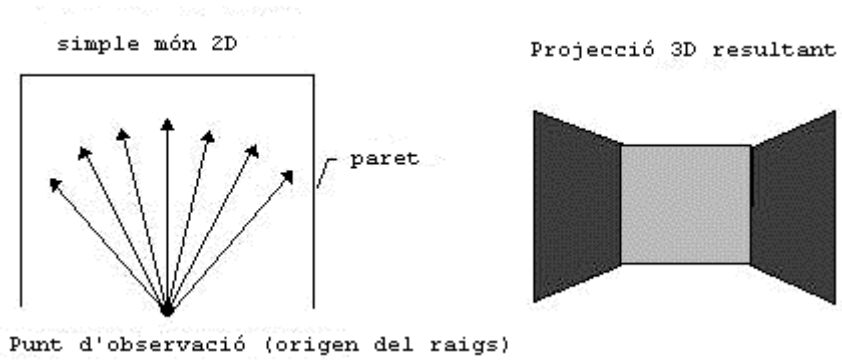
Capítol 1: Estudi d'un motor Ray-casting

1.1 Introducció

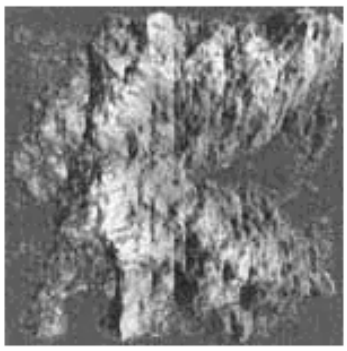
1.1.1 Definició Ray-casting

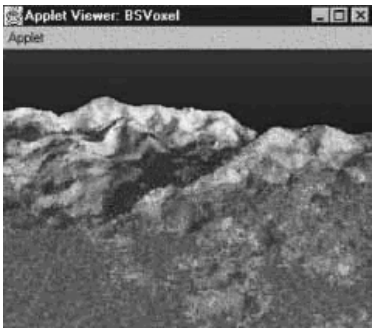
Ray-casting és una tècnica que transforma una estructuració del món limitat, com un mapa molt simplificat, a una projecció 3D pel traçament de raigs des d'un punt d'observació. La tècnica **ray-casting es basa en llançar un raig virtual des d'un punt d'observació per determinar la distància de col·lisió amb un objecte (en el nostre cas, una paret**).

Per exemple, en la següent figura es mostra una manera simple de llançat de raigs des d'un punt d'observació i la resultant projecció 3D de la paret,



Apart de la renderització de móns virtuals, el ray-casting també s'aplica a renderització de terrenys (figura 1.2), especialment per a simuladors de vol,





1.1.2 Història

L'aparició del ray-casting va ser degut a la necessitat de posseir una grau de realisme similar al **ray-tracing** [1] (veure *figura 1*.3), però que requerís menys temps per generar una imatge i menys espai de disc, a canvi d'unes restriccions que tenen a veure amb la simplificació del món que desitgem representar.

El ray-casting va començar amb l'acabat d'un joc: *Wolfenstein 3D* de id Software, en 1992 (veure figura 1.4). Dins el joc *Wolfenstein 3D*, el jugador és situat a un laberint 3D, on ell/ella ha de trobar la sortida mentre disputa la vida amb alguns oponents. *Wolfenstein 3D* va esdevenir un immediat clàssic a causa de l'animació ràpida i suau que oferia.

Wolfenstein 3D va ser creat per id Software, però principalment, pel programador de *id*, John Carmack; que va ser la persona que va desenvolupar el motor gràfic ray-casting.

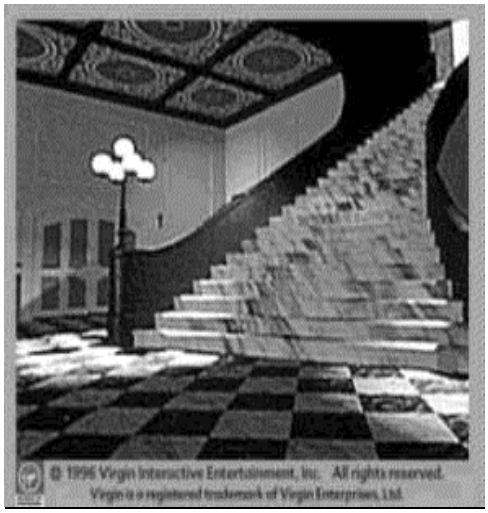


figura 1.3 *Escena del joc 7th Guest, el qual està basat en ray-tracing. El resultat de la renderització és quasi perfecte. No obstant, el moviment de jugador està restringit a un camí predeterminat perquè la quantitat de imatges prerenderitzades són limitades.*

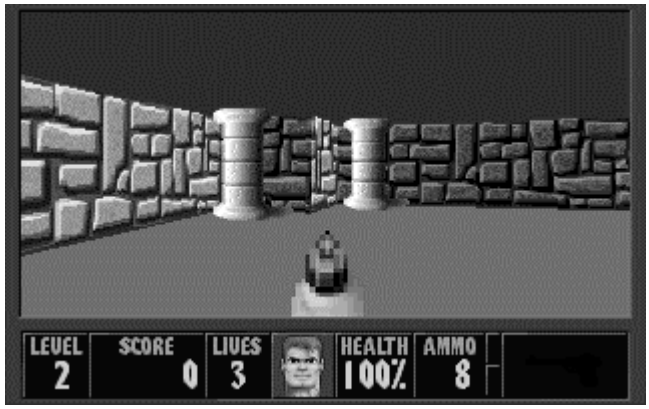


figura 1.4 *Escena del joc Wolfenstein 3D de id, el qual està basat en el ray-casting. Cal observar l'aspecte de bloc que tenen les parets. La pistola, la barra d'estat i les columnes són gràfics afegits.*

[1] **Ray-Tracing**: Tècnica de traçat de raigs. Es basa en el llançament de raig/pixel obtinguent una renderització amb molta precisió, per que a cada pixel té la informació exacte de l'objecte col·lisionat.

1.2 Definicions

1.2.1 Sistema de coordenades (S.C)

El sentit dels eixos del S.C que utilitzarem serà el que mostra a continuació,

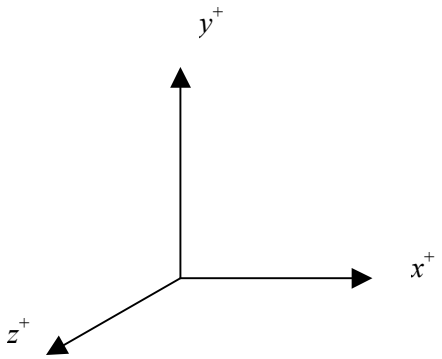


figura 1.5 *Sistema de coordenades al que es farà referència durant tota la documentació*

1.2.2 Orientacions

Les orientacions que farem servir, diferents a les que estem acostumats a veure, seran les següents,

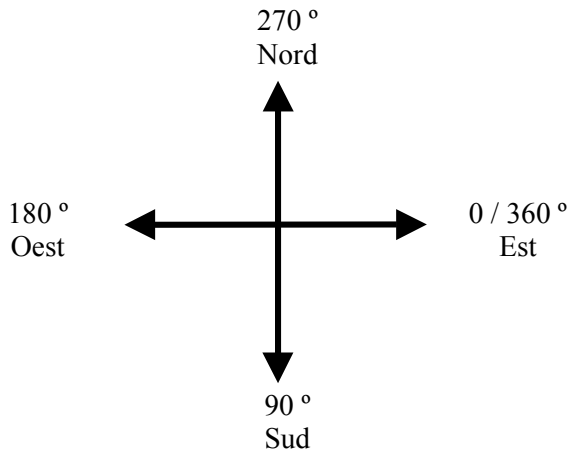


figura 1.6 *Orientacions*

1.3 El mapa (1era versió)

Una restricció a imposar al món en aquest model, és una col·locació de cubs sobre una malla 2D en cel·les de 64x64 unitats (figura 1.7). Això provocarà, en la futura renderització, tenir parets d'igual amplada, altura i alçada i un aspecte d'estar envoltat de blocs, com passava amb el joc *Wolfenstein 3D* (figura 1.4).

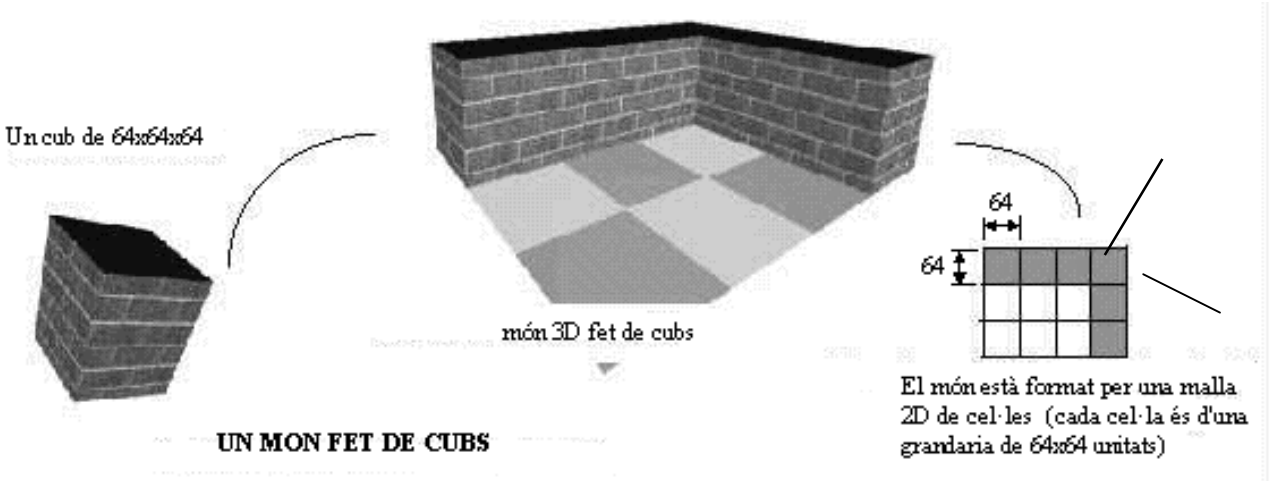


figura 1.7

La malla 2D representa el pla XZ del mapa. També representa el pla per on es mourà el nostre jugador, com es descriurà a la secció 1.4. S'especifica una grandària de cel·la de 64x64 unitats, llavors cada cub tindrà una dimensió de 64x64x64 unitats. (L'opció de 64 és arbitraria, però és molt útil la manipulació d'un numero quan és potència de 2 perquè podem executar operacions aritmètiques equivalents que són més ràpides que la multiplicació o la divisió). Veure la figura 1.7 per més informació.

Si la cel·la està ocupada per un cub, aquest s'haurà de representar gràficament en el procés de pintat. Cada cel·la de la malla 2D comença en un offset múltiple de 64 (0,64,128,...) i acaba en un offset múltiple de 64-1 (63,127,191,...), ja que s'hi vol associar els rangs d'una textura de 64x64. A la figura 1.8, presentem algunes textures definides al nostre motor,

A la figura 1.9, veiem com el pintat d'un cub a l'espai 3D amb la textura número 4 de la figura 1.8.

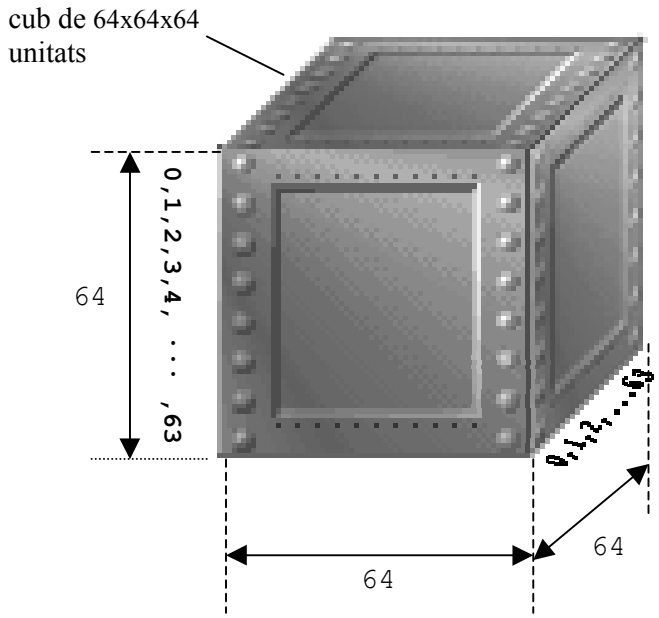


figura 1.9 *Representació del cub a l'espai 3D. Els rangs del cub, iguals al de cel·la, estan associats als rangs de textura establerts ([0,...,63]x[0,..,63] unitats), perquè en la representació de cub a l'espai 3D coincideixi amb la textura.*

1.3.1 Estructura del món

La estructura del nostre món representa una malla 2D de cel·les de 64x64 unitats (veure figura 1.7). Com que cadascuna de les cel·les tenen igual dimensió d'amplada i alçada, s'estructurarà com un vector 2D de enters anomenat **mapa**, on el valor de cada enter hi constarà el codi del cub.

Un tipus estructurat de dades per al mapa seria com el que segueix,

```
Const
    DIMENSIO_CEL·LA = 64   {La dimensió de cel·la es igual en amplada que altura}
    MAX_CELES_Z = 12
    MAX_CELES_X = 12
fconst

tipus
   Mapa : taula[1...MAX_CELES_Z][1...MAX_CELES_X] de enter
ftipus
```

1.3.2 Codificació del mapa

Cadascuna de les enters del mapa pot contenir el valor '0', el qual representa que no hi ha cub a la cel·la, altrament hi ha un cub. Aquests valors fan referència al numero de textura (veure figura 1.8) la qual mapejarà el cub respectiu.

Per exemple, en el següent codi hi tenim la codificació d'un mapa ray-casting,

```
const char Mapa[MAX_CELES_Z+1][MAX_CELES_X+1] =
{
        //
        // '0'representa que no hi ha cub,
        // altrament hi ha cub.
        // Numero diferent de '0'fa referència
        // a la textura per representar el cub
        // gràficament en el procés de renderitzat.

        0,0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0,1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        0,1, 3, 3, 0, 0, 0, 0, 0, 3, 3, 1,
        0,1, 3, 3, 0, 0, 0, 0, 0, 3, 3, 1,
        0,1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
        0,1, 0, 0, 0, 4, 4, 4, 0, 0, 0, 1,
        0,1, 0, 0, 0, 4, 6, 4, 0, 0, 0, 1,
        0,1, 0, 0, 0, 4, 0, 4, 0, 0, 0, 1,
        0,1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
        0,1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
        0,1, 3, 3, 0, 0, 0, 0, 0, 3, 3, 1,
        0,1, 3, 3, 0, 0, 0, 0, 0, 3, 3, 1,
        0,1, 2, 2, 2, 5, 5, 5, 2, 2, 2, 1

};
```

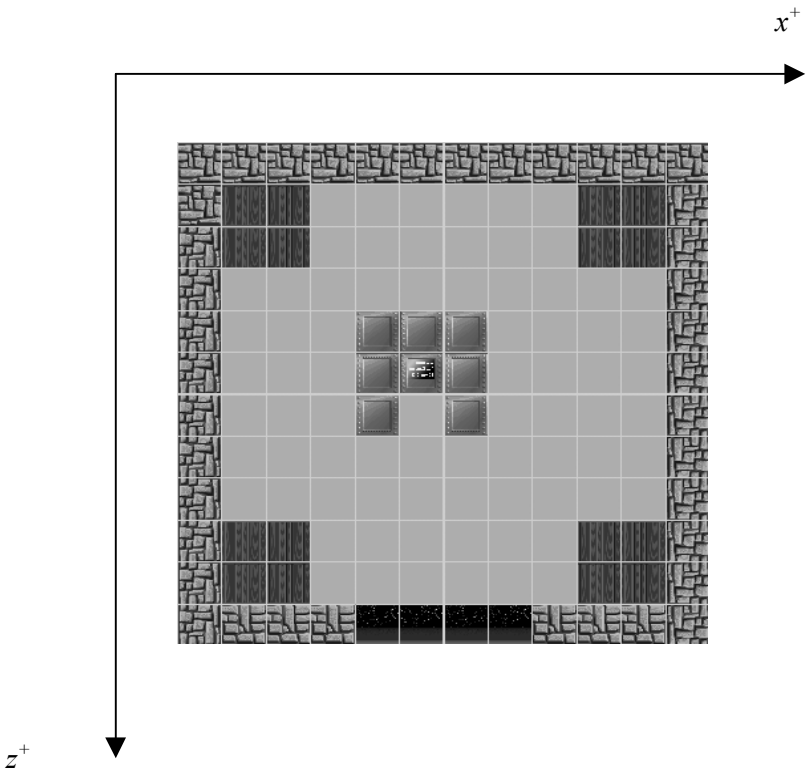


figura 1.10 *Representació de la malla 2D en el món 3D vist per sobre, amb la textura corresponent.*

A la figura 1.10 hi tenim la representació del món amb els cubs texturitzats, segons la codificació anterior.

Cal fer una observació a la figura 1.10 que el sentit positiu de Z s'ha establert cap avall per que sigui coherent al sentit del vector 2D, ja que la fila 0 es la posició inicial i la fila 12 és la final. Per qüestions de disseny, la fila i columna 0 no es tenen en compte.

