **sense efectes** es a dir, sense il·luminació, ni objectes etc, per una escena constituïda per un mapa de 30 sectors.

OPTIMITZACIÓ	DESCRIPCIÓ OPTIMITZACIÓ	Mitjana FPS	FACTOR ACCELERACIÓ
Motor Inicial	*Ray-casting basat en un mapa sectors i portals, amb desnivells ortogonals i mapeig de terra/sostre sense cap optimització.*	0,0051	-
Optimització aritmètica 1	*Eliminació d'arrels quadrades, divisions, etc.*	0,035	x3
Optimització aritmètica 2	*Substitució aritmètica real per la Aritmètica de Punt Fixa (APF).*	0,67	x19,28
Mapeig de paret òptim	*Pintat a una resolució de 120x160 i en assemblador.*	12,09	x18,05
Millora tècnica de render	*Renderització de 1 raig-sector.*	20,5	x1,66
Mapeig per franges horitzontals		25	x1'219
Operació escalat hardware	*Renderització a una resolució de 120x80, i escalat vertical x2.*	35	x1,4

L'ultima cosa que vam contemplar en aspectes de optimització va ser, fer més ràpid les renderitzacions amb sectors transparents implicats, que es va aconseguir fent un preprocessat de l'escena, accelerant fins a un factor de acceleració **1,7** respecte la renderització sense preprocessat.

Acabada la part de optimització, es va acabar l'estudi d'aquest projecte amb la incorporació de tècniques avançades de motor com el filtre anti-aliasing mip-mapping, il·luminació per-sector, efectes de portal, transparència de objectes, renderització d'objectes i utilització d'una estructura de món 3D, com ja s'han explicat a la secció 3.4. Molts d'aquests havien de col·locats en situacions de mapa adequats, per evitar gaire consum de temps.

Finalment, amb unes quantes imatges renderitzades del motor, a la GBA i a la l'emulador GBA, demostrem que hem aconseguit les característiques de motor que ens havíem proposat a l'inici del d'aquest treball

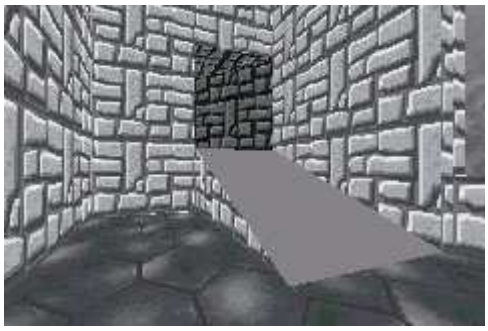
- **Renderització de paret, terra i sostre amb mapeig de textura amb il·luminació,**

GBA

Emulador GBA

- **Renderització de pendents,**

GBA



Emulador GBA

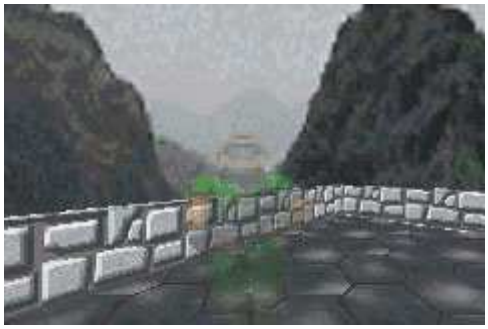
- **Mapeig de portal amb una textura transparent,**

GBA

Emulador GBA

- **Renderització de paisatge i un objecte transparent,**

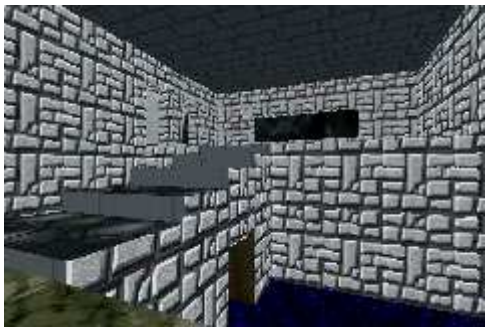
GBA



Emulador GBA

- **Renderització de desnivells i una estructura 3D extesa,**

GBA



Emulador GBA

Conclusions

Un cop acabat aquest projecte, hem aconseguit el nostre, i tant esperat, objectiu que es marcava des de el començament d'aquest capítol: dissenyar un motor 3D per la GBA interactivament.

