Visualització en temps

crowd
GENERATION

Especialitat: ETIG pla 2001

Centre: EPS, UdG

Director/ tutor: Gustavo Patow

Convocatòria: Juny 2008

Departament: IMA

Àrea: LSI

Centre: EPS, UdG

EPS Escola Politècnica Superior
UdG

Projecte/Treball Final de Carrera

Estudi: Eng.Tècn. Informàtica de Gestió. Pla 2001

Títol: Visualització en temps real de multituds

crowd
GENERATION

Document: Memòria

Alumne: Sergi Valls Vilà

Director/Tutor: Gustavo Patow

Departament: Informàtica i Matemàtica Aplicada

Àrea: LSI

Convocatòria (mes/any): Juny / 2008

Agraïments

Primer de tot, i abans que res, vull donar les gràcies als meus pares, per haver-me donat l'oportunitat d'haver estudiat aquesta carrera, i sobretot, per haver-me donat tot el que necessitava, i més, per arribar fins al final. Sense vosaltres, jo no seria avui aquí.

En segon lloc a la meva germana Laura, per estar sempre al meu costat, i al meu cunyat, Pere: gràcies per la teva gran experiència... i per la teva paciència!

A l'Isaac, Oriol, David, Ferran... i a tota la resta que heu passat pel meu costat tots aquests anys: gràcies per arrencar-me un somriure cada dia i aconseguir que no m'oblidi mai de vosaltres ni del meu pas per la UdG.

A Gustavo Patow, el meu tutor, per donar-me l'oportunitat de crear el GdM i per haver-me ajudat quan l'he necessitat!

I finalment a tu, Sara, per haver-me fet costat tot aquest temps, per haver-me animat quan més ho necessitava i per haver-me ensenyat a estimar. Que la fi d'aquest projecte signifiqui el començament del nostre!

A tots, MOLTÍSSIMES GRÀCIES!

Índex

Presentació del Projecte

Aquest document presenta el contingut del nostre Projecte Final de Carrera, l'objectiu del qual ha estat desenvolupar un sistema de visualització interactiu de multituds de persones. L'estructura que segueix el document és la següent:

1.- Introducció: en aquest apartat introduirem el tema de les generacions de multituds avui en dia, així com quin enfocament li hem donat al nostre projecte. Explicarem també la temporalització, definició i els objectius.

2.- Conceptes previs: explicarem les idees bàsiques que el lector ha de conèixer per a poder comprendre de manera satisfactòria tot el contingut de la memòria i així entendre el per què de les solucions adoptades per la resolució dels diferents aspectes del projecte. Principalment parlarem dels motors gràfics que hi ha al mercat i del programa de disseny gràfic que hem escollit pel desenvolupament del projecte.

3-a.- Generació de les multituds: és el capítol on explicarem com hem dissenyat el Generador de Multituds (en endavant GdM). Explicarem també l'apartat del disseny del personatge per recrear la multitud.

3-b.- Càlcul de col·lisions: aquí explicarem com es va idear i implementar aquest mòdul per donar més realisme al GdM. Tot i estar en un apartat diferent, l'explicarem dins el punt 3 degut a la seva estreta relació.

4.- Resultats: en aquest capítol farem un petit estudi dels resultats obtinguts del GdM.

5.- Conclusions: capítol dedicat a exposar les idees finals i a presentar les possibles millores que es poden aplicar al GdM.

Un cop situats passem a introduir el nostre GdM.

Capítol 1: introducció

Avui en dia, hi ha moltes pel·lícules o videojocs on apareixen grans quantitats de persones.

La visualització en temps real de multituds de persones ha estat, fins ara, un problema exclusiu de l'àmbit de l'animació pel cinema. Això és així perquè sovint és molt costós aconseguir grans quantitats de persones per realitzar les escenes, amb els problemes que això comporta (ja siguin econòmics o d'infraestructura) i la majoria de les vegades resulta inviable. Amb l'arribada de les noves targes gràfiques, és possible calcular aquesta visualització en temps real, donades una sèrie de simplificacions respecte de l'estructuració dels models a visualitzar.

El que es pretén amb el GdM no és oferir una visualització molt realista de multituds (com ho faria el M*assive Software*, per exemple), sinó que el que es vol és la simulació de multituds usant molt menys temps per obtenir uns resultats prou bons per a simular el comportament i el moviment d'una multitud humana.

En el GdM s'aconsegueix un bon rendiment a l'hora de generar les multituds gràcies a l'ús d'impostors, que més endavant explicarem, i l'ús d'estructures de dades específiques per al càlcul en temps real de les col·lisions.

Cal dir també que aquest projecte consta de dues parts ben diferenciades. La primera és la creació del model de persones que formaran la multitud. La segona és la creació del GdM i la integració d'aquest model per la generació de la multitud.

Així doncs, podem dir que amb la combinació d'aquestes dues tècniques, el GdM obté uns resultats prou bons en relació als recursos usats i el temps utilitzat.

1.1- Situació actual

Actualment, un dels programaris més coneguts per la generació de multituds (probablement per la seva participació en grans superproduccions d'Hollywood) és el *Massive Crowds*.

Massive Crowds és un programari d'animació 3D per la generació de multituds i d'efectes visuals per la televisió i la gran pantalla. Usant el *Massive Crowds* un dissenyador pot dissenyar personatges capaços de reaccionar als events que els envolten.

Les reaccions dels personatges determinen què fan i com ho fan. Aquestes reaccions poden ser també qualitats emotives, com per exemple valentia, tristesa o alegria.

Cada personatge amb un comportament concret s'anomena agent. *Massive Crpwds* és un sistema per dissenyar i animar agents. Quan aquests agents arriben a ser un número important (una multitud) la interacció entre ells dóna com a resultat una animació amb un realisme espectacular.

Si a més a més es van modificant les reaccions de cada agent, el dissenyador pot fer créixer una escena amb grans multituds on cada personatge sigui diferent en tot a qualsevol altre. Amb tot això s'aconsegueix una gran riquesa, com es pot comprovar en pel·lícules com la Trilogia del Senyor dels Anells (figura 1.1), I robot (figura 1.2), Happy feet (figura 1.3) o 300 (figura 1.4).