1.5 Gestió de la informació del jugador

El jugador representa l'element camera (o punt d'observació) dins el món 3D, que l'usuari farà moure pel pla XZ del mapa mitjançant els perifèrics d'entrada (teclat o comandament) de l'ordenador o consola.

1.1 atributs del jugador

1.1.1 Coordenada X,Z del jugador

Es necessari definir les coordenades del jugador que recorren al pla XZ del mapa. En aquesta documentació farem referència a les coordenades de jugador com $(\mathbf{Xj}, \mathbf{Zj}) \in \mathbb{Z}^2$.

1.1.2 Direcció del jugador

La direcció en la qual mira el jugador es representarà mitjançant el seu angle equivalent. El valor d'aquest angle estarà comprès a l'interval [0º...360º) i serà tipus real, ja que els seus increments seran valors de coma flotant.

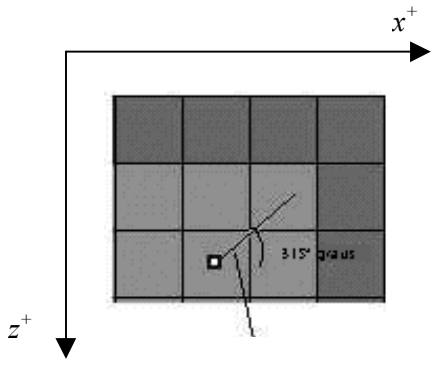


Figura 1.11 *Situació jugador en el mapa. En el dibuix, l'angle de direcció és 315º.*

L'angle de direcció s'incrementarà amb un valor constant d'angle quan es premi la tecla de *dreta* i, amb el mateix valor, és decrementarà quan es premi la tecla *esquerra*. El mínim increments/decrements d'angle va en funció de la resolució horitzontal de pantalla que tenim, però d'això ja en parlarem amb més detall a la secció 1.2.

1.1.3 Camp de visió del jugador (FOV)

El jugador ha de veure el que té davant d'ell. Per això, necessitem definir un camp de visió o, en anglès, denominat *Field Of View* (FOV). El FOV determina en quina amplada veu el món davant del jugador(v*eure figura 1.12*). El humans tenen un camp de visió superior als 90º, no obstant, un FOV de 90º no es visualitza correctament en una renderització de tipologia ray-casting. Quasi tots els motors ray-casting utilitzen un FOV de 60º.

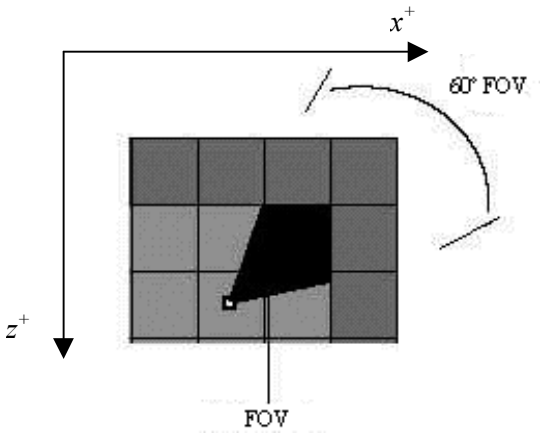


Figura 1.12 *Situació del nostre jugador en el mapa amb un FOV de 60º*

1.1.3 Altura del jugador

L'altura del jugador en aquest tipus motor sempre serà la meitat de l'altura de paret (o cub) o sigui que, al nostre cas, l'altura del jugador **sempre serà de 32 unitats**.

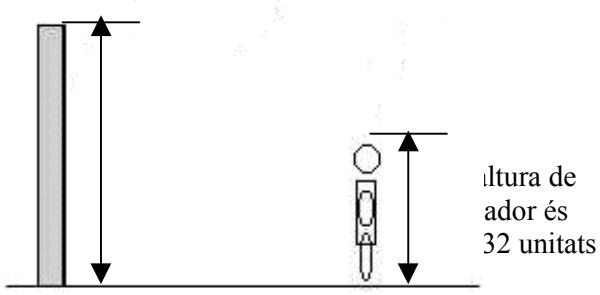


Figura 1.13 *Relació altura jugador-paret*

1.1.4 Definició del tipus jugador

El tractament de la informació de jugador dins pseudocodi o codi de programa l'efectuarem a partir del tipus jugador, una tupla que a continuació es defineix.

```
TUPLA tJugador

        X,Z : enter          { Les coordenades Xj i Zj }
        AngleDireccio: real  { L'angle direcció de l'observador}

FTUPLA
```

1.2 Actualització dels atributs

L'actualització dels atributs del jugador s'efectuarà segons els esdeveniments generats per l'usuari mitjançant el teclat o comandaments de la consola. En aquest procés es modificarà l'estructura de jugador passada per paràmetre.

1.2.1 Actualització del angle de direcció

L'angle de direcció s'actualitzarà **només quan l'usuari premi les tecles o controls direccionals *esquerra* o *dreta*** (← , →),

PSEUDOCODI

```
Si ApretaTeclaDreta() llavors
       DireccioJugador ← AngleDireccioJugador + IncrementAngle
Altrament
    Si ApretaTeclaEsquerra() llavors
       DireccioJugador ← AngleDireccioJugador – IncrementAngle
    FSi
FSi

{ Corregim si ens hem superat els 360 º o és negatiu}

Si DireccioJugador >= 360 o DireccioJugador < 0 llavors {Corregim}
       DireccioJugador ← abs(360 – abs(DireccioJugador))
FSi
```

L'increment d'angle més petit que podem tenir va en funció de la resolució horitzontal de pantalla i del camp de visió del nostre jugador.

La GBA té una resolució horitzontal de 240 columnes. Si tenim un FOV de 60º, llavors l'increment d'angle serà, el mínim canvi perceptible en el que vegi el jugador,

```
IncrementAngle ← 60 / 240 = 0.250° / columna
```

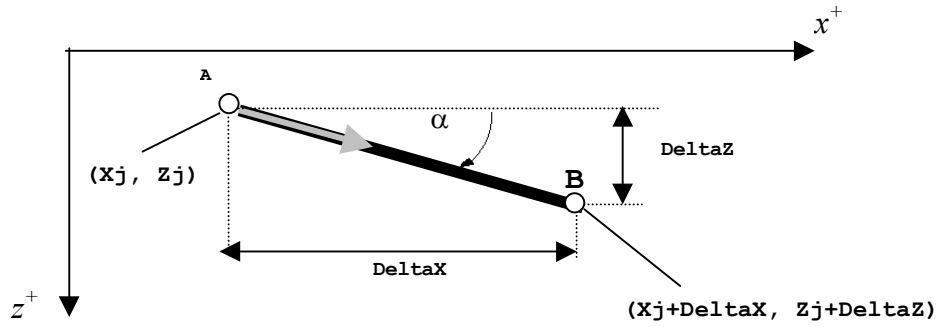
Aquesta és la resolució de gir més petita que pot realitzar el nostre jugador, podent fer girs a porcions més grans. El principal efecte d'aquest increment es veurà al procés de renderització (secció 1.6).

1.2.2 Actualització de les coordenades XZ.

Les coordenades XZ del jugador s'actualitzaran **només quan l'usuari premi la tecla o control direcció *endavant* o *enrera* (↑,↓)**, sumant o restant unes deltes $\in \mathbb{Z}^2$ segons el resultat del seu signe. Aquestes deltes s'aconsegueixen a partir del angle de direcció del jugador.

- **<u>Actualització XZ quan l'usuari prem *endavant*.</u>**

A la figura 1.14, s'estudia una situació d'exemple en que l'usuari prem la tecla *endavant*. Segons l'angle direcció del jugador observat a la figura 1.14, el resultat serà un desplaçament positiu a les coordenades de jugador perquè les deltes (*DeltaX* i *DeltaZ* $\in \mathbb{Z}^2$) seran positives, resultants d'aquest angle. Les deltes desplaçaran el jugador des del punt **A** al punt **B** (veure figura 1.14).



On:

▬ = Recorregut del jugador des de A fins a B.
→ = Direcció Jugador.
α = Angle direcció del jugador.
DeltaX = Un increment de jugador en X.
DeltaZ = Un increment de jugador en Z.

figura 1.14 *Exemple d'avançament jugador quan l'usuari prem la tecla de direcció endavant.*

Quan l'usuari prem *endavant*, el valor de les deltes XZ s'aconsegueix amb les següent formules.

```
DeltaX ← cosinus(AngleDireccioJugador)*VelocitatJugador      Equació 1.1
DeltaZ ←   sinus(AngleDireccioJugador)*VelocitatJugador      Equació 1.2
```

La velocitat jugador (VelocitatJugador) és el numero d'unitats màximes que es mouen en cada avançament del jugador. En la pràctica s'ha provat que una velocitat de jugador de 16 unitats es correcte.

- **<u>Actualització XZ quan l'usuari prem *enrera*.</u>**

  Quan l'usuari premi *enrera,* només s'haurà de invertir les deltes obtingudes a l'equació 1.1 i 1.2.

```
DeltaX ← -DeltaX
DeltaZ ← -DeltaZ
```

1.2.3 Funció ActualitzarJugador().

Segons les explicacions anteriors, la funció *ActualitzarJugador()* en pseudocodi, seria el següent,

```
Const
    VELOCITAT_GIR     = 0.25 * 16 = 2
    VELOCITAT_JUGADOR = 16
    DISTANCIA_PARET   = 32
Const



accio ActualitzarJugador(e/s Jugador: tJugador)
var
  DeltaX,DeltaZ :enter
fvar

 DeltaX ← cos(Jugador.AngleDireccio)*VELOCITAT_JUGADOR
 DeltaZ ← sin(Jugador.AngleDireccio)*VELOCITAT_JUGADOR


 si TeclaDreta() llavors { Actualitzem angle direcció jugador cap a l'esquerra}

    Jugador.AngleDireccio ← Jugador.AngleDireccio + VELOCITAT_GIR

    si(Jugador.AngleDireccio ≥ 360) llavors
       Jugador.AngleDireccio ← Jugador.AngleDireccio - 360
    fsi

 fsi

 si TeclaDreta() llavors { Actualitzem angle direcció jugador cap a la dreta}

    Jugador.AngleDireccio ← Jugador.AngleDireccio - VELOCITAT_GIR
    si(Jugador.AngleDireccio < 0) llavors
            Jugador.AngleDireccio ← Jugador.AngleDireccio + 360
    fsi
 fsi


  si(TeclaEndavant()) llavors { És suma les deltes a les respectives coordenades de jugador}

    Jugador.X ← Jugador.X + DeltaX
    Jugador.Z ← Jugador.Z + DeltaZ

  fsi

 si (TeclaEnrera()) llavors { Es suma el negat de les deltes a les coordenades XZ de jugador}

    {Es nega les deltes, pel fet de que el jugador marxa en sentit oposat}

    DeltaX ← -DeltaX
    DeltaZ ← -DeltaZ
```

1.6 La renderització, 1era versió

1.6.1 Introducció.

La renderització d'escenaris basat en un ray-casting consta llançar tants raigs com columnes de pantalla disposem i, llavors, representem la paret per tires degudament escalades segons la distància a l'observador. Per exemple, la GBA disposa de 240 columnes gràfiques, doncs, en aquest cas, la renderització constarà d'un escenari format per 240 tires de paret.

La renderització d'una tira de paret té dos etapes:

1. **Etapa de geometria.**
2. **Etapa de la renderització de tira.**

- **<u>Etapa de geometria</u>**

  En la aquesta etapa, es realitzen càlculs geomètrics necessaris per obtenir els paràmetres del renderitzat de tira. L'etapa de geometria **es resumeix en traçar d'un raig** a l'espai de mapa, que parteix des del jugador (o observador) fins a topar amb una paret (cub diferent de 0). Al següent exemple mostra la projecció d'un raig dins el FOV del jugador, traçat des de la posició del jugador fins a intesercar una paret.

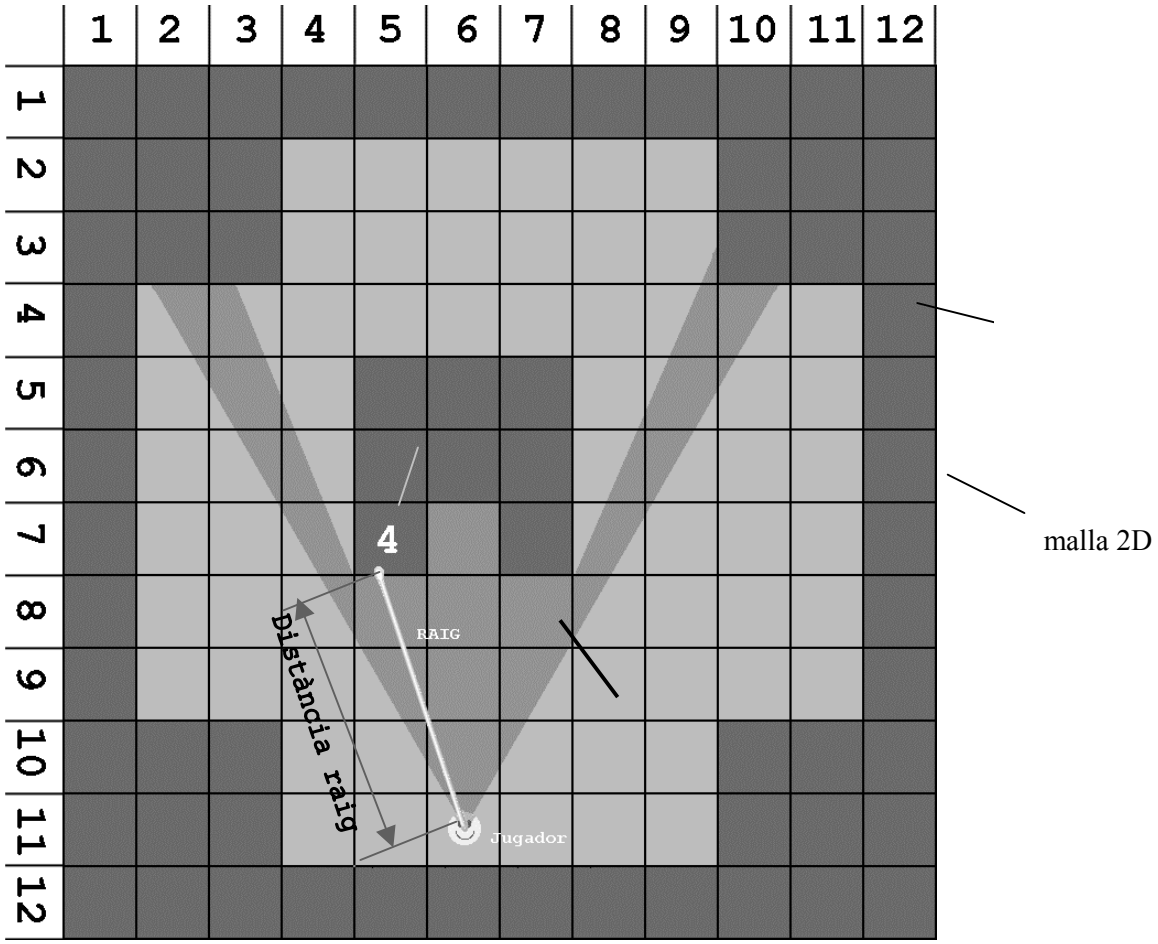


figura 1.15 *Exemple de projecció d'un raig, des del jugador fins a la paret.*

- **Etapa de renderització**

L'etapa del renderitzat consta del pintat d'una tira vertical. Aquesta tira serà pintada a la columna de pantalla per on s'hagi llançat el raig, en un offset en *y* degudament posicionat de pantalla.

La llargada d'aquesta tira serà de *y2d* pixels (figura 1.17). *Y2d* representa el numero de pixels equivalent a la projecció a pantalla, calculat a partir de la **distància** del raig (figura 1.15).

Si es vol pintar la tira de paret amb la textura del cub intesercant, l'etapa geomètrica retornarà la component *u* de textura que es mantindrà fix durant tot el pintat de tira. I per últim, es calcularà un increment adequat pel recorregut en *v* per representar la tira de gràfica de textura original (figura 1.16), al número de pixels *y2d*.