Durant el transcurs d'aquest treball, s'ha demostrat que la tècnica del nostre motor era la que tècnica buscàvem. Un renderitzador fàcil de implementar i, alhora, eficient i flexible a nous canvis. Si més no, recordem vam poder incorporar una renderització de pendents, desnivells ortogonals, estructura de mon 3D, etc. sense canviar l'estructura 2D del mapa, ni apenes cap modificació al renderitzador.

Encara que aquesta tècnica hagi demostrat la seva gran potencia, no assegurem que hagi estat la millor solució, en aspectes de optimització. Hi han altres tècniques de rendering, els BSP, POV, etc. Seria important, doncs, implementar un motor basat en altres tècniques i intentar comparar els resultats obtinguts amb la tècnica dels portals i sectors.

• Treballs futurs

Hem deixat un motor 3D bastant complert, però encara es molta la feina que s'ha proposat de cara al futur.

- Mapeig de pendents

Fer l'estudi pertinent per aconseguir mapejar els pendents, ja que disposen d'una perspectiva diferent, a la renderització d'un pla.

- Editor de mapes

Actualment, cal introduir la informació de món a mà. La construcció d'un editor de mapes o la construcció d'una eina de conversió d'un format de mapa, basat en l'arquitectura del nostre motor, facilitaria la ràpida creació de móns.

- Eines

Juntament amb GFX2GBA, JGT vistes al annex 2, crear altres eines que tinguin com objectiu facilitar les introducció de noves informacions dins de codi, per ser llegit més tard, a la GBA.

- Intel·ligència

Implementar un motor de intel·ligència que sigui capaç de dirigir els objectes en funció dels moviments del jugador, o en determinades situacions.

- Ascensors/Portes

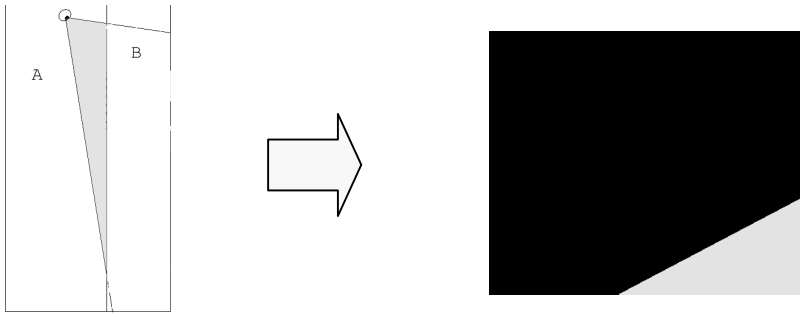
Ara les altures de sostre i terra de cada sector estan fixades durant tot el recorregut. Fent variable l'altura del terra o de sostre a temps de execució, podem simular sectors per fer-los servir com plataforma o comportes que connecten a altres sectors. La altura podria estar controlada per un motor de intel·ligència.

- Renderització de objectes 3D

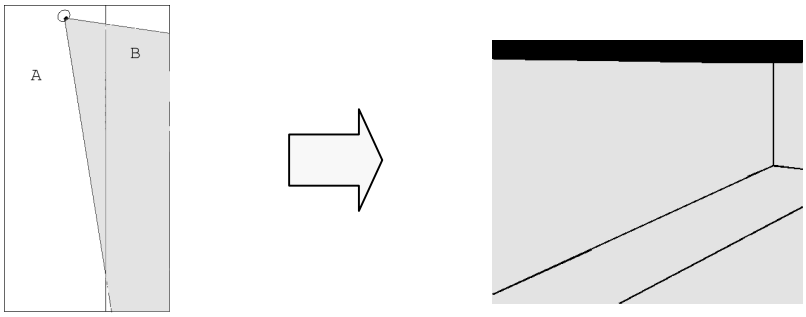
El nostre motor es capaç de renderitzar objectes representats per imatges 2D. Estaria bé doncs, que es pogués fer l'estudi i implementació d'un renderitzador d'objectes 3D per incorporar-lo al nostre motor. A la vegada, podríem fer servir el renderitzador d'objectes, per accelerar el processat dels sectors transparents i parcials, si aquests, els simulem per objectes 3D cúbics. A la figura la secció 5.2, es va estudiar un preprocessat de l'escena per evitar fer renderitzacions de sectors visitats.