Fig 1.1- "El senyor dels anells

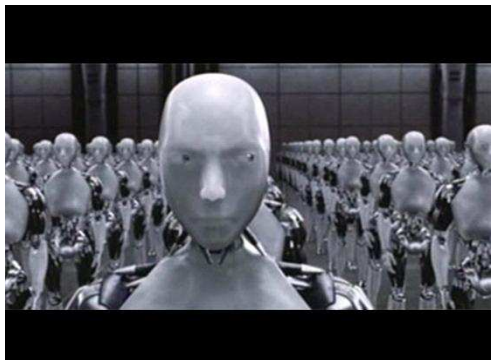


Fig 1.2- "I, robot"

Fig 1.3- "Happy feet"

Fig 1.4- "300"

1.2- Definició del GdM

El Projecte tracta sobre la implementació i disseny del GdM, una aplicació per la generació i visualització de multituds en temps real.

Primer de tot, cal haver dissenyat un personatge i animar-lo, tasca a realitzar per un dissenyador. Abans que el GdM s'executi, s'haurà d'haver creat l'atles de textura mitjançant les imatges obtingudes del moviment del personatge abans animat. A grans trets, podem dir que un atles de textura és una imatge formada per sub-imatges més petites. El motiu pel qual s'ha de crear és per generar els impostors (l'ús d'impostors significa que en comptes de visualitzar el model original es visualitzen imatges d'aquest model, estalviant recursos al no generar tota la geometria real del model), cosa que millorarà el rendiment. Aquest moviment del personatge s'obtindrà canviant les imatges cada cert temps, de forma que apareixent una després de l'altre s'observi un moviment continu del personatge. L'atles de textura està explicat més detalladament en el capítol 2, de conceptes previs.

Un cop ja estigui disponible l'atles, l'usuari haurà de triar el número de persones, el número i el tipus d'edificis que es volen generar, obtenint una multitud caminant per entre edificis amb un elevat grau de realisme.

El personatge usat pel GdM per generar la multitud s'haurà creat usant l'eina d'autor Maya i contindrà la informació del moviment del personatge que serà usat pel GdM per la creació de la multitud. Cal dir també que el procés de disseny del personatge recau sobre el dissenyador i és totalment invisible a l'usuari final.

Conceptualment són dos eines independents però finalment han de treballar conjuntament, com podem veure a les figures 1.5, 1.6 i 1.7.

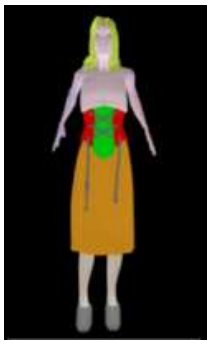


Fig 1.5- A partir d'un model d'entrada es genera el seu atles de textura

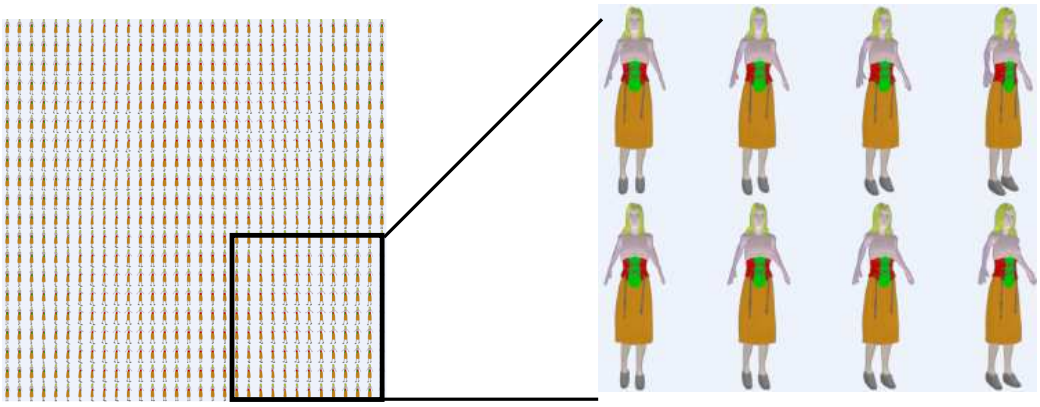


Fig 1.6- L'atles de textura del personatge

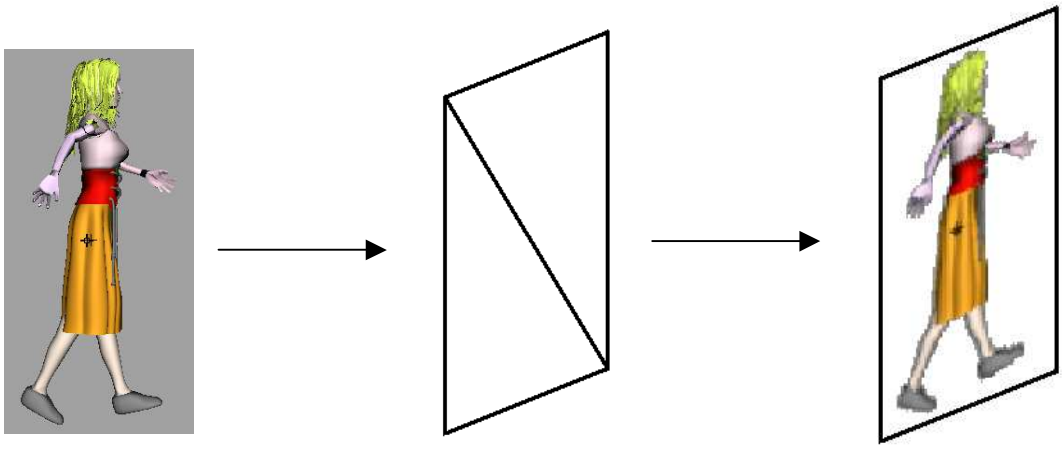


Fig 1.7- El procés de creació de l'impostor

Fig 1.8- Amb les dades entrades, el GdM genera la multitud

1.3- Etapes de creació del GdM

Pel disseny i implementació dels diferents mòduls que formen el GdM hem seguit les següents etapes:

- ✓ Animació del model a visualitzar
- ✓ Obtenció de les imatges d'aquest model per la realització de l'atles de textura, mitjançant el llenguatge de programació de Maya (*MEL*)
- ✓ Creació de l'atles mitjançant aquestes imatges.
- ✓ Disseny i implementació d'una tècnica de generació de multituds en C++, observant una bona qualitat d'imatge, moviments realistes i en temps real, amb bones quantitats de persones.
- ✓ Disseny de l'algoritme de detecció de col·lisions.
- ✓ Interacció amb l'usuari via interfície.

1.4- Temporalització del Projecte

El temps útil per la creació del GdM ha estat des dels inicis de Novembre de 2007 fins al Juny de 2008 (figura 1.9). Bàsicament, el Projecte s'ha separat en dues parts: la primera part està dividida en el disseny del personatge i el disseny del seu moviment:

a) Tria del programa de disseny gràfic adequat per la generació del personatge i el seu moviment.
b) Estudi dels manuals i documentació referent a aquest programa. Primeres pràctiques.
c) Creació del personatge i el seu moviment caminant.
d) Implementació de l'algoritme per la obtenció de les fotografies corresponents del seu moviment en cada instant de la seva animació.
e) Acoplament de l'algoritme generat dins el programa escollit.