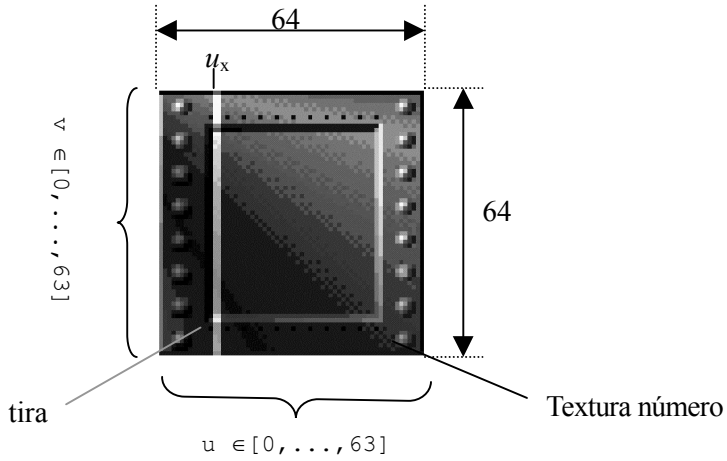


Figura 1.16 *Tira gràfica de textura selecionada per u_x*

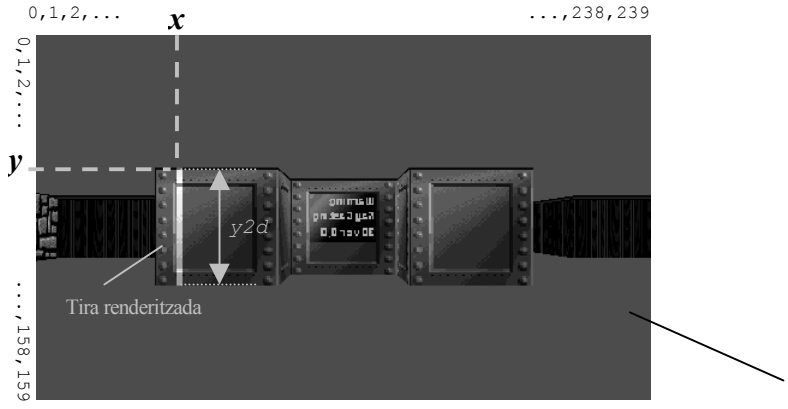


Figura 1.17 *Exemple de renderitzat de tira.*

Aquesta és precisament la restricció geomètrica del ray-casting. En ves de traçar un raig per pixel a la pantalla, com passava amb ray-tracing, es traça un raig per columna gràfica de pantalla (pla projecció). El raig de l'extrem esquerra del *FOV*, que estarà a 30º de l'esquerra de la l'angle direcció del jugador, es projectarà a la columna 0 de la pantalla. El raig extrem

dret, que estarà a 30º de l'esquerra de la direcció del jugador, serà projectat a la columna 239 de la pantalla (*fig. 1.18*).

A continuació presentem els passos, en forma simplificada, al pseudocodi corresponent:

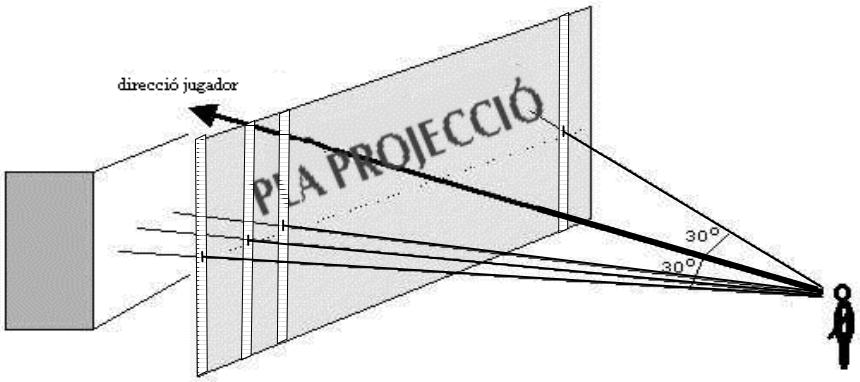


figura 1.18 *raigs jugador que intersecten a les columnes del pla de projecció*

1. Basat en l'angle de la posició del jugador que tenim, es resta 30º respecte l'angle de la direcció del jugador.
2. Començant des de la columna 0:
   a. Es traça un raig des de l'observador fins al primer objecte intesercat, que en aquest cas es la paret. El raig es una línia "imaginaria" que s'estén des del jugador (figura 1.19) dins el mapa. Per qüestions de disseny, el raig en representarà en dos parts: Una part intesercarà les columnes de malla anomenat *RaigX* i l'altre part instersectarà les files de malla anomenat *RaigZ*.
   b. Es tracem els dos raigs. Cada raig seguirà el seu curs fins que col·lisiona amb una paret.
   c. Quan *RaigX* o *RaigZ* col·lisiona amb una paret, es guarda la distància que es té des de el jugador a la paret (La distància és igual a la longitud del raig), la columna i la referència de textura a pintar de cada raig.
3. De les dues parts del raig llançat ens quedem amb la part que ha retornat la distància més curta i pintem la tira en funció dels seus atributs. (Distància, columna de textura i la referència de textura).
4. Sumem l'angle d'increment, per projectar el següent raig. Aquest angle d'increment està condicionat al FOV i l'amplada de pantalla. Repartirem els 240 raigs en un FOV de 60º, per tant l'increment d'angle és:

```
FOV div AMPLADA_PANTALLA = 60 div 240 = 0.250 °/columna)
```

5. Repetir els passos 2, 3 i 4 per cada subseqüent columna fins a llançar 240 raigs.

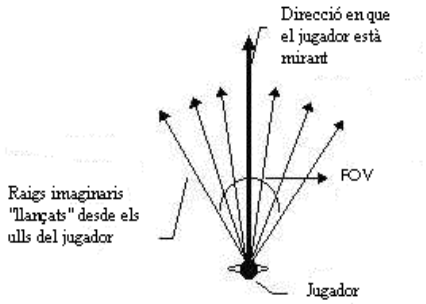


Figura 1.19 *Jugador llançant raigs imaginaris segons el FOV.*

I la funció de renderització basat en els passos anteriors és,

PSEUDOCODI

```
Accio Renderitzar(e Jugador:tJugador)
  var
       AngleRaig, DistanciaRaigX, DistanciaRaigZ:real
       ColumnaActual, TexturaX, TexturaZ, uX, uZ: enter
  fvar

  {1.- Basat en el FOV que tenim,  es resta 30° (meitat del FOV)}

  AngleRaig ← Jugador.AngleDireccio – 30.0

   Si AngleRaig < 0 llavors { Estem per sota els 360°, cal corregir}
       AngleRaig ← AngleRaig + 360.0
   FSi

   Per ColumnaActual desde 0 fins a 239 pas +1 fer

        { 2.- Es llença les dos parts del raig (Etapa geomètrica)}

        DistanciaRaigX ← RaigX(Jugador.X, Jugador.Z,AngleRaig,TexturaX,uX)
        DistanciaRaigZ ← RaigZ(Jugador.X, Jugador.Z,AngleRaig,TexturaZ,uZ)

        { Es traçarà la tira segons els atributs de la part de raig més propera al jugador }

        Si DistanciaRaigX < DistanciaRaigZ llavors
               {3.- Es traçar la tira segons els atributs de raig X}

               PintarTira(MatriuVideo,Textura[TexturaX],DistanciaRaigX,uX,ColumnaActual)

        Altrament
               {3.- Es traçar la tira segons els atributs de raig Z}
               PintarTira(MatriuVideo,Textura[TexturaZ],DistanciaRaigZ,uZ,ColumnaActual)

        FSi

        AngleRaig ← AngleRaig + IncrementAngle {IncrementAngle = 0.250}

        Si AngleRaig > 360.0 llavors { Si estem per sobre els 360 °,cal corregir  }
             AngleRaig ← AngleRaig – 360
        FSi

   FPer


FAccio
```

D'aquesta funció cal implementar les funcions *RaigX*, *RaigZ* i *PintarTira*, que explicarem a les seccions 1.6.2.1, 1.6.2.2 i 1.6.3 respectivament.

1.6.2 Etapa geomètrica, 1era versió: El raig

En aquesta versió, l'etapa geomètrica llença un raig separat en dos parts. Una part del raig que inteserca les columnes de la malla 2D denominat *RaigX* i una altre part del raig que inteserca les files de la malla 2D denominat *RaigZ*. Les dues parts retornaran la distància de la paret col·lisionada, entre altres informacions. **Sempre ens quedarem amb la informació de la part del raig que retorni la distància més petita [2].**

1.6.2.1 Implementació de RaigX

Aquesta funció retorna la distància des de la posició del jugador a l'intersecció del raig amb alguna de columna de la malla 2D on la respectiva cel·la està ocupada per un cub (*figura 1.20*). Si el raig surt del límit mapa, es retorna la distància més llarga possible.

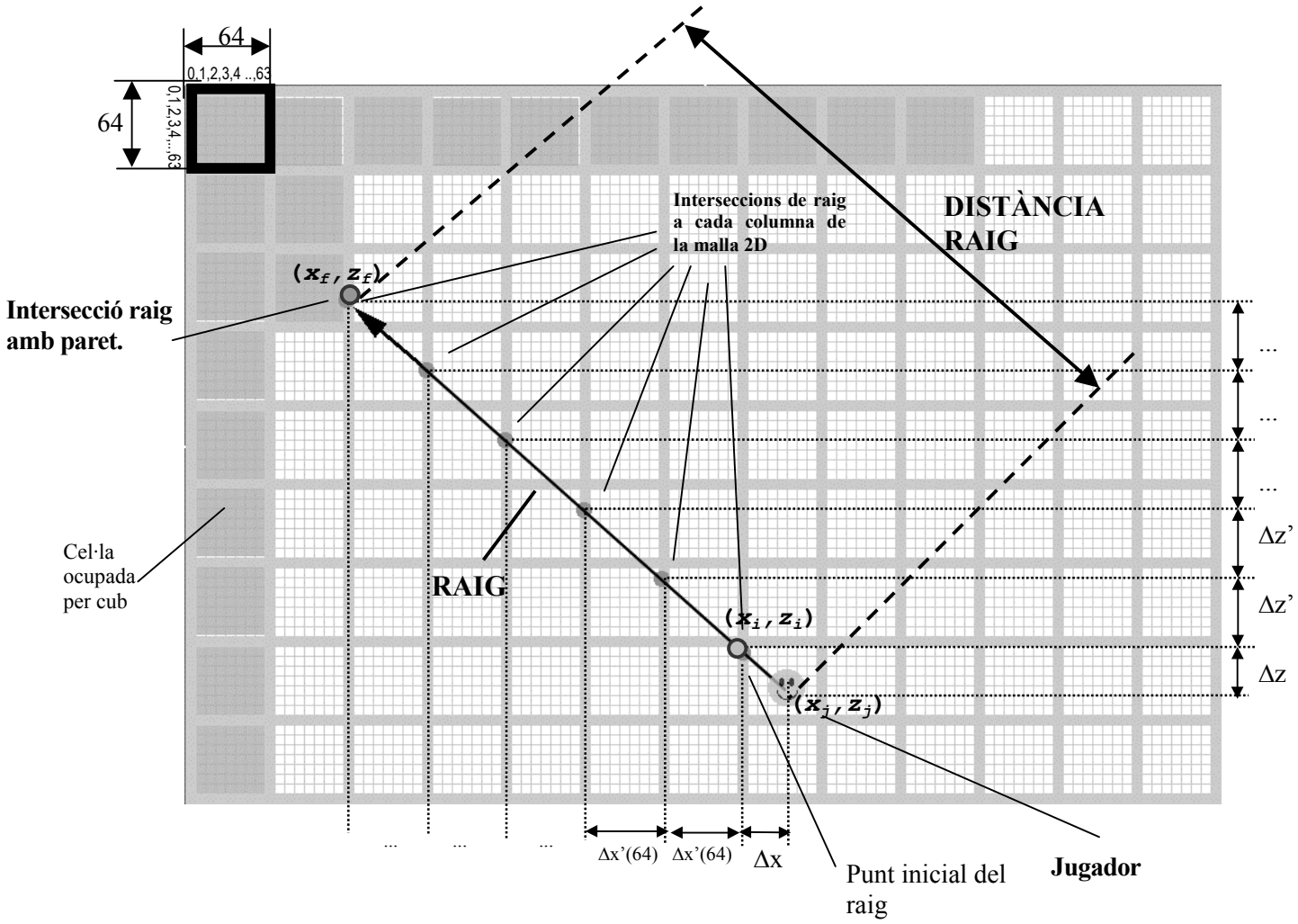


Figura 1.20 *Situació del jugador amb llançament d'un raig fins a arribar a intersecar amb una columna on la respectiva cel·la de la malla està ocupada per un cub. El procediment és simple, el raig parteix d'un punt inicial* ***(Xi,Zi)*** *i llavors es va traçant el raig amb uns increments calculats i constants* ***(Δx',Δz')****, intersecant les columnes de la malla 2D. Quan el raig interseca a una cel·la ocupada per cub, es pararà el traçat de raig i es retornarà la distància, component u de la textura i la referència de*

---

[2] Cal dir que això és equivalent a traçar un únic raig però, en aquesta aquesta versió, s'ha fet així per generar un codi més senzill.

- **El punt de partida del *RaigX* (Xi,Zi)**

Com es pot observar a la figura 1.20, el raig es llença des de la seva 1a intersecció (x_i, z_i), per motius d'optimització. L'optimització es estar en el fet que, un cop realitzat el desplaçament $(\Delta x, \Delta z)$, les successives interseccions que es donen a les

següents cel·les de la malla seran increments de 64 unitats en *X* i un constant desplaçament en *Z (Δz')* a l'espai de mapa.

- Obtenció de Xi

Xi coincideix amb l'offset d'una columna de la malla o offset inicial en X de cel·la (0,64,128,..). Xi és la coordenada inicial en X del *raigX* i es determina amb l'orientació de l'angle de raig. Si l'angle de raig està orientat a l'interval [90º,..,270º) (regió oest), Xi serà **l'offset inicial en X on es troba el jugador**. D'altra banda, si l'angle de raig està orientat a l'interval [270º,..,360º) o a l'interval [0º,..,90º) (regió est), Xi serà **l'offset inicial en X a la cel·la adjacent dretana del jugador**. A la figura 1.21 es mostra un exemple,

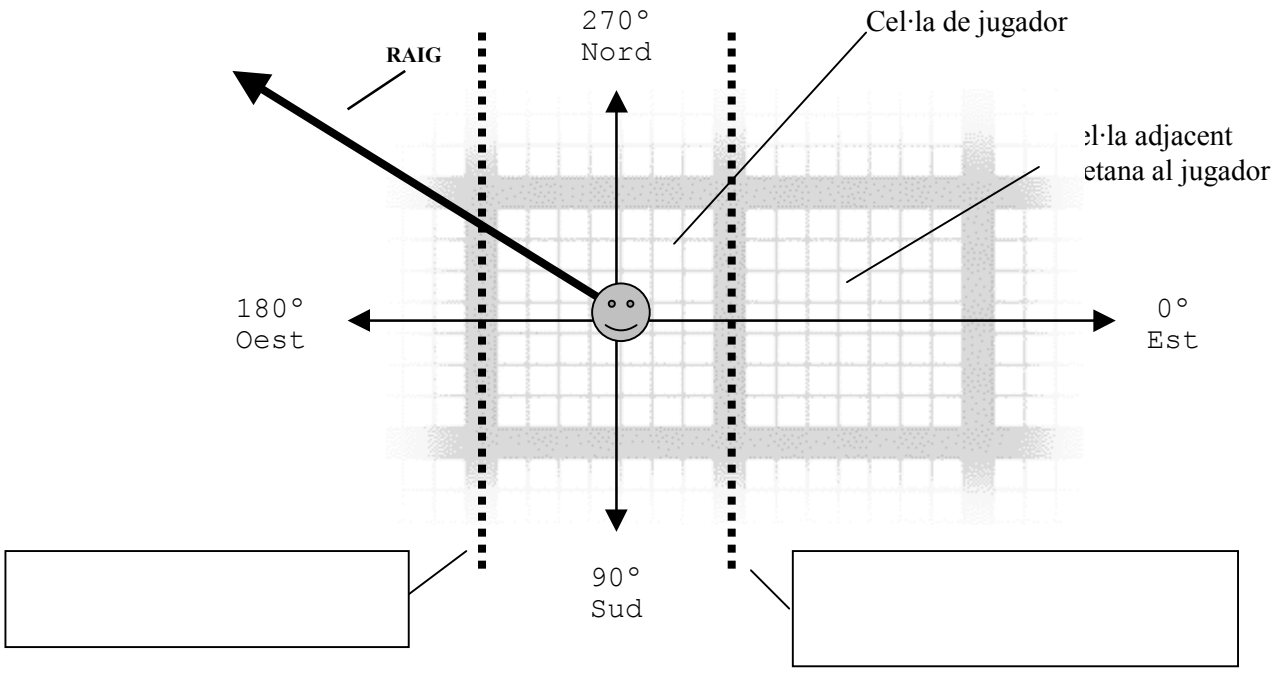


figura 1.21

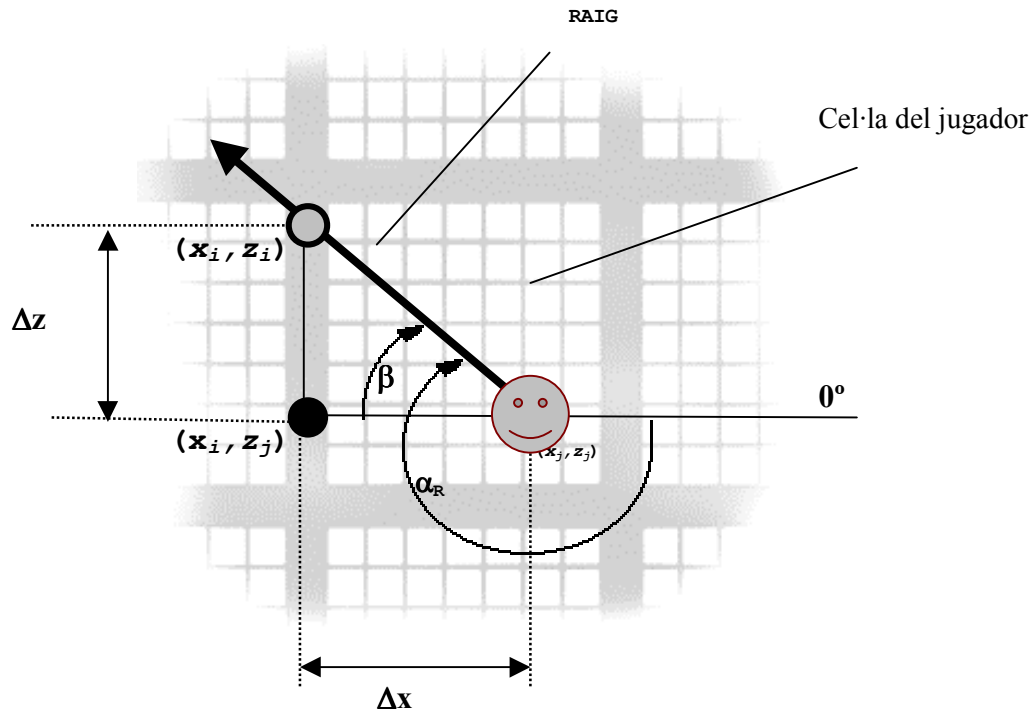
Observant l'exemple de la figura 1.21, veiem que el raig està orientat a l'interval [90º,..,270º) (regió oest), per tant **Xi seria l'offset inicial de la cel·la on es troba el jugador**.