No obstant, el preprocessat tampoc es la millor solució amb una escena amb molts sectors transparents, ja que es sobrecarrega bastant el preprocessat. Existeix una altre manera de fer més òptim el renderitzat de sectors transparents si, per exemple, es fes la combinació de la renderització del món amb objectes 3D cúbics.

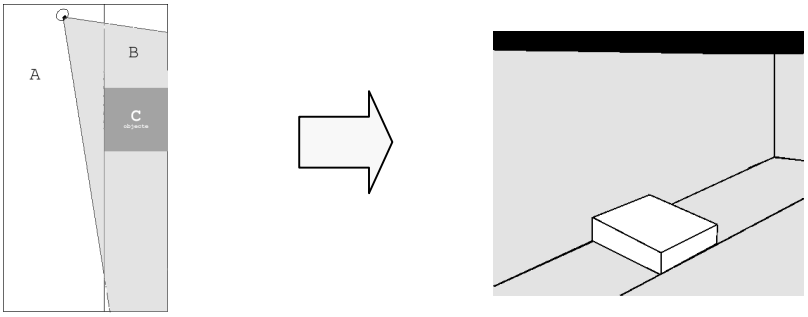
Per l'exemple 5.8, només caldria la renderització de dos sectors i un objecte 3D. Vegem-ho amb detall.



Renderització del sector A



Renderització del sector B



S'acaba amb la renderització del sector C, simulat per un objecte 3D.

7. Bibliografia

Bibliografia i referències segons citació MLA

[JCC02] **Catà Castillo, Jordi**
"La tècnica dels portals i sectors". *Recorregut interactiu d'escenes arquitectòniques.* UdG (Girona): Projecte fi de carrera, 2002. 19-41.

[LOW02] **Low**, **Kok-Lim**
Perspective-Correct Interpolation (2002) 2p. En línia.
http://www.cs.unc.edu/~lowk/research/writings/lowk_persp_interp.pdf
(26 Gener del 2005)

[HCK97C] **Hecker, Chris**
"It's Affine Day". *Texture Mapping Part IV:Approximations* (1997) 7p. En línia.
http://www.d6.com/users/checker
(03 del Juny de 2001)

[AIR90] **Airey, John.M**
Increasing Update Rates in the building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculation
PhD Thesis, UNC Chapel Hill, 1990.

8. Agraïments

Voldria agrair i dedicar el projecte a un conjunt de persones que m'han ajudat d'una manera o una altra a la realització d'aquest projecte que és l'últim pas per assolir la titulació d'Enginyer Tècnic en Informàtica de Sistemes. Entre aquestes persones vull destacar:

- Francesc Espada i Antònia Brau, els meus pares per haver-me donat suport incondicionalment durant aquests anys, haver-me ensenyat a espavilar-me i a lluitar per aquesta gran i dura vida.

- A la meva amiga Anna Fernández, per haver-me donat un punt de reflexió sobre la manera de conèixer' s a un mateix i orgullós de ser com és.

- Als meus companys, per haver-me animat a acabar aquest llarg treball.

- Als autors de llibres, pagines web, etc, que m' han omplert de informació essencial per la finalització d'aquest projecte.

- Per últim i més important, en Gustavo Patow per haver-me orientat durant el transcurs d'aquest treball. De no haver estat per ell, aquest treball no hagués sortit a la llum.

Moltes gràcies a tots.