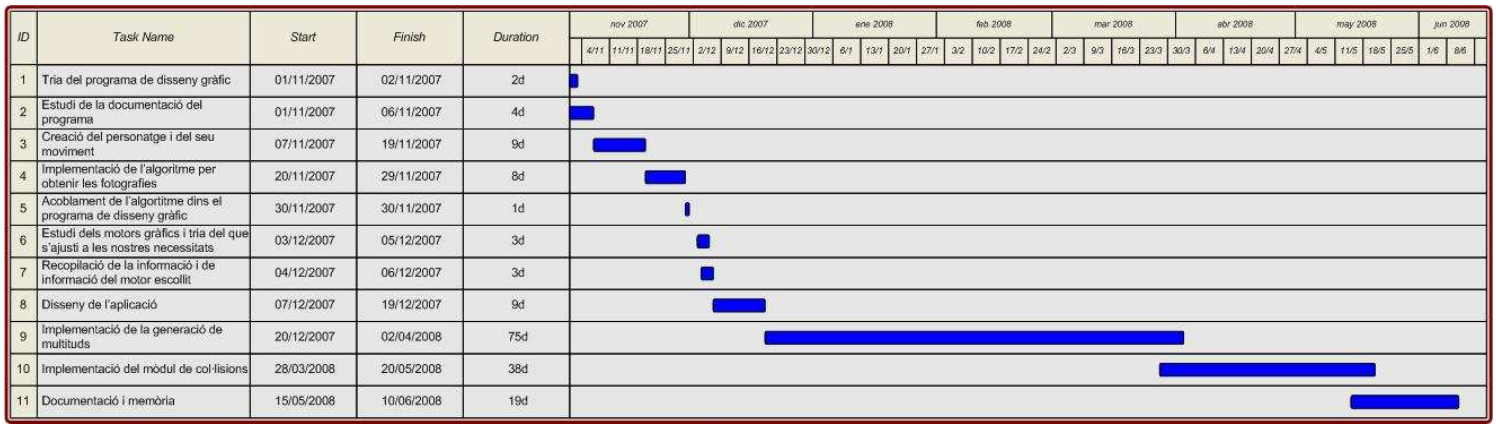
La segona part és la més extensa, ja que correspon al nucli principal del programa:

a) Estudi dels motors gràfics amb codi obert disponibles al mercat i tria del que més s'ajusti a les nostres necessitats.
b) Recopilació d'informació pràctica, així com tutorials i exemples del motor gràfic a usar, via llibres i majoritàriament via web.

c) Disseny de l'aplicació.

d) Implementació de la generació de multituds.

e) Implementació de la tècnica de detecció de col·lisions.



ID	Task Name	Start	Finish	Duration
1	Tria del programa de disseny gràfic	01/11/2007	02/11/2007	2d
2	Estudi de la documentació del programa	01/11/2007	06/11/2007	4d
3	Creació del personatge i del seu moviment	07/11/2007	19/11/2007	9d
4	Implementació de l'algoritme per obtenir les fotografies	20/11/2007	29/11/2007	8d
5	Acoblament de l'algortitme dins el programa de disseny gràfic	30/11/2007	30/11/2007	1d
6	Estudi dels motors gràfics i tria del que s'ajusti a les nostres necessitats	03/12/2007	05/12/2007	3d
7	Recopilació de la informació i de informació del motor escollit	04/12/2007	06/12/2007	3d
8	Disseny de l'aplicació	07/12/2007	19/12/2007	9d
9	Implementació de la generació de multituds	20/12/2007	02/04/2008	75d
10	Implementació del mòdul de col·lisions	28/03/2008	20/05/2008	38d
11	Documentació i memòria	15/05/2008	10/06/2008	19d

Fig 1.9- Temporalització del projecte

1.5- Metodologia

Com que el GdM s'ha implementat en un llenguatge orientat a objectes (C++) i en MEL, hem optat per presentar els diagrames de casos d'ús, diagrames de classe, diagrames de seqüència i diagrames d'activitat en l'estàndard del llenguatge de modelatge unificat (*Unified Modeling Language*, UML).

Capítol 2: conceptes previs

Per la realització del projecte hem hagut de tenir en compte certs coneixements per tal d'assolir els objectius marcats.

Abans de res, però, explicarem amb més detall el concepte de l'atles de textures.

2.1- Atles de textures

Avui en dia, un problema molt important en el camp dels gràfics per ordinador és el gran cost de visualitzar milers d'objectes diferents, on cadascun d'ells pot usar centenars de textures diferents. En aquest cas, el fet de canviar les textures d'aquests objectes per unes altres sense que disminueixi el rendiment del joc és un dels objectius a assolir. És precisament aquí on apareix la idea d'atles de textures.

En el camp dels gràfics per ordinador, un atles de textura és una gran imatge, o textura, que conté diverses sub-textures (o imatges) que no tenen perquè tenir relació entre elles. Aquestes textures ocupen una porció d'aquesta gran textura (veure figura 2.1). Alguns jocs, per exemple, guarden en aquests atles totes les textures que necessiten pelss seus escenaris i personatges. A mesura que es van necessitant, aquestes imatges poden ser usades simplement canviant les coordenades de textura de l'atles, obtenint una regió concreta d'aquest. Bàsicament la idea de l'atles es fa servir quan es pretén millorar l'eficiència de certes tasques. Per exemple, en una aplicació on es fan servir de forma freqüent textures petites, sovint és més eficient guardar en un atles totes aquestes imatges i llavors carregar a memòria aquesta textura més gran que anar carregant aquestes imatges més petites a mesura que es van necessitant.

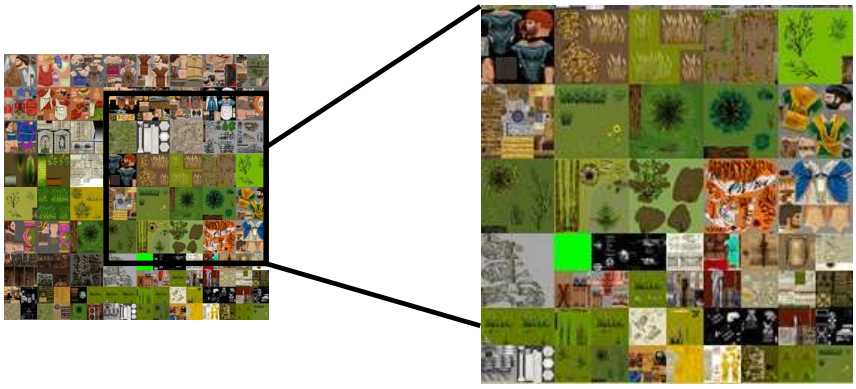


Fig 2.1- Un possible atles de textures

De forma general els atles poden contenir subtextures de mida uniforme o de mida variable. En el nostre cas serà de la primera forma.