El següent pseudocodi determina Xi en funció del angle de raig,

```
Si 90° ≤ AngleRaig i AngleRaig < 270° llavors {Partim de l'offset X inicial de cel·la
                                                on es troba el jugador}
      Xi ← (Xj div 64) * 64
Altrament { Partim a l'offset X de la malla adjacent dreta}
      Xi ← ((Xj div 64) + 1) * 64
Fsi
```

- Obtenció de Zi

A partir de *Xi* trobat a la secció anterior, podem trobar *Zi*. La figura 1.22 següent explica el procediment.



α_R = Angle del Raig.
β = Angle diferenciat del angle raig (α) en 180 º.

figura 1.22

L'increment Z*(Δz)* és conegut empleant la diferencia següent,

$$\Delta z = z_i - z_j \qquad \textit{Equació 1.3}$$

L'increment X *(Δx)* és,

$$\Delta x = x_i - x_j \qquad \textit{Equació 1.4}$$

La següent formula trigonomètrica relaciona els dos increments anteriors,

$$tan(\beta + 180) = tan(\alpha_R) = \frac{\Delta z}{\Delta x}$$

Aïllem **Δz,**

$$\Delta z = \Delta x \cdot tan(\alpha_R)$$

Assignant els valors pertinents als increments de les *equació 1.1* i *equació 1.2* ,

$$z_i - z_j = (x_i - x_j) \cdot tan(\alpha_R)$$

Aïllem el valor a trobar x_i,

$$\boxed{z_i = (x_i - x_j) \cdot tan(\alpha_R) + z_j} \qquad \textit{Equació 1.3}$$

• Càlcul dels increments de *RaigX*

Un cop situat el raig al punt de partida, és traça el recorregut del raig a partir d'uns desplaçaments constants en X i Z anomenats **Δx'** i **Δz'** respectivament. Aquests increments estan basats en el gruix de la cel·la. La principal optimització del algorisme és que, amb aquests increments s'obtindrà intersecció successives del raig amb la columna de la malla, de forma immediata.

- Increment X (Δx')

Aquest increment és igual a la dimensió de la malla en X, o sigui, 64 unitats. Això farà que, a cada increment X el raig es situï directament a la propera cel·la (veure *figura. 1.20*).

L'increment **Δx'** del raig serà de -64 unitats si la orientació del raig pertany a la regió oest i de +64 unitats en el cas de que estigui orientat a la regió est.

Mostrem el pseudocodi per determinar l'increment X en funció del angle de raig,

```
Si 90° ≤ AngleRaig i AngleRaig < 270° llavors  { el raig avançarà cap a l'esquerra }
        IncrementX ← -64
Altrament { el raig avançarà cap a la dreta }
        IncrementX ← +64
FSi
```

- L'increment Z (Δz')

Observant la figura 1.23,

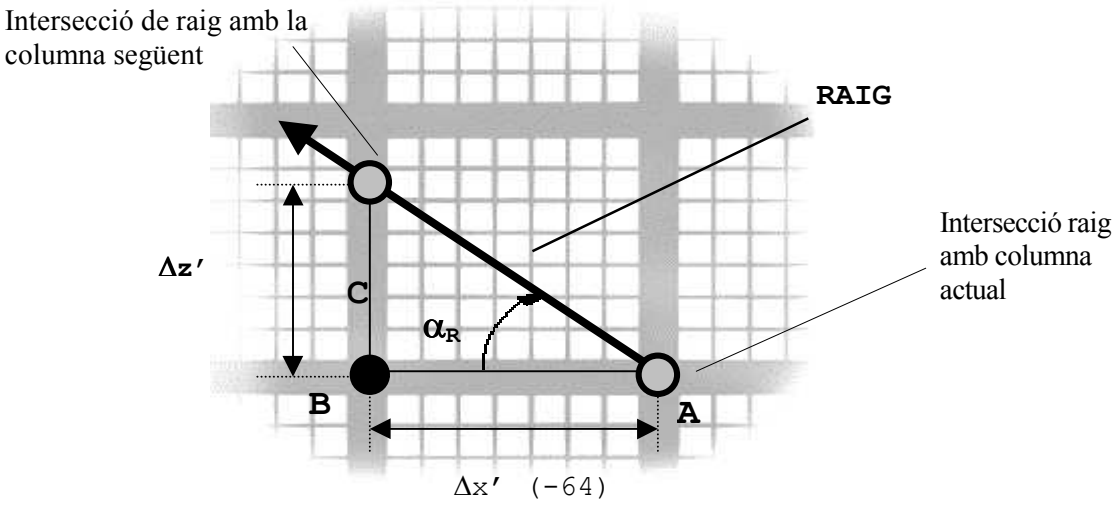


figura 1.23 *El raig es desplaça a la següent intersecció de columna a columna, segons els increments Δx' i Δz'.*

L'increment Δx' és constant, el qual, com s'ha vist abans, només podia de ser 64 unitats i l'angle de raig també es constant durant tot el traçat. Llavors, l'increment Z serà , també, constant amb la següent fórmula,

$$IncrementZ \leftarrow IncrementX \cdot tangent(\alpha_R) = \Delta x' \cdot tangent(\alpha_R)$$

Cal observar a l'exemple de la figura 1.23, Δx és negatiu, i l'angle del raig (α_R) es troba entre 180º i 270º, llavors la tangent de α_R és positiu i per conseqüent el signe d'Δz és negatiu. Automàticament**,** Δx i tangent(α_R) farà el signe adequat a Δz'.

- **Avaluació del contingut de cel·la durant el traçat del raigX**

Ara, un cop hem posicionat el raig a la coordenada **(Xi, Zi)**, es comença a traçar el raig amb els increments $\Delta x'$ i $\Delta z'$ obtinguts anteriorment. A cada iteració, es sumen els increments tal que cada cop inteserca de forma automàtica a la següent columna de la malla. Cada columna de malla representa l'offset inicial d'una cel·la (0,64,128,etc).

Cal determinar la posició de cel·la en el mapa definit a la secció 1.3.2 referent a la columna de malla intesercada, per saber si la cel·la té cub. Si el contingut és diferent de 0 representarà cub i per tant, **s'haurà de parar el traçat del raig perquè la columna intesercada representarà una paret**. Si el contingut de cel·la és igual a zero es prosseguirà el traçat de raig.

Per esbrinar el contingut de la cel·la intesercada, utilitzarem les coordenades XZ del raig intesercant amb la columna de malla actual. Les coordenades XZ del raig s'anomenaran: **PosicioRaigX, PosicioRaigZ $\in \Re$.**

```
Si 90° ≤ AngleRaig i AngleRaig < 270° llavors

   PosicioMapaX ← PosicioRaigX div 64
   PosicioMapaZ ← PosicioRaigZ div 64 + 1

Altrament

   PosicioMapaX ← PosicioRaigX div 64 + 1
   PosicioMapaZ ← PosicioRaigZ div 64 + 1


Fsi

{Es fa entrega del contingut de la cel·la...}

ContingutCel·la ← Mapa[PosicioMapaZ][PosicioMapaX]

Si (ContingutCel·la ≠ 0) llavors {Està ocupada per cub}

      {Parar traçat i guardar informació necessària...}

Altrament {No hi ha cub}

      {Es continua el traçat del raig...}

Fsi
```

- **Dades a guardar quan el raig inteserca un cub**

Quan *RaigX* inteserca amb una cel·la ocupada (diferent de 0), s'atura el traçat del raig i és guarda les següents informacions: **L'identificador de textura, columna de la textura (segons la intersecció Z del raig amb paret -Zf-) i la distància del raig**,

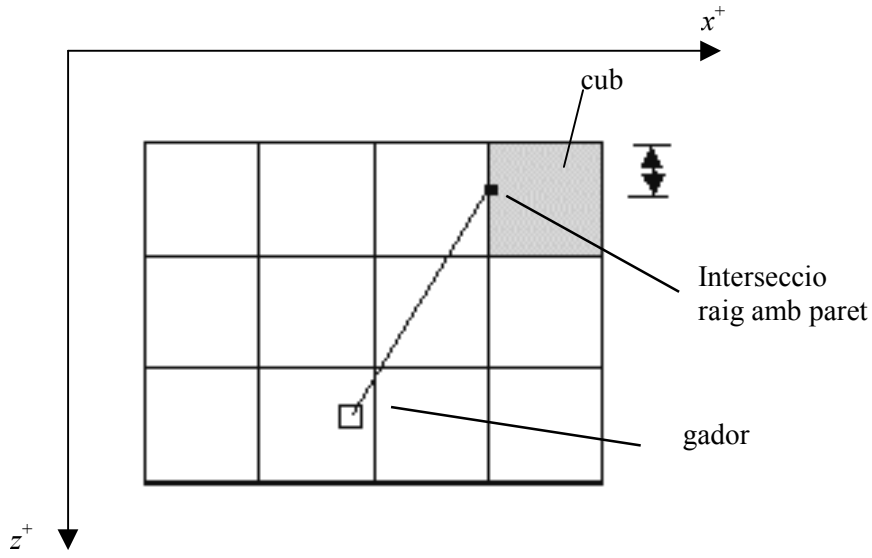
a) L'identificador de textura

En la secció anterior s'ha vist com obtenir el contingut de cel·la intesercada pel raig. El contingut ens explica la referència de textura, la qual es farà referència a l'hora de pintar la tira de paret.

b) La columna de textura (*u*)

Hem de trobar quina columna de textura (u) pintarà la tira de paret. Per això, fem referència a la intersecció del raig amb paret (x_f, z_f), en coordenades de món.

Com s'havia dit, les dimensions de la cel·la coincideix amb les dimensions de textura (64x64). Així doncs, podem saber quina serà la tira de textura pel pintat si sabem què val z_f **mòdul** 64.



Per tant,

$$u = Z_f \ \textbf{mod}\ 64$$

Com s'ha mencionat, Z_f és la component Z de la ultima intersecció del raig (PosicioRaigZ). Com s'havia dit anteriorment, **PosicioRaigZ $\in \Re$** . Cal forçar que PosicioRaigZ sigui enter, perquè la textura esta definit en l'espai dels naturals. Per això, a la fórmula s'ha de afegir alguna funció que aproximi el real a un enter. Es farà servir la funció *ceil()*, la qual retorna el valor enter més positiu d'un real entrat per paràmetre.

$$\text{ColumnaTextura} = \text{ceil}(Z_f)\ \textbf{mod}\ 64$$

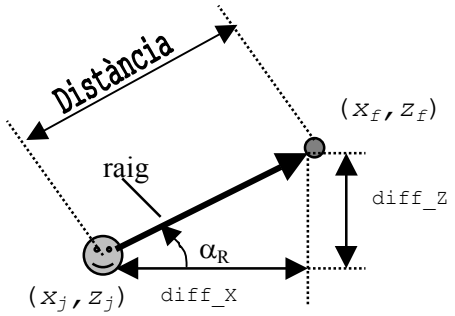
c) La distància

Si el raig surt del mapa es retornarà la **distància més llarga** possible del mapa. La distància més llarga seria la diagonal màxima de la malla 2D, i es calcula de la següent manera,

$$\text{DistànciaMesLlarga} = \sqrt{(12 \cdot 64)^2 + (12 \cdot 64)^2} = 1086 \text{ unitats}$$

Altrament, si el raig inteserca una paret es calcularà la distància des del jugador a la intersecció amb un cub.

Observant la següent figura,



On:

diff_Z = Zj – Zf
diff_X = Xj – Xf
α_R = Angle de raig.

Una manera de calcular la distància del raig és,

$$distancia = \sqrt{(diff\_X)^2 + (diff\_Z)^2} \qquad \textit{Equació 1.4}$$

L'arrel quadrada més els quadrats resulta una operació bastant costosa en temps d'execució. Aplicant trigonometria, hi ha dos maneres diferents d'obtenir la distància de manera més òptima que l'equació 1.4,

$$distancia = abs\left(\frac{diff\_x}{\cos(\alpha_R)}\right) \qquad \textit{Equació 1.5}$$

$$distancia = abs\left(\frac{diff\_z}{sin(\alpha_R)}\right) \qquad \textit{Equació 1.6}$$

Però ambdós presenten l'inconvenient d'una excepció de divisió per 0, a l'equació 1.5 quan l'angle raig és de 0º o 180º i a l'equació 1.6 quan l'angle raig és 90º o 270º.

Per evitar aquesta situació, es comprova quina de les dos diferències absolutes és més gran, per saber quina de les dos equacions és convenient utilitzar,

```
si abs(x_diff) > abs(z_diff) llavors {És convenient utilitza l'equació 1.5
                                     per evitar divisió entre 0.}

      Distancia ← abs(x_diff div cos(αR))

altrament {És convenient utilitza l'equació 1.6 per evitar divisió entre 0.}

      Distancia ← abs(z_diff div sin(αR))

fsi
```

- **L'implementació de la funció *RaigX***

Tenim tota la informació necessària per l'implementació de la funció RaigX. El procediment tracta d'un bucle que traçarà el raig al llarg del mapa utilitzant els valors calculats del punt de partida. A partir d'aquest punt, els valors d'incrementX i incrementZ fan avançar el raig a nivell de cel·la, fins a col·lisionar amb un bloc ocupat o fins sobrepassar els límits establerts del mapa. Tot seguit mostrem el pseudocodi de la funció RaigX,

```
Const
       DIMENSIO_CELA = 64
       MAX_CELES_X = 13
       MAX_CELES_Z = 13
       MAX_UNITATS_MAPA_X =  MAX_CELES_X*UNITATS_AMPLADA_CEL·LA = 13*64 = 832
       MAX_UNITATS_MAPA_Z = MAX_CELES_X*UNITATS_AMPLADA_CEL·LA = 13*64 = 832
FConst


Funció RaigX(e Xj,Zj:enter;AngleRaig:real
                e/s Textura, ColumnaTextura:enter) retorna real
var
       PosicioMapaX,PosicioMapaZ:enter
       PosicioRaigZ,Zi,IncrementZ,PosicioRaigX, Xi,IncrementX:real
fvar

Si (AngleRaig ≠ 90.0) i (AngleRaig ≠ 270.0) llavors { Angle vàlid }

     CalcularPassosRaigX(Xj,Zj,AngleRaig,Xi,Zi,IncrementX,IncrementZ);

     {S'obté la posició actual del raig en la matriu 2D}
     PosicioMapaRaigX(Xi,Zi,AngleRaig,PosicioMapaX,PosicioMapaZ);

     {Inicialitzem les posicions de partida del raig}

     PosicioRaigX ← Xi
     PosicioRaigZ ← Zi

     Si (Mapa[PosicioMapaZ][PosicioMapaX] = 0) i
          PuntDinsMapa(PosicioRaigX,PosicioRaigZ) llavors { Traçem el raig}

       repetir

         PosicioRaigX = PosicioRaigX + IncrementX
         PosicioRaigZ = PosicioRaigZ + IncrementZ
         PosicioMapaRaigX(PosicioRaigX,PosicioRaigZ,AngleRaig,PosicioMapaX,PosicioMapaZ);

       fins que  (Mapa[PosicioMapaZ][PosicioMapaX] ≠ 0) i
                  no PuntDinsMapa(PosicioRaigX,PosicioRaigZ)
     FSi

    Si no PuntDinsMapa(PosicioRaigX,PosicioRaigZ) llavors{el raig ha sortir dels limits }

       { Es retorna la distància més llarga }

       RaigX ← DISTANCIA_MES_LLARGA

    Altrament { retornem informacions corresponents }

       { Textura }
        Textura = Mapa[PosicioMapaZ][PosicioMapaX]-1;

       { Columna de textura }
        ColumnaTextura = enter(PosicioRaigZ) mod DIMENSIO_CELA

       { Retornem la distància }
         RaigX ← Distancia(Xj,Zj,PosicioRaigX,PosicioRaigZ,AngleRaig)
    Fsi
Altrament { Error, per que els calculs de tan(90) o tan(270) són infinits}
  { Retornem la distància més llarga }

   RaigX ←DISTANCIA_MES_LLARGA
Fsi
FFunció
```