2.2- Introducció al Maya

Abans de començar a explicar les funcionalitats que ens ofereix Maya, hem de dir que, com a alternativa d'ús, teniem el 3dStudioMax. 3dStudioMax és també de la companyia Autodesk, i, pràcticament, ofereixen els mateixos resultats. El motiu principal pel qual ens vam decantar pel Maya és que ja tenteníems coneixements bàsics de modelatge i d'animació amb aquest software. A més, ja posseíem aquest programa, per la qual cosa no vam haver de pagar-lo.

Maya és un programa informàtic dedicat al desenvolupament de gràfics en 3D, efectes especials i animació. Va sorgir de l'evolució de *Power Animator* i de la fusió de la companyia *Alias* i *Wavefront*, dos empreses canadenques dedicades als gràfics generats per ordinador. Actualment el programa pertany a *Autodesk*, conegut principalment pel seu programa *AutoCad*.

Maya es caracteritza per la seva potència i les possibilitats d'expansió i personalització de la seva interfície i eines. MEL (*Maya Embedded Language*) és el codi amb el qual estan fets els *scripts* que formen el nucli de Maya, gràcies al qual es poden crear *scripts* i personalitzar el paquet. És gràcies a aquesta funcionalitat que podrem crear el nostre generador d'atles i inserir-lo com a eina dins el programa, fent-lo així accessible a qualsevol dissenyador.

Una característica important de Maya és que és fàcilment adaptable a programari de tercers, el qual pot canviar completament l'aparença de Maya. És precisament aquest aspecte que va fer de Maya un programa molt interessant pelss grans estudis que tendeixen a escriure molt de codi personalitzat per la seva producció. Cal remarcar també que va guanyar un Òscar gràcies a l'enorme impacte que ha tingut en la indústria cinematogràfica com a eina d'efectes especials i visuals, amb un ús molt extès gràcies a la seva capacitat d'ampliació i personalització.

Per la realització del mòdul del generador de l'atles s'ha fet servir la versió 7.0, ja que és la que tenia disponible. Tot i no ser la última versió del mercat, ha estat suficient per la realització dels objectius.

2.2.1- Principals funcionalitats del Maya

En els següents punts expliquem les característiques bàsiques i més importants del Maya de forma general, ja que considerem que no és objecte del projecte aprofundir-hi molt. Per obtenir més informació es pot consultar: [Learning Maya 7].

2.2.1.1 - Geometria

Bàsicament Maya treballa amb malles de poligons i NURBS. NURBS és l'acrònim en anglès de *Non Uniform Rational B-Splines*. Són representacions matemàtiques de geometria 3D capaces de descriure grans quantitats de formes amb precisió, des de simples línies en 2D, cercles, arcs... fins a complicats sòlids en 3D. Gràcies a la seva flexibilitat i precisió, es poden utilitzar models NURBS en qualsevol procés, des d'il·lustració i animació fins la fabricació.

Com es pot comprovar a les imatges 2.3, 2.4, es poden generar primitives bàsiques com ara cubs, esferes, cons... i modificar la mida, l'orientació o la posició amb les eines adequades, tot això des de la mateixa interfície (figura 2.2).

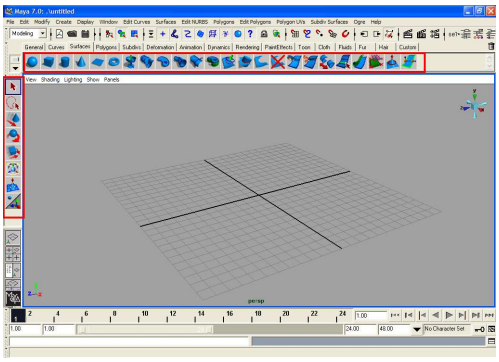


Fig 2.2- La interfície del Maya

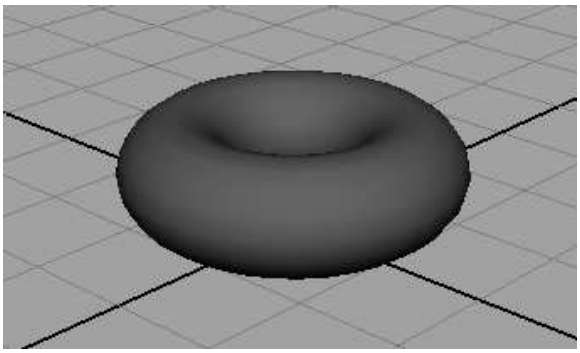


Fig 2.3- Un torus generat pel Maya

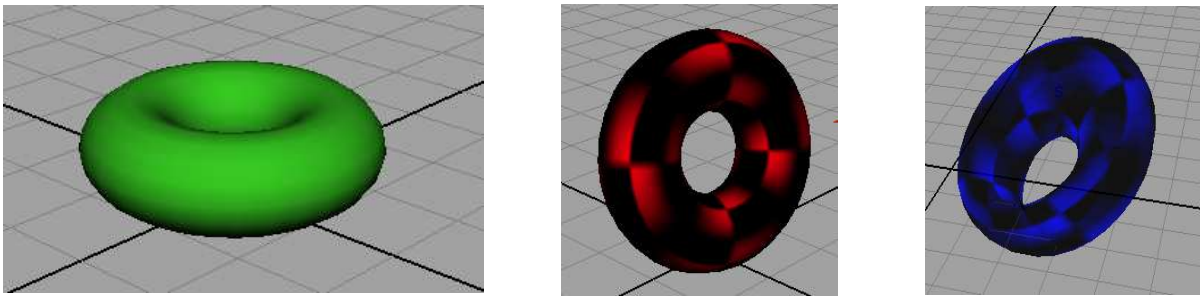


Fig 2.4- Els torus després d'aplicar les transformacions de forma i textura

Com es pot comprovar les possibilitats són extenses. De la mateixa manera que hem estat modificant un torus, hauríem pogut crear un cub, pla o cilindre. A partir d'aquí es pot jugar amb combinacions de geometries bàsiques per obtenir geometries més complexes, com es pot comprovar a la figura 2.5.

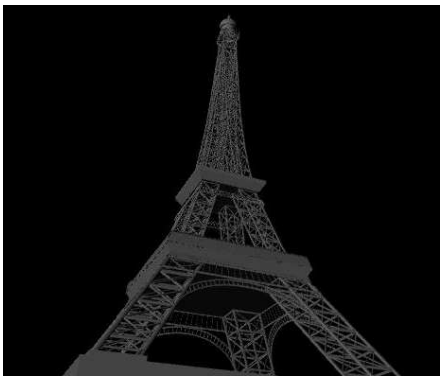


Fig 2.5- La torre Eiffel generada amb el Maya

2.2.1.2 - Animació

Aquest és un dels punts forts del Maya. Gràcies a aquest mòdul hem aconseguit fer caminar el personatge, tot i que, al no ser artistes, l'animació pot no acostar-se a al moviment real d'una persona.

Dins el Maya, qualsevol element es pot animar. L'animació dels elements es pot controlar fàcilment amb la barra de temps (figura 2.6), situada sota de tot de la interfície.

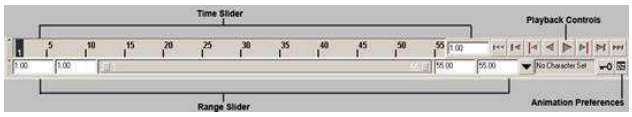


Fig 2.6- La barra de temps

La barra de temps és molt important ja que és la que controla el temps total de l'animació de l'element. Si movem la barra a l'esquerra o a la dreta veurem com l'element va seguint el seu cicle d'animació.

Hi ha 3 tipus d'animació: animació basada en un camí, animació basada en claus i animació no lineal. Nosaltres usarem només l'animació basada en claus.

Animació basada en un camí: bàsicament consisteix en dibuixar un camí a sobre d'una superfície i l'element el va seguint (figura 2.7).

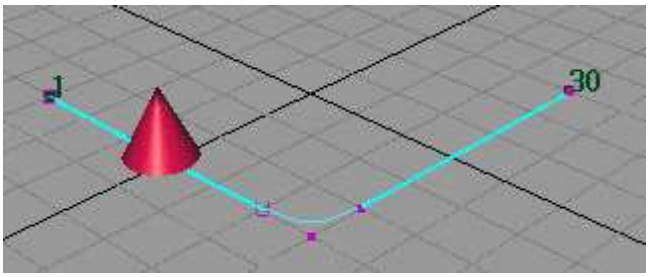


Fig 2.7- Animació per camí

Animació basada en claus: és l'animació estàndard. Consisteix en col·locar una clau per l'element corresponent al principi de l'animació i al final. Aquestes claus són marques en la barra de temps i simbolitzen un punt concret dins l'animació. El Maya llavors interpola el moviment. Observem la figura 2.8 per veure un exemple.

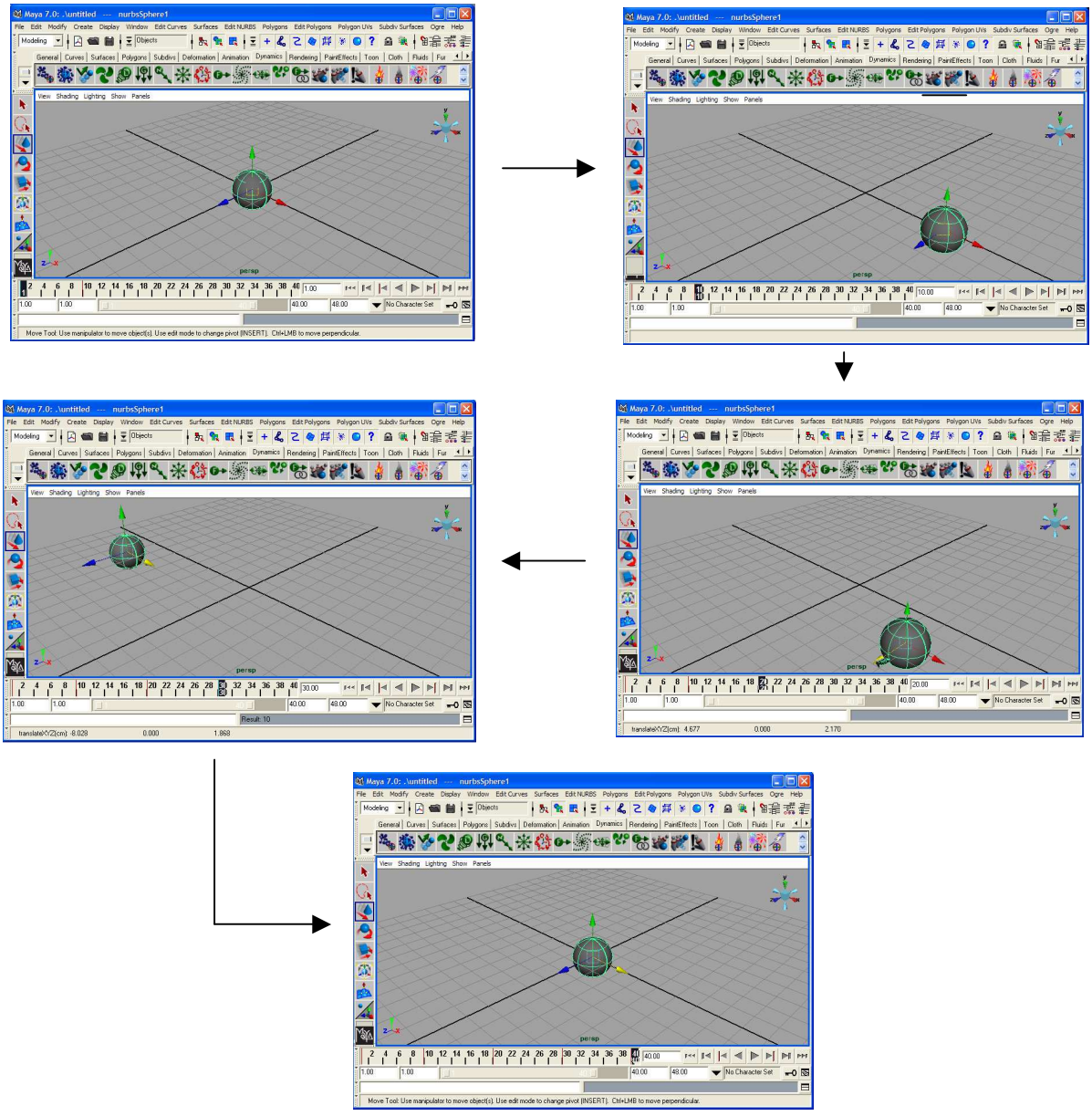


Fig 2.8- El cicle d'animació per claus

Animació no lineal: aquest és un tipus d'animació més avançat. A diferència dels altres tipus d'animacions, aquesta no depèn del temps. Com el seu nom indica, permet generar seqüències d'animacions de forma completament lliure, anomenades "clips".