La funció *CalcularPassosRaigX()*, calcula el punt de partida del raig i els seus increments del traçat. El pseudocodi es detalla a continuació,

```
    {Calcul del punt de partida del raig(Xi,Zi) i dels increments X i Z (Δx,Δz)}

    Accio CalcularPassosRaigX(e Xj,Zj:enter;AngleRaig:real
                              e/s Xi,Zi,IncrementX,IncrementZ:real)
    Var
     ComencamentX:enter
    FVar

   ComencamentX ← (Xj div DIMENSIO_CELA)*DIMENSIO_CELA
   IncrementX ← DIMENSIO_CELA

   si((AngleRaig < 90.0) o (AngleRaig > 270.0)) llavors {El raig s'orienta a l'est }

        Xi ← ComencamentX + DIMENSIO_CELA {Offset a la malla adjacent dreta d'on es troba el
                                    jugador}

       si(AngleRaig = 0.0)llavors {IncrementZ = 0 i Zi = Zj}

                Zi ← Zj
                IncrementZ ← 0

       altrament{ s'aplica càlculs fets a la secció 1.6.2.1.1 i 1.6.2.1.2 per esbrinar Zi,Δz}

                Zi ← (Xi-Xj)*tan(AngleRaig);
                IncrementZ = IncrementX*tan(AngleRaig)
       fsi
   altrament {El raig s'orienta a l'est }

        Xi ← ComencamentX {Offset a la malla on es troba el jugador.}

       IncrementX ← - IncrementX;

        si(AngleRaig = 180.0) llavors {IncrementZ = 0 i Zi = Zj}

           Zi ← Zj
           IncrementZ ← 0

        altrament{ s'aplica càlculs fets a la secció 1.6.2.1.1 i 1.6.2.1.2 per esbrinar Zi,Δz}

            Zi ← (Xi-Xj)*tan(AngleRaig) + Zj
            IncrementZ ← IncrementX*tan(AngleRaig)
        Fsi
Fsi
FFunció
```

La funció *PosicioMapaRaigX()*, retorna la l'offset inicial de casella segons l'angle de raig i les coordenades de jugador,

```
{Obté la casella en el mapa a l'intersecció corresponent per raigX }
accio PosicioMapaRaigX(e PosicioRaigX,PosicioRaigZ,AngleRaig:real;
e/s PosicioX,PosicioZ:real)

 si(AngleRaig > 270.0) o (AngleRaig < 90.0 ) llavors { Raig orientat a l'oest...}

   PosicioX = (PosicioRaigX div DIMENSIO_CELA) + 1;
   PosicioZ = (PosicioRaigZ div DIMENSIO_CELA) + 1;

 altrament { Raig orientat a l'est...}

   PosicioX = (PosicioRaigX div DIMENSIO_CELA);
   PosicioZ = (PosicioRaigZ div DIMENSIO_CELA) + 1;
 fsi
faccio
```

Tot seguit es mostra la funció que calcula la distancia respecte dos punts (observador intersecció) i l'angle de raig,

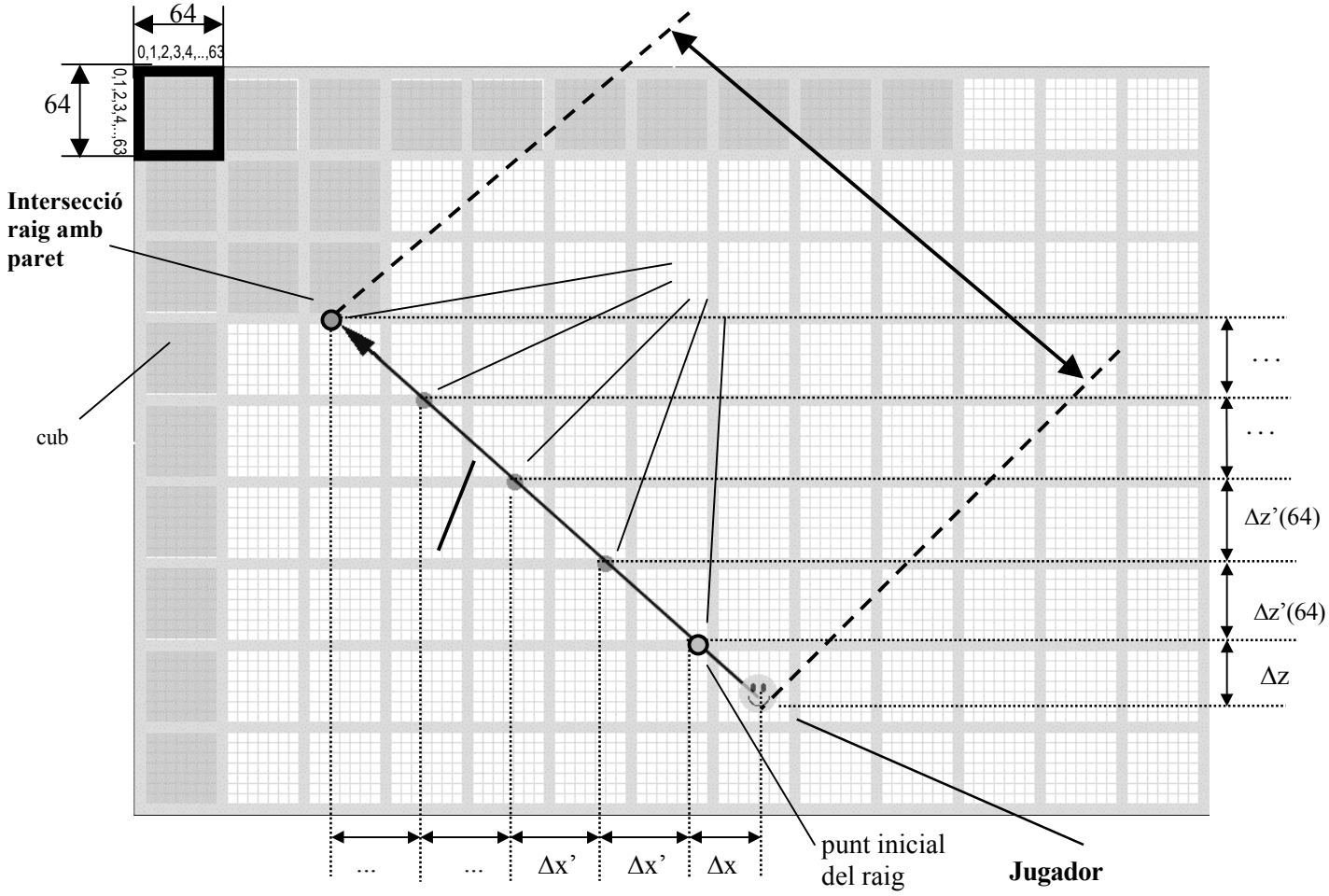
```
Funció Distancia(e Xj, Zj, Xf, Zf, AngleRaig: real) retorna real
Var
      x_diff, z_diff : real
fvar

  z_diff = Zj – Zf
  x_diff = Xj – Xf


  si abs(x_diff) > abs(x_diff) llavors {És convenient utilitza l'equació 1.5 per evitar divisió
       entre 0.}

       Distancia ← abs(x_diff div cos(α))

  altrament {És convenient utilitza l'equació 1.6 per evitar divisió entre 0.}

       Distancia ← abs(x_diff div sin(α))
   Fsi
Ffunció
```

La funció *PuntDinsMapa()* determina si el punt passat per paràmetre pertany al quadrat englobant del mapa,

```
{Retorna la perntença punt ∈ ℜ² passat per paràmetre pertany al dins el quadrat que
       engloba del mapa }

Funció PuntDinsMapa(e X,Z:real) retorna boolea

   PuntDinsMapa ← (X ≥ 0) i (X < MAX_UNITATS_MAPA_X) i (Z ≥ 0) i (Z <
          MAX_UNITATS_MAPA_Z)

FFunció
```

1.6.2.2 Implementació de RaigZ

RaigZ efectua un traçat de raig semblant a RaigX. La diferència està en què, el raig va intesercant la **files** de la malla.



- **El punt de partida del RaigZ (Xi,Zi)**

De manera anàloga a la secció 1.6.2.1 (punt partida raig X), el raig es llançarà des de la seva 1a intersecció (x_i, z_i). Cal observar en l'exemple de la *figura 1.25* que ara, la variable independent és *Z* (**PosicioRaigZ**) i la variable dependent és *X* (**PosicioRaigX**). Per aquest motiu es comença trobant la component *Z* inicial (Zi) per trobar la posterior *X* inicial (Xi).

- Obtenció de Zi

Zi és l'offset inicial de cel·la de d'on partirà el *raigZ* i es determina amb l'orientació de l'angle de raig. Si l'angle de raig està orientat a l'interval [180º,..,360º) (regió nord), **Zi serà l'offset inicial de la cel·la on és troba el jugador**. D'altra banda, si l'angle de raig està orientat a l'interval [0º,..,180º) (regió sud), **Zi serà l'offset inicial de la cel·la adjacent inferior a on és troba el jugador**. A la figura 1.26 es mostra un exemple,

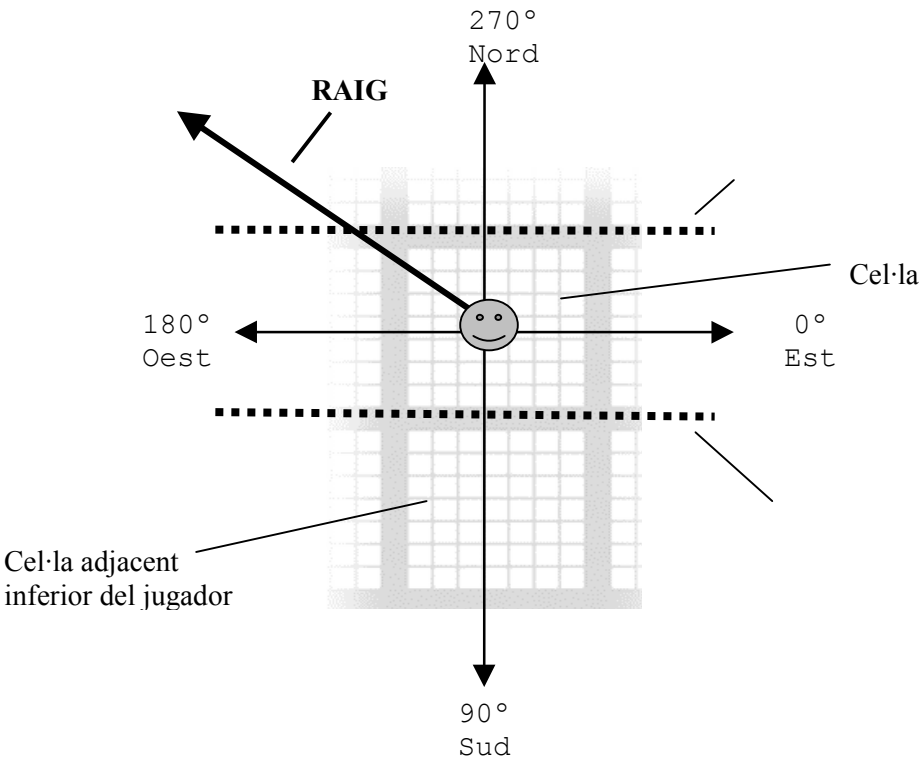


figura 1.26

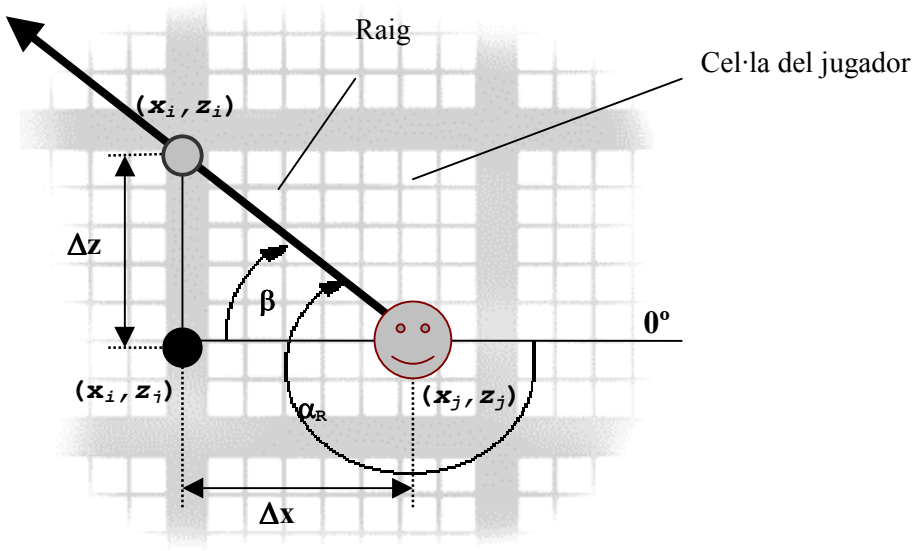
Observant l'exemple de la figura 1.26, veiem que el raig està orientat a l'interval [180º,..,360º) (regió nord), per tant **Zi** serà l'offset inicial de la malla on es troba el jugador.

A continuació es mostra el pseudocodi necesari per determina l'offset Z inicial en funció l'angle del raig,

```
Si AngleRaig ∈ RegioNord llavors {Serà l'offset inicial de la cel·la del jugador}
      Zi ← (Zj div 64) * 64
Altrament { Serà l'offset inicial de la cel·la adjacent inferior on es troba el jugador}
```

- **Obtenció de Xi**

S'explicarà els passos de l'obtenció de *Xi* a partir de la figura següent,



α_R = Angle del Raig.
β = Angle diferenciat del angle raig (α) en 180 º.

Figura 1.27

L'increment Z *(**Δz**)* és conegut empleant la diferencia següent,

$$\Delta z = z_j - z_i \qquad \textit{Equació 1.7}$$

L'increment X *(**Δx**)* és,

$$\Delta x = x_j - x_i \qquad \textit{Equació 1.8}$$

La següent formula trigonomètrica relaciona els dos increments anteriors,

$$tan(\alpha_R) = \frac{\Delta z}{\Delta x}$$

Aïllem ***Δx*,**

$$\Delta x = \frac{\Delta z}{tan(\alpha_R)}$$

Assignant els valors pertinents als increments de les equacions 1.7 i 1.8,

$$x_i - x_j = \frac{z_i - z_j}{tan(\alpha_R)}$$

Aïllem el valor a trobar, o sigui, x_i,

$$\boxed{x_i = \left(\frac{z_i - z_j}{tan(\alpha_R)}\right) + x_j} \qquad \textit{Equació 1.9}$$

- **Càlcul dels increments del RaigZ**

- Increment Z (Δz')

En el cas d'aquest raig, el seu traçat és independent en tot l'eix Z del mapa, per tant, la grandària de l'increment en Z és igual a la dimensió de la cel·la en Z, o sigui, 64 unitats. Això farà que, a cada increment Z el raig es situï directament a la propera fila de la malla.

Fent referència a la figura 1.25, la coordenada Z del raig és decrementarà si la orientació raig pertany a la regió nord i s'incrementarà en el cas de que estigui orientat a la regió sud.

Mostrem el pseudocodinecessari per determinar l'increment Z, en funció l'orientació del raig,

```
Si AngleRaig ∈ RegioNord llavors
        IncrementZ ← - 64
    Altrament
        IncrementZ ← +64
FSi
```

- L'increment X (Δx')

Ara, *l'increment X* és dependent a *l'increment Z*. *L'increment Z* és constant, del qual com s'ha vist abans, és de 64 unitats.

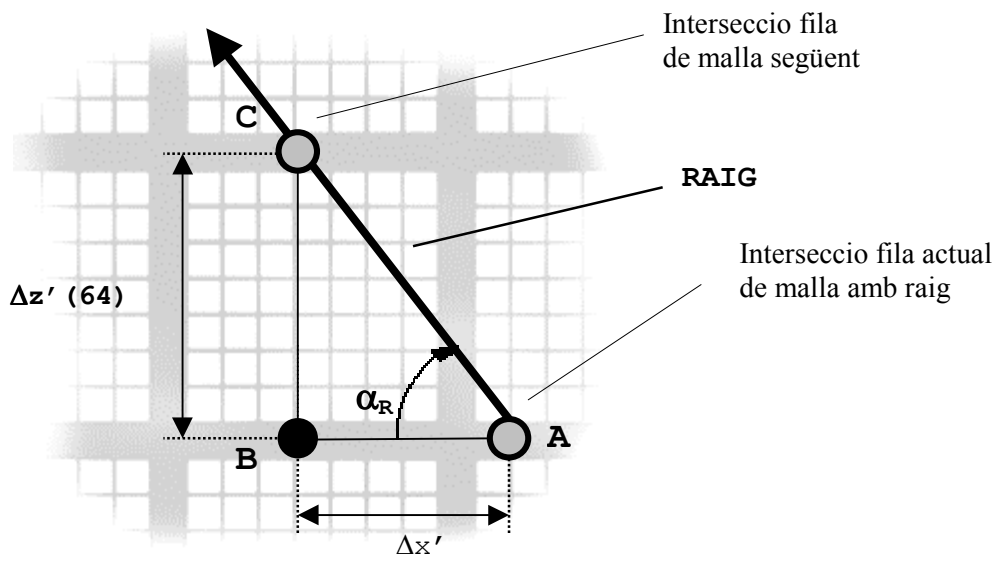


figura 1.28

La fòrmula següent resol l'increment X *(Δx)* constant,

$$\Delta x \leftarrow \Delta z' \ \textbf{div}\ \text{tangent}(\alpha_R)$$

De igual manera, *tangent(α_R)* i *Δz'* donen el signe adequat a *Δx* per que incrementi o decrementi.