2.2.1.3 - Llums

El disseny de la il·luminació en el Maya és molt semblant a com es faria a la vida real. Hi ha per triar varis tipus de llums, cadascuna amb els seus avantatges i inconvenients. La tria de la llum és molt important, ja que variant aquesta, l'escena pot canviar de forma molt variada.

En el Maya tenim per escollir varis tipus de llum, d'entre les quals destaquem:

Llum ambient: dóna llum sobre totes les superfícies. Podríem dir que il·lumina de forma uniforme totes les parts de l'escena per igual. S'ha de vigilar amb aquesta llum ja que usada de forma incorrecta ens podria donar mals resultats a l'escena.

Llum direccional: és la que es crea per defecte en les noves escenes. És útil quan es volen emular raigs de llum, per exemple, del sol. Cal dir que els raigs d'aquesta llum són paral·lels els uns amb els altres.

Llum puntual: a diferència de l'anterior, els raigs d'aquesta llum no són pararl·lels, sinó que els envia en totes direccions des del punt on s'ha col·locat. Equivaldria a una bombeta a la vida real.

Llum de con: aquesta llum il·lumina l'àrea definida per un con. Es caracteritza perquè a mesura que els raigs s'allunyen de la direcció inicial, aquests es fan més dèbils, atenuantse.. Per exemple: fars, torres de vigilància, etc...

Llum d'àrea: costosa de visualitzar. Aquesta llum emet raigs des d'un rectangle definit a l'espai. Usada per il·luminacions realistes.

Llum de volum: aquestes llums tenen un rang d'influència molt definit. D'aquesta manera hom pot veure on comença el raig i on acaba. Per defecte la intensitat d'aquests raigs cau de forma lineal amb la distància a l'origen. Útil per il·luminacions d'interiors.

2.2.1.4 - Càmeres

Quan el Maya s'inicia tenim quatre càmeres diferents disponibles: davant, costat, superior i perspectiva. Tres d'elles són ortogràfiques, essent visibles les seves icones per a la translació o rotació. La càmera en perspectiva no té icona per defecte. Tot i això podem moure la càmera amb les tecles i el ratolí, simplificant una mica la tasca del posicionament i orientació.

El posicionament de la càmera ens determina el que es veurà en l'escena final.

Tenim tres tipus diferents de càmeres en perspectiva:

Càmera normal: aquesta càmera rota lliurement. És aconsellable fer-la servir quan es lliga al moviment d'un altre objecte o quan es fixa en un lloc per no moure's.

Càmera i objectiu: aquesta càmera es capaç de tenir un objectiu, el qual no perdrà mai de vista. Cal dir també que es manté sempre mirant a nivell de l'horitzó, motiu pel qual és de les més usades.

Càmera, objectiu i vector normal: és com la càmera anterior, però ara porta un control de vector normal que permet anivellar-la al nostre gust.

2.2.2- El llenguatge MEL

Com hem comentat anteriorment, el Maya disposa d'un llenguatge pròpi per la realització d'*scripts*.

MEL és l'acrònim de *Maya Embedded Language* (Llenguatge Immers de Maya). Aquest forma part del nucli de Maya, siguen la totalitat de la interfície de Maya pràcticament escrita en aquest llenguatge. Degut a això, podem deduïr que tot el que podem fer amb la interfície, ho podem fer amb MEL. MEL ens permetrà crear les nostres pròpies interfícies d'usuari, els nostres propis efectes i automatitzar tasques.

En certa manera MEL deriva del Shell Script de Unix. Posseeix una sintaxi molt semblant a la de C, com més endavant comprovarem. Hem de ressaltar que en ser un llenguatge amb un propòsit específic no trobarem característiques comunes als altres llenguatges de programació, com per exemple la creació de les nostres estructures de dades.

En els següents punts donarem una visió general de les característiques del llenguatge.

2.2.2.1- Sintaxi

A continuació es mostren punts bàsics a tenir en compte per escriure qualsevol sentència en MEL:

- El MEL és sensible a les majúscules. No és el mateix $var que $Var.
- Les sentències van separades per ';'.
- Els comentaris es fan amb '/*','*/' i '//', com en C++.
- Un conjunt de sentències es poden agrupar en blocs, mitjançant '{' i '}'.

2.2.2.2- Operadors

El conjunt d'operadors en MEL és bàsicament un subconjunt dels operadors de C:

- Comparació: == (igual a), != (diferent de), < (menor que), <= (menor o igual que), > (major que), >= (major o igual que).
- Lògics: || (o), && (i), ! (no).
- Increment i decrement de variables: ++,--.
- Indexació de vectors i matrius: [].
- Aritmètics: +,-,*,/.

2.2.2.3- Control de flux

Aquestes sentències ens permeten controlar el flux d'execució dels *scripts* basant-se en condicions booleanes (cert o fals):

- *if-else*: permet prendre una decisió en funció de la certesa o falsedat d'una expressió. Sintaxi idèntica a C:

```
if (<expressió>)
{
     //sentències a executar si <expressió> és
     //certa
}
else
{
     //sentències a executar si <expressió> és
     //falsa
}
```

- *?*, operador ternari:

```
<condició> ? <expressió 1> : <expressió 2>
```

```
//Si condició es certa, executa expressió 1,
//si és falsa executa expressió 2.
```

- *while*: repeteix un bloc de sentències mentre la condició sigui certa:

```
while(<expressió>)
{
     //sentències a executar si <expressió> és
     //certa
}
```

- *for*: la idea és la mateixa que el while, però el for usa una variable sentinella que guia el procés d'iteració:

```
for(<sentències inici iteració>; <condició>
;(<sentències final iteració>)
{
     //sentències
}
```

2.2.2.4- Procediments

Un procediment és un bloc de sentencies associades a un únic identificador. Seria l'analogia amb el concepte de subrutina o subprograma amb el qual qualsevol programador està familiaritzat. Actualment la possibilitat per estructurar els nostres programes és una característica bàsica de qualsevol llenguatge de programació. Usant procediments aconseguim que el nostre codi estigui millor estructurat, guanyant en claretat i modularitat.

Per declarar un script en MEL es fa de la següent manera:

```
     [global]  proc  [  <tipus  de  retorn>  ]<nom  del
procediment>( [<llista d'arguments>] )
     {
          //sentències
          [return <valor de retorn>;]
     }
```

Les expressions que van entre [] siginifica que són opcionals. Per exemple *global:* si no hi porta res vol dir que el procediment és local, significa que només existeix dins de l'script. Tant la llista d'arguments com el valor de retorn ho són perquè es pot donar el cas que el procediment no ho necessiti.

2.2.2.5- Funcions del Maya

Com hem comentat abans, la gran majoria d'accions que es poden fer amb la interfície es poden fer mitjançant funcions. Per exemple crear un polígon, moure la càmera o canviar el temps d'animació, totes aquestes accions tenen una funció en MEL associada.

Com que és impossible resumir en aquest punt totes les funcions que té el Maya hem pensat només comentar les que hem fet servir en el nostre *script*:

- *currentTime* -e: aquesta funció s'usa per assignar un valor a la variable que control el temps a l'escena. Cal posar-hi l'argument *–e* per dir-li que estem actualitzant el temps. Per exemple:

  ```
  currentTime -e 10; //Actualitzem el temps a 10 unitats de temps.
  ```

- *orbit* -ha: aquesta funció controla el moviment de rotació de la càmera, ja sigui horitzontal com vertical. Amb l'argument –ha li diem que estem rotant la càmera sobre el seu eix y, per tant, horitzontal. Per exemple:

  ```
  orbit -ha 12; //rotem 12 graus horitzontalment.
  ```

- *setAttr*: aquesta funció serveix per canviar el valor dels atributs dels objectes. Per exemple, si tenim definida una esfera amb el nom de sph, podem canviar-li el radi així:

  ```
  SetAttr sph.radius 5; //li posem un radi de 5 unitats.
  ```

- *renderWindowEditor snapshot*: aquesta funció és l'encarregada de generar una captura de pantalla. En el següent exemple, es genera una captura de pantalla de la càmera perspectiva, amb un mida de 256 per 256 píxels.

  ```
  renderWindowEditor -snapshot persp 256 256;
  ```

- *saveImage*: aquesta funció serveix per guardar una imatge a la ubicació proporcionada. Per exemple, en la següent línia el que fem és guardar una imatge de la vista actual amb el nom d'imatge a c:\.

  ```
  saveImage –currentView "c:\imatge";
  ```

2.3- Els motors gràfics

Per tal de crear qualsevol aplicació que contingui visualitzacions interactives amb objectes geomètrics, llums i una sèrie d'efectes per tal de donar més realisme a l'escena, cal tenir una sèrie de components que siguin gestionats de forma eficient i no molt complicada. Hem de pensar que dins aquesta escena s'hi ha de poder navegar, interactuar amb l'entorn i també hi ha d'haver càlculs d'il·luminació i visualització dels objectes que la formen.

Un motor gràfic fa referència a una sèrie de rutines de programació que permeten el disseny, la creació i la representació d'aquesta visualització interactiva. L'analogia amb un motor de cotxe és bastant representativa: el motor sota el capó no es veu, però li dóna la funcionalitat de transportar. La mateixa analogia permet explicar, per exemple, alguns dels aspectes que generalment tracta un motor gràfic: les textures i models 3D serien la carrosseria, pintura i interiors. Així doncs, sense un motor, la carrosseria i els exteriors no funcionarien, com tampoc ho farien les textures i models 3D sense un motor gràfic. Entre els motors gràfics de codi lliure més coneguts es poden destacar:

- Nebula
- OpenSceneGraph
- Ogre3D
- Fly3D
- Crystal Space

Tant Nebula com Crystal Space s'orienten molt als videojocs, amb aspectes tècnics com sò i fisica d'objectes 3D. Tenen un codi font difícil de tractar per la seva desestructuració i poca documentació.

El Fly3d és un bon motor gràfic, més simple que els anteriors però no tant potent. Orientat sobretot a l'educació.

Ogre3D i OpenSceneGraph són molt bons motors gràfics amb unes característiques potents, les quals permeten aprofitar millor les possibilitats de programació per la GPU (unitat de processament gràfic) de les targes d'última generació. Els dos tenen bons codis fonts, tot i que l'OpenSceneGraph resulta més complex. Com a punt a favor de l'Ogre3D, podem dir que disposa de major documentació i actualitzacions més freqüents.

Cal també dir que hem estudiat els de codi lliure per la impossibilitat d'aconseguir un motor de pagament, com podria ser per exemple l'Unreal Engine, un motor gràfic molt potent usat en grans quantitats de jocs d'èxit i en vídeo consoles com poden ser la Xbox o PlayStation.

Un cop feta la presentació dels motors hem decidit escollir l'Ogre3D, principalment per dues raons:

- Molt bona documentació i ben explicada (amb una petita *Wikipedia* inclosa). Aquest motor també compta amb una gran comunitat d'usuaris que mantenen un fòrum de consultes, útil per la resolució de problemes o dubtes.
- És un motor gràfic de codi lliure, per tant no hem hagut de pagar res per la seva utilització.

2.3.1- El motor gràfic Ogre3D

L'OGRE (Object Oriented Graphics Rendering Engine) és un motor 3D escrit en C++ orientat a objectes, pel que la seva interfície és clara, intuitiva i fàcil d'usar. Hem de dir que l'OGRE no ha estat dissenyat exclusivament per crear jocs, sinó que és un motor de gràfics 3D en general. Degut a això, no porta suport per so ni física. Això no és un problema ja que gràcies a l'enorme comunitat existent a internet existeixen mòduls especialment dissenyats per OGRE que permeten tenir aquestes facilitats.

La biblioteca permet abstreure tots els detalls de les classes de Direct3D i OpenGL, i proporciona una interfície amb els objectes globals i altres classes necessàries pel desenvolupament d'aplicacions.

L'OGRE és troba disponible sota llicència de GNU Lesser General Públic License (LGPL).OGRE ha servit per dissenyar aplicacions molt variades, des de software educacional (figura 2.9) fins a jocs comercials (figures 2.10 i 2.11).

Fig 2.9- First Aid Sim

Fig 2.10- The ugly prince Duckling

Fig 2.11- Ankh

Tot seguit explicarem les classes més importants que cal conèixer de l'OGRE per tal de, més endavant, entendre la implementació del GdM.

La versió utilitzada per la implentació del GdM ha estat la 1.4.5.

2.3.1.1- La classe *Root*

Com el seu nom indica, és la classe arrel del motor. D'ella en depenen totes les altres classes. Els objectes d'aquesta classe són els primers que cal instanciar, i els últims que s'han de destruir (figura 2.12).