• Avaluació del contingut de cel·la durant el traçat del raigZ

Ara, amb la intersecció **(xi,zi)** de partida obtinguda, es comença a traçat de raig amb els respectius increments calculats (Δx',Δz'). Segons la posició de raig, s'avaluarà la cel·la que esta intesercant a l'estructura de mapa per saber si esta ocupat per un cub.

La manera d'avaluar el contingut de cel·la amb les coordenades de raig **(PosicioRaigX, PosicioRaigZ)** per *RaigZ*, es mostra al següent pseudocodi,

```
Si AngleRaig ∈ RegióNord llavors

       PosicioMapaX ← PosicioRaigX div 64 + 1
       PosicioMapaZ ← PosicioRaigZ div 64

Altrament

       PosicioMapaX ← PosicioRaigX div 64 + 1
       PosicioMapaZ ← PosicioRaigZ div 64 + 1


Fsi

       {Es fa entrega del contingut de la cel·la...}

      ContingutCela ← Mapa[PosicioMapaZ][PosicioMapaX]

Si (ContingutCela ≠ 0) llavors {Hi ha cub}

     {Parar traçat i guardar informació necessària...}

Altrament {No hi ha paret}

       {Es continua el traçat del raig...}

Fsi
```

- **Dades a guardar quan el raig inteserca paret**

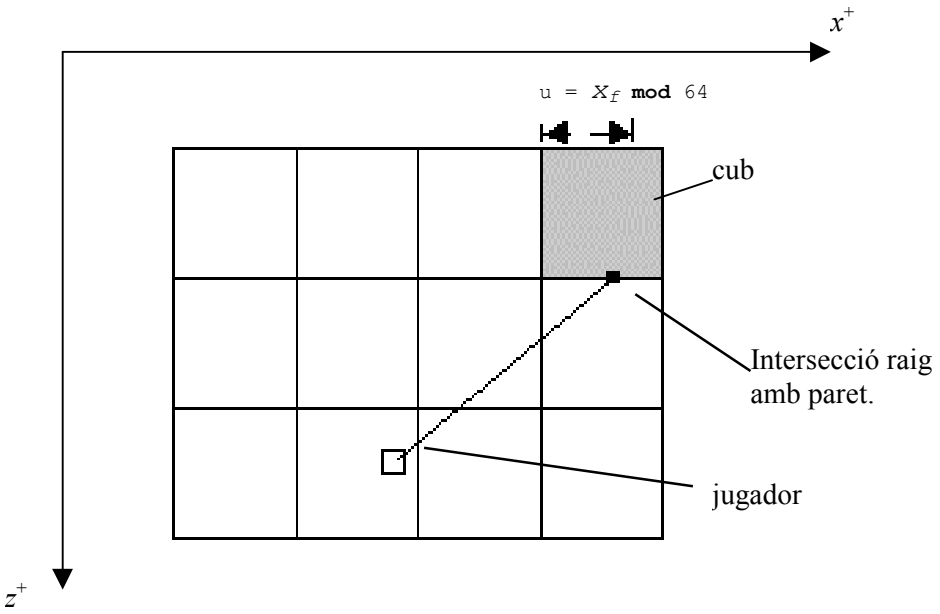
Quan el raig inteserca amb una paret, es para el traçat del raig i és guarda les mateixes informacions següents: **identificador de textura, la columna de textura (segons la intersecció raig paret en X –Xf-), i la distància del raig**.

a) La textura

Anteriorment, s'ha vist com obtenir el codi de bloc segons les posicions PosicioRaigX i PosicioRaigZ. El codi de bloc explica la referència de textura.

b) La columna de textura (*u*)

En aquí, cal saber la *component X* (x_f) del raig interseccionant per esbrinar quina tira de textura (*component u*) s'utilitzarà per mapejar la tira de paret. Només s'ha d'aconseguir la posició en X relativa a la cel·la del cub, com mostrem a la figura 1.29.



Donada la component X de la intersecció amb paret del raig, s'aplica el mòdul de l'amplada de cel·la (64 unitats), de tal forma que, directament s'obté la tira de columna pel mapejat.

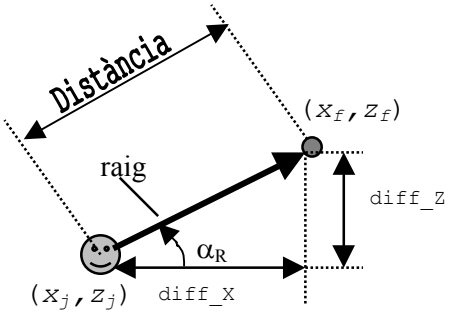
```
u = ceil(xf) mod 64
```

c) La distància

Igualment, es retornarà la **distància més llarga** del mapa en cas de que el raig surti del mapa.

$$\text{DistànciaMésLlarga} = \sqrt{(12 \cdot 64)^2 + (12 \cdot 64)^2} = 1086 \text{ unitats}$$

Altrament trobem la distància del jugador, fent refrència a la següent figura,



On:

diff_Z = Zj – Zf
diff_X = Xj – Xf
α_R = Angle de raig.

El pseudocodi és el següent,

```
si abs(x_diff) > abs(z_diff) llavors {És convenient utilitza l'equació 1.5
                                      per evitar divisió entre 0.}
      Distancia ← abs(x_diff/cos(αR))
altrament {És convenient utilitza l'equació 1.6 per evitar divisió entre 0.}
      Distancia ← abs(z_diff/sin(αR))
fsi
```

- # L'implementació de la funció RaigZ

Ara tenim tota la informació necessària per l'implementació de la funció *RaigZ*. El procediment tracta d'un bucle que traçarà el raig al llarg del mapa utilitzant els valors calculats del punt de partida. A partir d'aquest punt, els valors *d'incrementX* i *incrementZ* fan avançar a nivell de cel·la, fins a col·lisionar amb un bloc ocupat o fins sobrepassar els límits establerts del mapa. Tot seguit mostrem la seva implementació,

```
Const
       DIMENSIO_CELA = 64
       MAX_CELES_X = 13
       MAX_CELES_Z = 13
       MAX_UNITATS_MAPA_X =  MAX_CELES_X*UNITATS_AMPLADA_CEL·LA = 13*64 = 832
       MAX_UNITATS_MAPA_Z = MAX_CELES_X*UNITATS_AMPLADA_CEL·LA = 13*64 = 832
Fconst

 Funcio RaigZ(e Xj,Zj:enter;AngleRaig:real
              e/s Textura, ColumnaTextura:enter) retorna real
 var

  PosicioMapaX,PosicioMapaZ:enter
  PosicioRaigZ,Zi,IncrementZ,PosicioRaigX,Xi,IncrementX:real
 fvar


 Si (AngleRaig ≠ 180.0) i (AngleRaig ≠ 0.0) llavors { Angle vàlid }

    {Calculem el punt de partida del raig + els increments X i Z, explicats als apartats
             1.2.2.1 i 1.2.2.2}
    CalcularPassosRaigZ(Xj,Zj,AngleRaig,Xi,Zi,
                        IncrementX,IncremenZ);

    {Obtenim la posició actual del raig en la matriu 2D}

    PosicioMapaRaigZ(Xi,Zi,AngleRaig,PosicioMapaX,PosicioMapaZ);

    {El punt de partida del raig }

    PosicioRaigX ← Xi
    PosicioRaigZ ← Zi

    Si (Mapa[PosicioMapaZ][PosicioMapaX] = 0) i
       (PuntDinsMapa(PosicioRaigX,PosicioRaigZ) llavors { Es Traça el raig}

       repetir

        PosicioRaigX = PosicioRaigX + IncrementX
        PosicioRaigZ = PosicioRaigZ + IncrementZ
        PosicioMapaRaigZ(PosicioRaigX,PosicioRaigZ,AngleRaig,PosicioMapaX,PosicioMapaZ);

      fins que  (Mapa[PosicioMapaZ][PosicioMapaX] ≠ 0) i
                no(PuntDinsMapa(PosicioRaigX,PosicioRaigZ))
    FSi

    Si no (PuntDinsMapa(PosicioRaigX,PosicioRaigZ)llavors {Es Retorna la distància més gran}

       RaigZ ← DISTANCIA_MES_LLARGA

    Altrament { Es retorna la informació de paret : Textura, Columna i distància}
           Textura ← Mapa[PosicioMapaZ][PosicioMapaX]-1
           ColumnaTextura ← enter(PosicioRaigX) mod DIMENSIO_CELA
           RaigZ ← ArrelQuadrada((PosicioRaigX-Xj)^2 + (PosicioRaigZ-Zj)^2))
    Fsi
    Altrament { Error, per que els calculs de x/tan(0) o x/tan(180) són infinits }
       { Es retorna la distància més llarga }
              RaigZ ←DISTANCIA_MES_LLARGA;
    Fsi
Ffunció
```

Mostrem la funció que calcula l'offset inicial i passos d'increment de raig Z, segons l'orientació de raig,

```
{Calcul del punt de partida del raig(Xi,Zi) i dels increments X i Z (Δx,Δz)}
Accio CalcularPassosRaigZ(e Xj,Zj:enter;AngleRaig:real
                          e/s Xi,Zi,IncrementX,IncrementZ:real)
var
  ComencamentZ :enter
fvar

ComencamentZ ← (Zj div DIMENSIO_CELA)*DIMENSIO_CELA
IncrementZ   ← DIMENSIO_CELA

si(AngleRaig < 180.0) i (AngleRaig > 0.0) llavors {El raig s'orienta al sud }

        Zi ← ComencamentZ + DIMENSIO_CELA

        Si AngleRaig = 90.0 llavors { IncrementX = 0 i Xi és la posició Xj.}

          Xi ← Xj
          IncrementX ← 0

        Altrament { s'aplica càlculs fets a la secció 1.6.2.2.1 i 1.6.2.2.2  esbrinar Xi,Δx}

          Xi ← (Zi - Zj) div tan(AngleRaig) + Xj
          IncrementX ← IncrementZ div tan(AngleRaig)

        Fsi

   altrament { el raig s'orienta al nord → es parteix de la malla on es troba el jugador}

        Zi ← ComencamentZ
        IncrementZ ← - IncrementZ

        Si AngleRaig = 270.0 llavors { IncrementX = 0 i Xi és la posició Xj}

          Xi ← Xj
          IncrementX ← 0

        altrament { s'aplica càlculs fets a la secció 1.6.2.2.1 i 1.6.2.2.2 per esbrinar Xi,Δx}

           Xi ← (Zi - Zj) div tan(AngleRaig) + Xj
          IncrementX ← IncrementZ div tan(AngleRaig)
        Fsi
    Fsi
Ffuncio
```

Mostrem el pseudocdi que retorna la posició inicial de cel·la en funció l'orientació de raig i les coordenades de jugador,

```
accio PosicioMapaRaigZ(e Xj,Zj:enter;AngleRaig:real;e/s PosMapaX,PosMapaY:enter)

        si AngleRaig < 180.0 { Raig orientat cap al sud...}

              PosMapaX ← (Xj div DIMENSIO_CELA) + 1
              PosMapaZ ← (Zj div DIMENSIO_CELA)+ 1

        altrament { Raig orientat cap al nord...}

              PosMapaX ← (Xj div DIMENSIO_CELA) +1
              PosMapaZ ← (Zj div DIMENSIO_CELA)

        Fsi
faccio
```

1.6.3 Etapa del renderitzat de tira

En la secció anterior s'ha explicat la manera de traçar virtualment un raig, fins que aquest, col·lisiona amb una paret. Llavors, és retorna tres informacions: **la distància des del jugador a la intersecció de paret, la component u i el identificador de textura.** Amb aquesta informació es podrà renderitzar la tira de paret.

La renderització de tira consta d'una projecció de tira a pantalla més un pintat de textura. El pintat consta d'un pintat vertical a l'espai de textura i vídeo. Això es degut a que la distància és constant durant tot el pintat de la tira, la qual cosa diu que el traçat de textura a pantalla serà un recorregut amb *u* fix i un coeficient de variació constant en *v*.
Això explica que la renderització de tira de paret, només requereix escalar columnes de la textura al pla projecció (pantalla).
Com es pot observar a la figura 1.30, **la paret** representada a pantalla, no **és** res més que **una col·lecció de tires de textura *escalades* en funció de la distància respecte jugador**, que les representa.

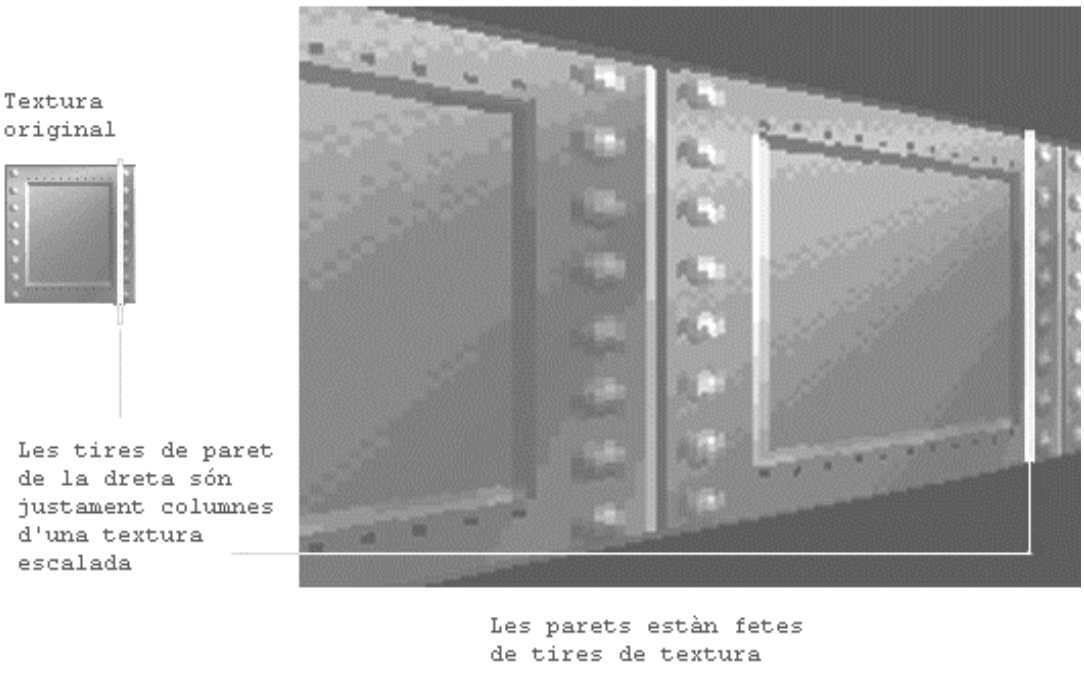
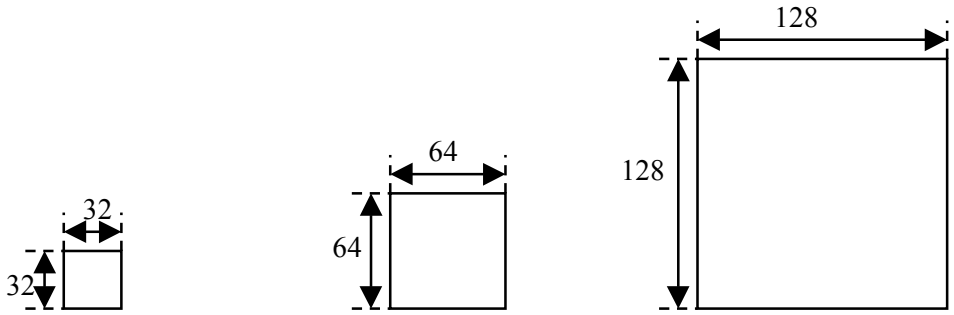


figura 1.30

El terme de correcció d'escalat és un factor numèric que explica els pixels a descartar o repetir en una iteració. Més concretament, aquest factor de correcció és una **transformació de gruix** d'un objecte a un gruix més gran o més petit respecte l'original. Cada tira de paret representa una columna de textura escalada en funció de la distància. Si la distància és gran és natural que el factor de correcció d'escalat sigui gran, per descartar més pixels i per conseqüència representar l'objecte llunyà. En canvi, si la distància és petita, és natural que el factor de correcció d'escalat sigui petit per repetir més pixels i per conseqüència representar l'objecte a prop.



correcció d'escalat 1:2 (doble de petit)	correcció d'escalat 1:1 (natural)	correcció d'escalat 2:1 (doble de gran)

figura 1.31 *Diferents representacions de gruix segons el seu escalat en vers el natural.*

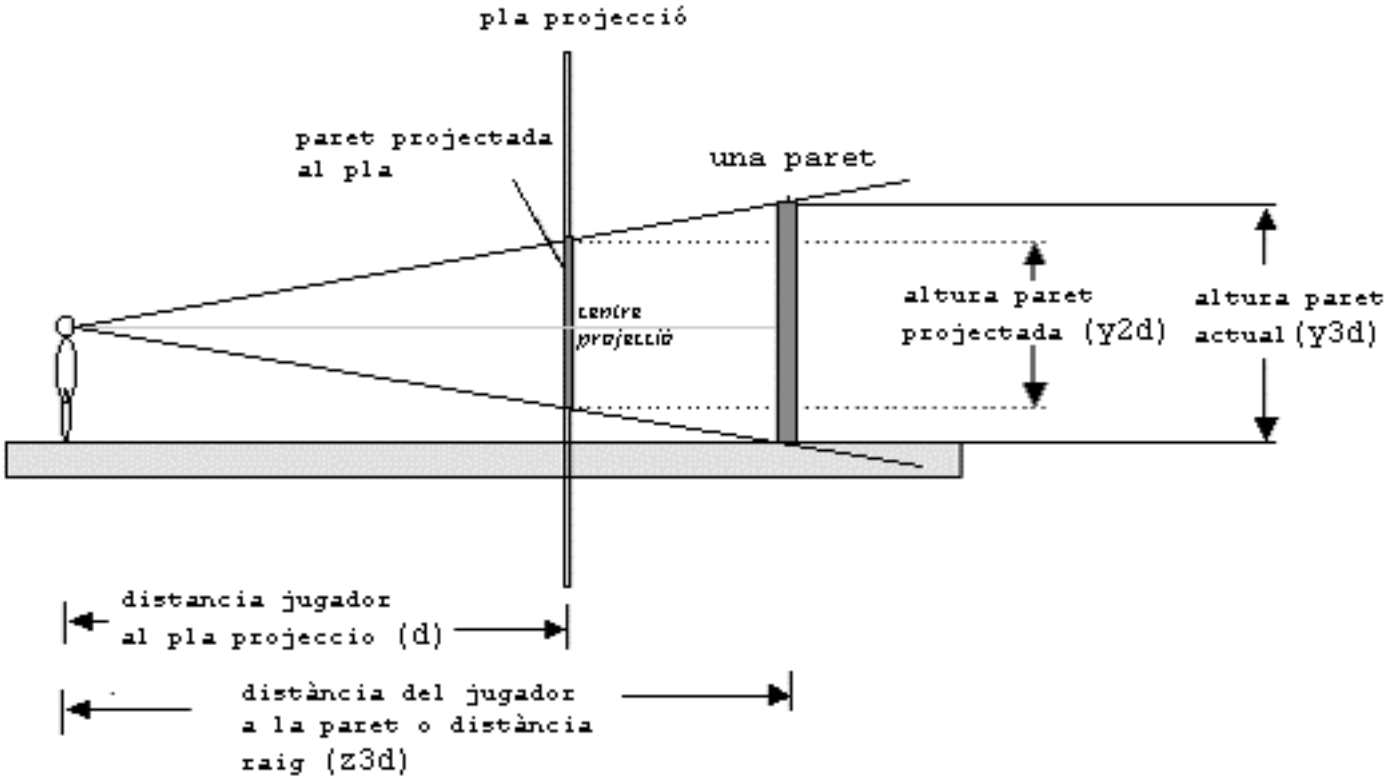
A la figura de la dreta de sobre la correcció d'escalat 1:2 descartaria 2 pixels en cada iteració representaria l'objecte llunyà i la figura de l'esquerra la correcció d'escalat 2:1 repetiria 2 pixels a cada iteració i per tant representaria l'objecte a prop.

I així és com tractem **les tires de paret**. En resum:

CADA TIRA ES TRACTA COM SI FOS **UN OBJECTE** A L'ESPAI 3D, QUE **ES FARÀ MÉS GRAN O MÉS PETITA EN FUNCIÓ DE LA DISTÀNCIA,** RESPECTE EL JUGADOR.

1.6.3.1 Càlculs previs

Es necessari fer càlculs abans de renderitzar la tira de paret al bùfer de treball. Principalment, s'ha d'obtenir el valor numèric de la correcció d'escalat comentat a la secció anterior, però abans, cal saber l'altura de la projecció tira de paret. Tot fent referència a la següent figura,



- **Càlcul de l'altura de paret projectada (y2d)**

Observant la *figura 1.31* i igualant per similaritat de triangles,

$$\frac{y2d}{d} = \frac{y3d}{z3d} \qquad \textit{equació 1.9}$$

La incògnita que interessa és *y2d*, ja que diu la relació d'altura de la paret a pantalla,

$$y2d = \frac{y3d}{z3d} \cdot d \qquad \textit{equació 1.10}$$

La *y3d* és coneguda i constant ja que és precisament la altura del cub (64 unitats), com s'havia establert a les definicions del nostre món. La *z3d* també és coneguda, ja que representa la distància que s'ha calculat anteriorment per alguna de les dues funcions que tracen el raig (*raigX*/*raigZ*).

Per obtenir *d* (distància des de jugador al pla projecció) s'estudiarà la figura 1.32.

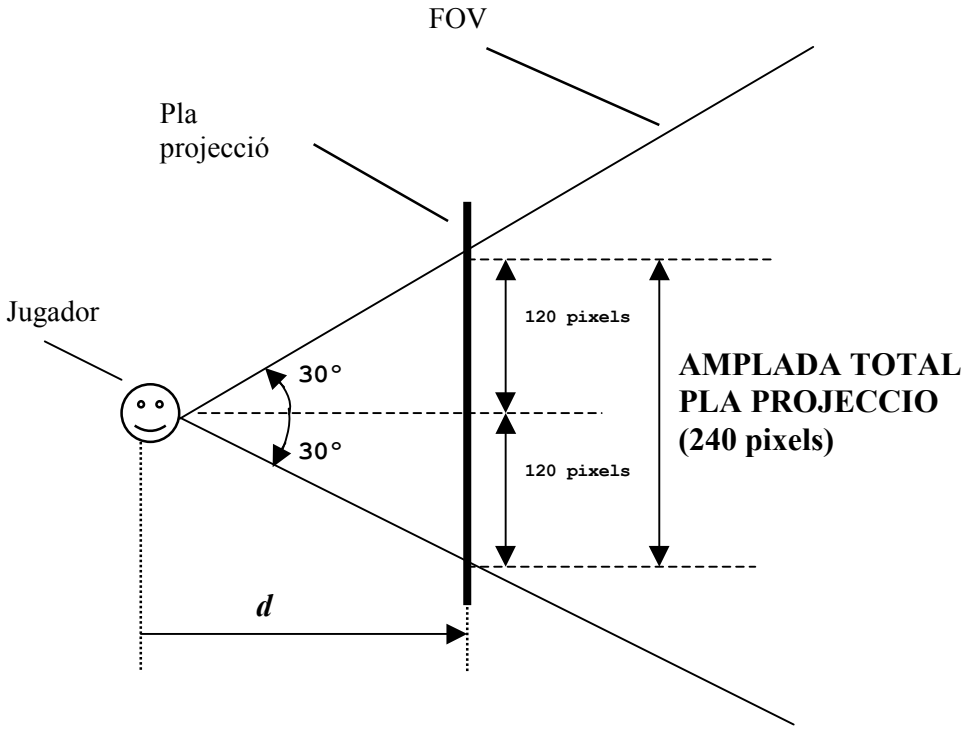


figura 1.32

Aplicant trigonometria, la constant *d* s'aconsegueix com,

$$tan(30º) = \frac{120}{d} \Rightarrow d = \frac{120}{tan(30º)} = 207{,}8460 \qquad \textit{Equació 1.11}$$

Coneixent les variables *y3d* i *d*, es substitueixen i tenim el numerador constant.

$$y2d = \frac{y3d}{z3d} \cdot d = \frac{64}{z3d} \cdot 207{,}8460 = \frac{13302{,}1502}{z3d} \qquad \textit{Equació 1.12}$$

Ja tenim l'equació per saber y2d, però cal corregir-la en cas de rebassar els límits de la màxima alçada de pantalla. Per tant,

```
si y2d > 160 llavors {Corregim per que s'ha passat del màxim}
     y2d← 160
fsi
```

I ja hem trobat l'altura de la tira.

- ## Càlcul de l'escalat (Escalat_v)

Un cop es sap l'altura de paret que representa al pla projecció, és determina l'escalat, simplement trobant quina proporció d'alçada representa la altura de la textura original al de la altura representada al pla,

$$Escalat\_v = \frac{Alçada\_textura}{y2d} = \frac{64}{y2d} \qquad \textit{Equació 1.13}$$

Es pot comprovar que,

- Si $y2d < 64 \rightarrow Escalat\_v > 1$, per conseqüent aquest coeficient descartarà pixels a l'espai de textura, i per això, és representarà la tira de forma llunyana al jugador.

- Si $y2d > 64 \rightarrow Escalat\_v < 1$, per conseqüent aquest coeficient repetirà pixels a l'espai de textura, i per això, és representarà la tira de forma propera al jugador.

- **Càlcul de la fila inicial de pantalla (coordenada y1)**

Un cop determinat el factor escalat (*Escalat_v*) cal saber a quina **fila de pantalla** es començarà a pinta la tira de paret. Si s'observa la *figura 1.31*, veiem que el centre de projecció coincideix a mitja alçada de la paret (alçada del jugador). Si la GBA té 160 files, el centre de projecció (*centre_y2d*) es troba a:

$$centre\_y2d = \frac{160}{2} = 80$$

Es sap que el centre de projecció està a la fila 80, ara cal restar la meitat de l'altura de tira de paret projectada, per possessionar-se al inici del pintat del pla projecció.

$$y1 = centre\_y2d - \frac{y2d}{2} = 80 - \frac{y2d}{2} \qquad \textit{Equació 1.14}$$

- **Càlcul de l'inici de pintat textura (coordenada v1)**

També s'ha de determinar quina fila es començarà a tractar la textura. La *v* inicial la trobarem en funció de la *y1* trobada anteriorment.

Observant la *figura 1.31*, veiem que el centre de projecció (*centre y3d*) és igual a l'altura del jugador, o sigui, 32 unitats respecte el terra.

$$centre\_y3d = 32$$

Si sabem que el centre de projecció és 32 i sabem la meitat de la paret projectada, sabrem la coordenada v1.

$$\texttt{v1 = centre\_y3d} - \frac{\texttt{y2d}}{2} \cdot \texttt{Escalat\_v} \Rightarrow \boxed{\texttt{v1 = 32} - \frac{\texttt{y2d}}{2} \cdot \texttt{Escalat\_v}} \qquad \textit{Equació 1.15}$$

1.6.3.2 El bucle general del renderitzat de la tira

Un cop sabem l'inici de les coordenades *y1*, *v1* i la correcció d'escalat *Escalat_v* l'últim pas és el renderitzat de tira al bùfer de treball. Fent servir la funció *ham_PutPixel()*,

estudiada a la secció 4.2 del capítol 4, el bucle del renderitzat de tira el compondria el pseudocodi següent.

```
si y2d > 0 llavors
   {bucle general del renderitzat de la tira de paret}
   repetir

      Color = ptr_textura[(ceil(v1*64) mod 64) + u]

      ham_PutPixel(Columna,y1,Color)

      y1 ← y1 + 1
      v1 ← v1 + Escalat_v
      y2d = y2d - 1

   Fins que y2d = 0
fsi
```

Ptr_textura, representa el punter a la imatge que fa referència a la textura retornada pel raig, la qual mapeja la tira de paret. Cal observar que s'obté del texel de la posició (u, v1) a partir del l'offset lineal equivalent (per més detall sobre l'offset lineal consultar secció 4.3.2 del capítol 4).

El **"mod 64"** es fa servir per assegurar un valor de v1 vàlid al rang de textura, llavors $v1 \in [0,..,63]$.

1.6.3.2 Exemple

Si algun dels raigs col·lisiona amb una paret a la **columna gràfica 150** a una distància de raig de **330 unitats,** aplicant l'*equació 1.12*, l'altura de paret projectada seria,

$$y2d = \frac{13302{,}1502}{z3d} + 0{,}5 = \frac{13302{,}1502}{330} + 0.5 = 40{,}7095 \sim 40 \ pixels$$

La meitat de la tira esta a la fila 80 del pla projecció. Llavors aplicant l'*equació 1.14* la coordenada *y1* seria,

$$y1 = centre\_y2d - \frac{y2d}{2} = 80 - \frac{40}{2} = 60$$

La figura 1.33 mostra com quedaria la tira de paret projectada a pantalla.

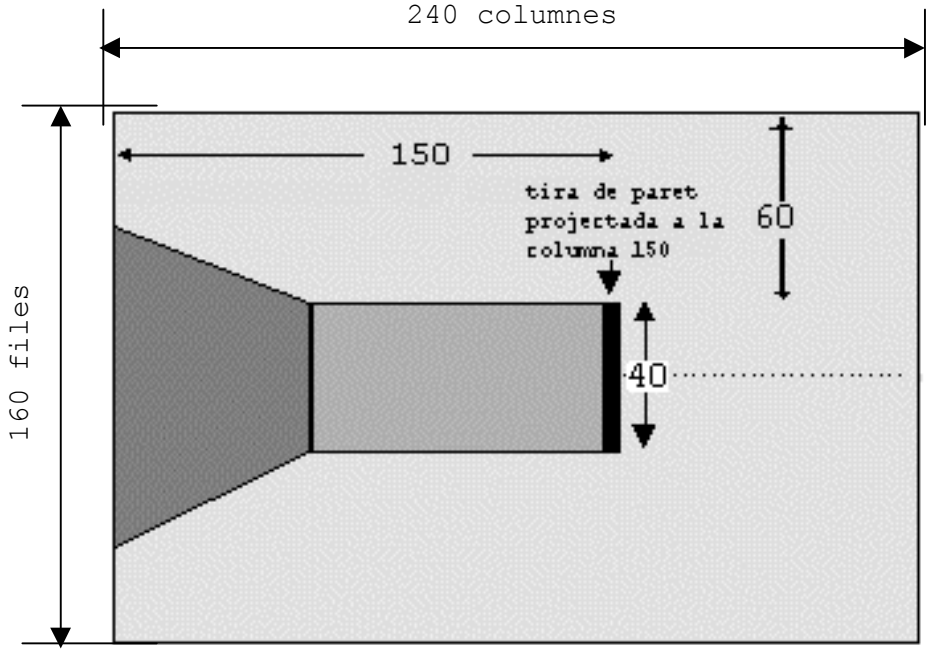


figura 1.33 *Exemple de tira de paret projectada.*

Pel pintat, trobem l'escalat fent servir *l'equació 1.13*,

$$Escalat\_v = \frac{Alçada\_textura}{y2d} = \frac{64}{40} = 1{,}6$$

I segons aquest escalat, es troba la coordenada *v1* amb *l'equació 1.15*,

$$v1 = centre\_y3d - \frac{y2d}{2} \cdot Escalat\_v = 32 - \frac{40}{2} \cdot 1{,}6 = 32 - 32 = 0$$

1.6.3.3 Implementació del renderitzat de tira de paret

Amb el hem explicat en aquesta secció, presentem la implementació necessària per pintar (o mapejar) una tira vertical de paret,

```
accio PintarTiraParet(entrada  ptr_textura:^byte;
                               z3d:real;u,columna:enter)
{
    Ptr_textura: Punter de bytes a la textura.
    u : Columna de textura el qual esta associada al pintat de la tira.
    Columna: Columna de pantalla on és pintarà la tira.
    z3d : Distància des de jugador a paret.

 }
 var
  y1,y2:enter;
  v1,Escalat_v: real;
 fvar

 {Es calcula l'altura de tira de paret projectada}

 y2d ← 13302,1502 div z3d

 si (y2d > 160) llavors {Corregim, ja que no podem pintar més de 160 files}

    y2d←160

 fsi

 {Es calcula l'escalat, segons l'altura de textura...}

 Escalat_v ← 64 div y2d

 {Es calcula l'inici fila de video..}

 y1 ← 80 - (y2d div 2)

 {Es calcula l'offset inici fila de textura..}

 v1 ← 32 - (y2d div 2)*Escalat_v

 si y2d > 0 llavors
 {bucle general del renderitzat de la tira de paret}

    repetir

        Color = ptr_textura[(ceil(v1*64) mod 64) + u]

        ham_PutPixel(Columna,y1,Color)

        y1 ← y1 + 1
        v1 ← v1 + Escalat_v
        y2d = y2d - 1

    Fins que y2d = 0
 fsi

faccio
```

1.6.4 Distorsió "ull de peix"

Cal explicar l'existència d'un problema amb aquesta implementació del ray-casting: el càlcul de les distàncies de raig (*equació 1.5)* provoca una distorsió anomenada "ull de peix". Aquesta distorsió passa perquè la implementació ray-casting projecta els raigs des de la posició jugador a les respectives interseccions de paret.

Exemple,

Imaginem que el jugador es troba direccionat als 270º on el seu *FOV* engloba tota una paret totalment recte al seu davant,

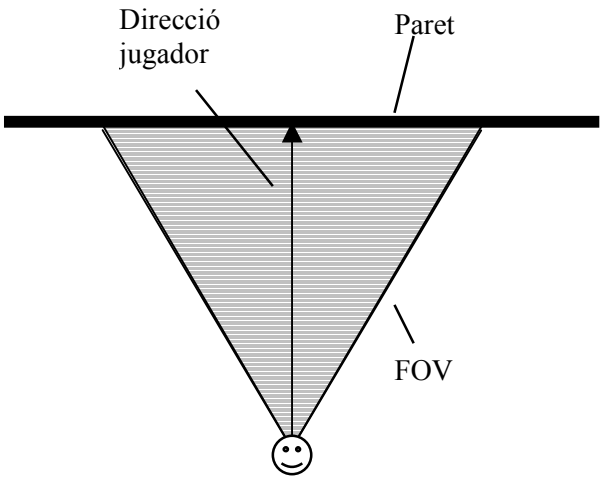


Figura 1.34 *Jugador amb direcció perpendicular davant d'una paret.*

Si executéssim la renderització seguint els procediments implementats fins ara, la paret renderitzada es veuria distorsionada com la que tenim a la figura 1.35.

Figura 1.35 *Distorsió "ull de peix"*

El fet principal d'aquesta distorsió es que, calculem la **distància 3d. La distància 3D provoca que als extrems tinguem una distància des de el jugador més llunyana que la que dona al centre** i per això les tires de paret són més curtes respecte el centre, com podem observar a la figura 1.35. Aquesta distorsió provoca una deformació al nostre ull i a l'algorisme.

Realment, nosaltres els humans, captem les imatges amb aquesta distorsió, però el cervell les corregeix. Nosaltres també la corregirem trobant **la distància a l'eix z** de cada raig, així totes les distàncies des del jugador a les interseccions de raig-paret seran les mateixes i per conseqüent arreglarem aquesta distorsió. La figura 1.36 mostra un exemple de distància en z pel primer raig llençat,

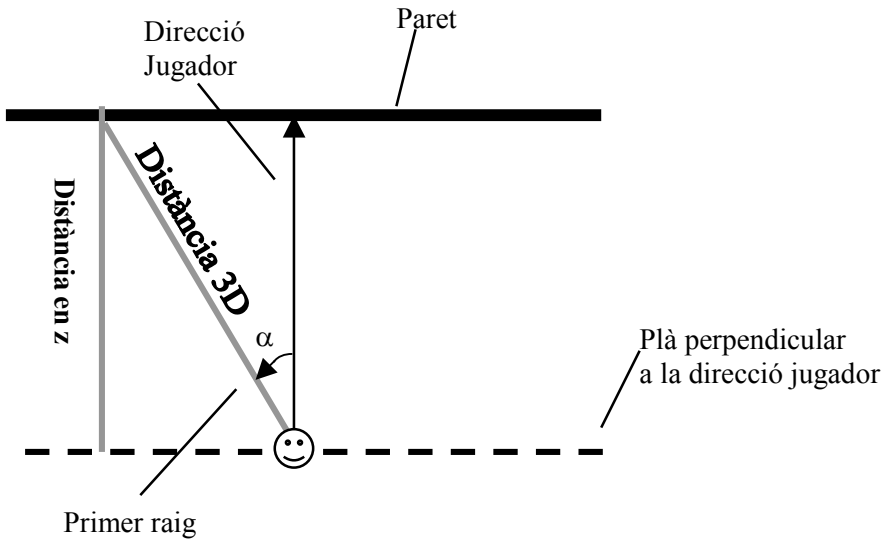
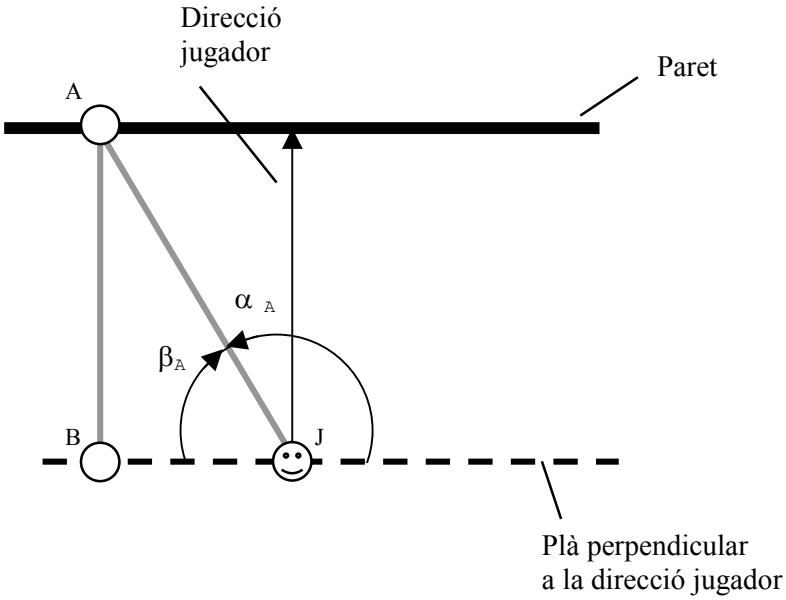


Figura 1.36 *Figura on mostra la Distància 3D trobada (distància amb distorsió) i la Distància en z (distància correcte).*

Les distàncies correctes (Distància en z) són les que es troben entre el plà perpendicular a la direcció que mira el jugador a les interseccions de respectius raigs amb paret.

Tot seguit expliquem com corregir la distorsió, seguint la figura 1.37.



α_A = L'angle de raig projectat a l'esquerra de l'angle direcció.
β_A = L'angle que complementa a α_A.

figura 1.37

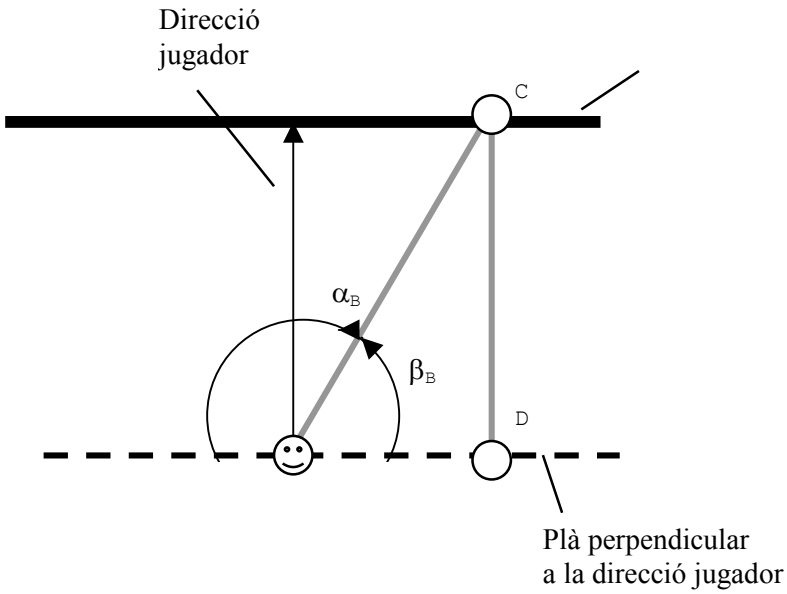
β_A s'aconsegueix com segueix,

$\beta_A \leftarrow$ `90° - ` α_A

i aconseguim la distància en z aplicant trigonometria,

`dist(J,A) ←  dist(J,A) * SIN(`β_A`)`

Per trobar les distàncies en z si α_A és projecta a la dreta de l'angle direcció, s'ha de buscar una nova una nova β. De manera similar a la figura 1.37, a la figura següent tenim un angle de raig projectat a la dreta del angle direcció, concretament l'últim raig (+30º a la direcció de jugador).



α_B = L'angle de raig projectat, a la dreta de la direcció de jugador.
β_B = L'angle que complementa a α_B.

Figura 1.38

en aquest cas beta serà (recordant que considerem raigs orientats)

$\beta_B \leftarrow \alpha_B - 90$

i com abans aconseguim la distància en z,

`dist(J,C)` ← `dist(J,C) * SIN(`β_B`)`

Cal mencionar que els sinus calculats es troben **sempre respecte la l'angle de direcció actual del jugador**. Sabem que la direcció del jugador sempre apunta a la meitat de la pantalla. Podem emplear la següent condició a l'hora de corregir la distància entregada per alguna de les funcions *raigX* o *raigZ* a la *columna* actual:

```
const
   FOV = 60.0
   INCREMENT_FOV = FOV/AMPLADA_PANTALLA = 60/240 {0.250 °/columna}
fconst

---

Beta ← 90  -  (30.0 - columna*INCREMENT_FOV )

{Corregim distància}

DistanciaCorrecte←Distancia*sin(Beta)
```

Tanmateix cal observar que el resultat de la operació sinus sempre serà el mateix a la columna actual. Per tant podem precalcular els seus valors en una taula de 240 posicions (240 columnes), on la construcció d'aquesta és veu en el següent pseudocodi,

```
per columna desde 0 a 239 fer

   Beta ← 90  -  (30.0 - columna*INCREMENT_FOV )

   CorrecioEfectePeix[columna] ← sin(Beta)

fper
```

Per corregir la distància només caldrà fer una multiplicació al valor sinus que farà referència la columna actual que estiguem disposats a renderitzar.

```
Distancia ← Distancia3D*CorrecioEfectePeix[ColumnaActual]
```

1.6.5 Intent de ampliar el FOV a 90º

Hem comentat a la secció 1.1.3 que molts motors de ray-casting utilitzen un FOV de 60º, per això, també hem fet us d'aquest angle en aquest capítol. Al fer la renderització ray-casting amb un FOV de 90º, vam poder observar que, efectivament, existia una distorsió a les imatges renderitzades: en algunes situacions del jugador es podía observar una distorsió sinoïdal als extrems inferior/superior de paret. (figura 1.39).

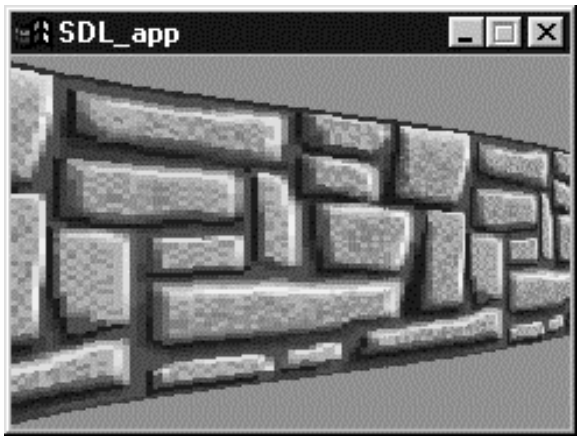


Figura 1.39 *Imatge renderitzada amb distorsió sinoïdal al extrem inferior/superior*

En aquesta secció demostrarem que aquesta distorsió es causa per els raigs que no intersequen per la columna on s'han llançat.

Ja haviem estudiat que el raig es llança per cada columna en un angle múltiple de la següent constant:

$d\alpha \leftarrow$ Amplada_Pantalla/FOV

Per això, el que l'angle de llançament de raig $\in[\alpha_i, \alpha_i + d\alpha, \alpha_i + 2 \cdot d\alpha, ..., \alpha_i + n \cdot d\alpha]$, on $\alpha_i = \alpha_j - FOV/2$.

Doncs, aquest angle de llançament de raig, **matemàticament, no inteserca la columna** pertinent. Analitzem els càlculs de l'angle de raig llançat per la columna 180, a la següent figura:

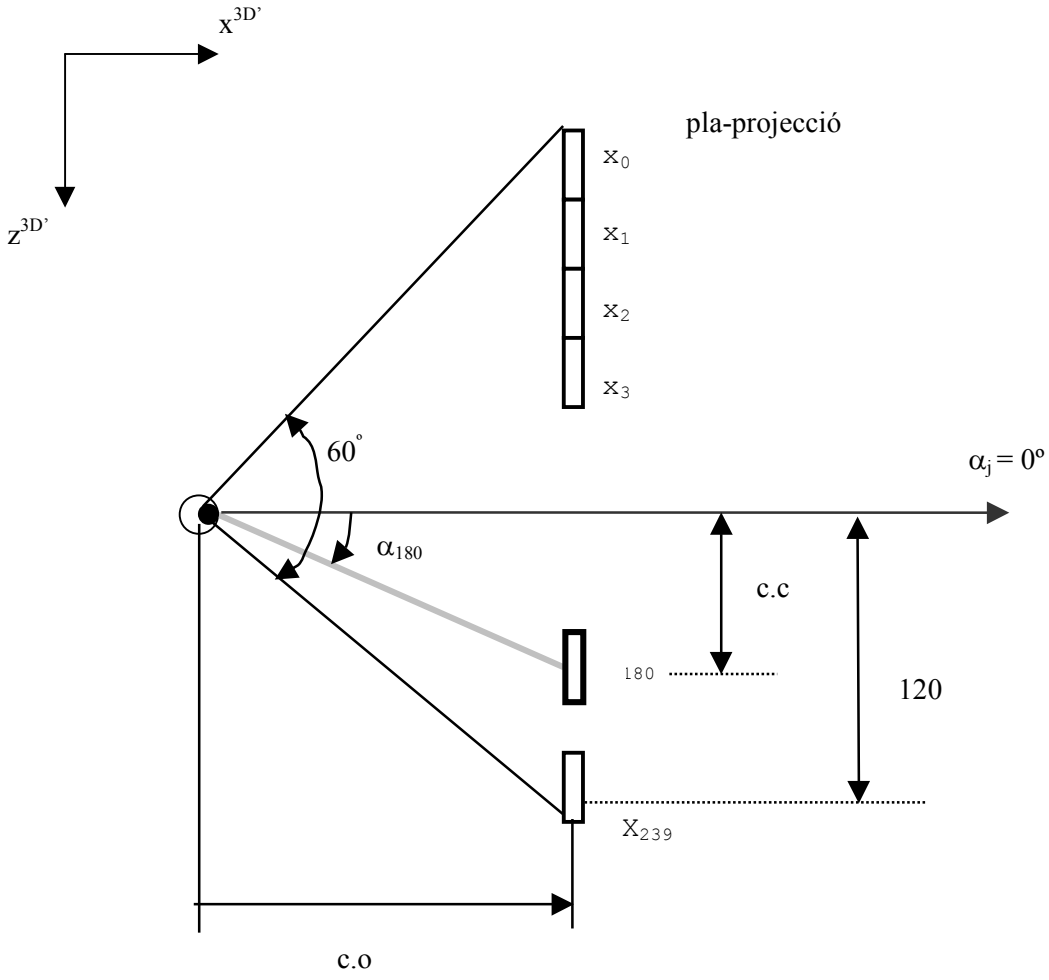


Figura 1.40

Segons la figura 1.40, volem demostrar que el raig llançat per la columna 180, hauria de tenir el la mateixa equivalència segons la formula trigonomètrica següent,

$$tan(\alpha_{180}) = \frac{c.o}{c.c}$$

Aplicant valors de la figura 1.40 obtenim l'equivalència,

$$tan(\alpha_{J} - \frac{60}{2} + 0.25 \cdot 180) = \frac{180 - 120}{\frac{\frac{240}{2}}{tan(\frac{60}{2})}}$$

$$tan(0 - 30 + 45) = \frac{40}{207,84}$$

$$0,2679 \neq 0,288$$

Es a dir, que s'ha demostrat que l'angle de raig que utilitzàvem fins ara per llançar-los no és correcta, però que per un FOV de 60º desprecia aquest error. Així doncs, si volguéssim fer servir un FOV de 90º caldrà corregir aquest error per la manera correcta de llançar els raigs, es a dir, mitjançant la formula trigonomètrica c.o/c.c.

1.6.6 La funció de render

Finalment, mostrem la implementació de la funció principal de renderització,

```
Accio Renderitzar(e Jugador:tJugador)
  var
       AngleRaig, DistanciaRaigX, DistanciaRaigZ, Distancia:real
       ColumnaActual, TexturaX, TexturaZ, uX, uZ: enter
  fvar

  {1.- Basat en el FOV que tenim,  es resta 30° a la direcció del jugador }

  AngleRaig ← Jugador.AngleDireccio – 30.0

   Si AngleRaig < 0 llavors { Estem per sota els 360°, cal corregir}
       AngleRaig ← AngleRaig + 360.0
   FSi

   Per ColumnaActual desde 0 fins a 239 pas +1 fer

       {2.- Etapa geomètrica → Llencem el raig separat en dues parts: RaigX i RaigZ }

       DistanciaRaigX ← RaigX(Jugador.X, Jugador.Z, AngleRaig,TexturaX,uX)
       DistanciaRaigZ ← RaigZ(Jugador.X, Jugador.Z, AngleRaig,TexturaZ,uZ)

       { Pintarem la tira de textura de la distància de paret més propera al jugador }

       Si DistanciaRaigX < DistanciaRaigZ llavors {Pintem la tira del raig X}

              {Corregim distorsió efecte peix}
              Distancia ← DistanciaRaigX*CorreccioEfectePeix[ColumnaActual]

              {3.- Renderitzem tira de paret segons paràmetres del RaigX}
              PintarTira(Textura[TexturaX],Distancia,uX,ColumnaActual)

       Altrament {Pintem la tira del raig Z }

              {Corregim distorsió efecte peix}
              Distancia←DistanciaRaigZ*CorreccioEfectePeix[ColumnaActual]

              {3.- Renderitzem tira de paret segons paràmetres del RaigZ}
              PintarTira(Textura[TexturaZ],Distancia,uZ,ColumnaActual)

       FSi

       AngleRaig ← AngleRaig + IncrementAngle {incorrecte, però visualment acceptable
   per un FOV de 60°}

       Si AngleRaig > 360.0 llavors { Si estem per sobre els 360 °,cal corregir  }
            AngleRaig ← AngleRaig – 360
       FSi

   FPer
```