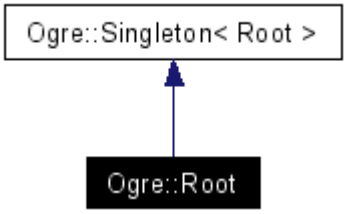


Fig 2.12- Classe Root

Aquest objecte ens permetrà configurar l'OGRE al nostre gust, modificant paràmetres com per exemple la mida de la finestra, la targeta gràfica a utilitzar,

etc... Normalment tot això es configura mitjançant el quadre de diàleg de configuració de l'OGRE (figura 2.13), fet que fa més intuïtiu tot aquest procés.

Fig 2.13- La pantalla de configuració de l'OGRE

És aquest objecte *root* l'encarregat d'engegar el bucle de visualització continua, que ens permetrà visualitzar l'escena creada.

Hem de dir també que l'objecte *root* ens permetrà instanciar les altres classes bàsiques pel funcionament de l'aplicació, com són la *SceneManager* o la *RenderSystem*, que tot seguit detallarem.

2.3.1.2- La classe *SceneManager*

Segurament, la segona classe més important del sistema. Com el nom indica, els objectes d'aquesta classe (figura 2.14) són els encarregats d'organitzar l'escena, és a dir, s'encarreguen de guardar i organitzar tots els elements que apareixen en l'escena per posteriorment enviar-los a visualitzar.

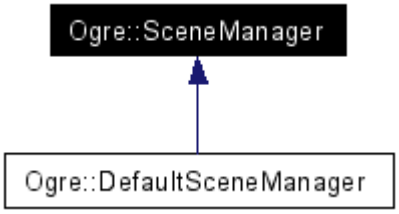


Fig 2.14- Classe SceneManager

Aquests objectes poden ser càmeres, llums o qualsevol altre element que es pugui visualitzar, anomenat Entitat. En aquest punt cal introduir el concepte d'entitat (*Entity*).

Una entitat en OGRE és qualsevol objecte que es pugui posar en escena, com per exemple una persona, un cotxe o un drac. Aquest objecte, a més, ha de ser

movible. Les entitats es basen en malles discretes, com són col·leccions de geometria representades amb l'objecte *Mesh*.

Una *mesh* és la representació d'elements geomètrics mitjançant polígons. Les superfícies es separen en polígons (per exemple triangles) i els volums en poliedres (per exemple tetraedres). Els polígons resultants s'anomenen cel·les. Normalment, com més densitat de cel·les i més petites siguin, més bons resultats obtindrem (figura 2.15) però el cost de procés serà major.

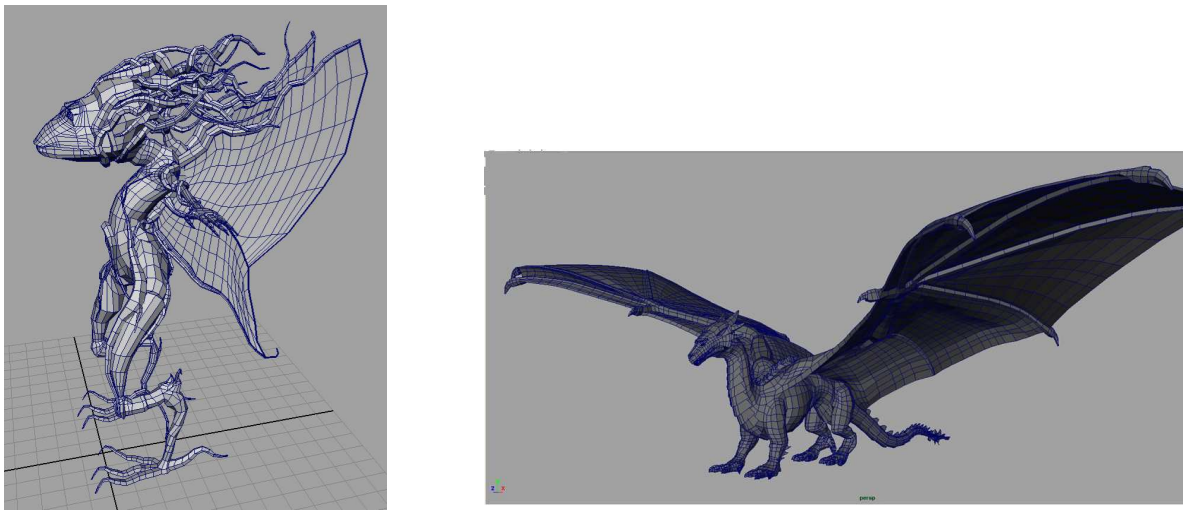


Fig 2.15- Exemples de mesh

Podem treballar amb múltiples entitats basades en la mateixa malla estàtica, donat que ens pot interessar crear múltiples còpies del mateix objecte a l'escena (com és el nostre cas, que simulem una ciutat copiant varis edificis). Aquests objectes no es consideren part de l'escena fins que no són enganxats a un *SceneNode*, que explicarem al següent apartat.

D'aquesta manera, es poden crear grans jerarquies de nodes i entitats, amb complexes relacions entre elles, donant així una gran potència al motor a l'hora de dissenyar una escena amb una forma no massa complicada de controlar tots els objectes que hi apareixen.

2.3.1.3- La classe *Entity*

Com hem comentat al punt anterior, una entitat en OGRE és qualsevol objecte que es pugui posar en escena. Els objectes *Entity* (figura 2.16) es basen en les malles de triangles (*mesh*).

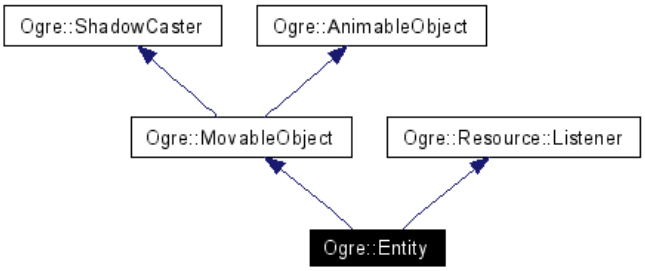


Fig 2.16- Classe Entity

Per tal de crear una entitat invocarem a *SceneManager*::*createEntity*, facilitant-li un nom i especificant el nom de l'objecte *Mesh* en el que es basa. El *SceneManager* s'encarregarà de comprovar que la *Mesh* hagi estat carregada cridant al *MeshManager*. Tenim la limitació que només podem tenir una copia d'una determinada *Mesh* carregada a memòria.

2.3.1.4- La classe *AxisAlignedBox*

Un objecte *AxisAlignedBox* representa un volum alineat amb els eixos de coordenades x, y i z, que envolta l'element al qual pertany (figura 2.17). Podem crear objectes *AxisAlignedBox* pràcticament de qualsevol tipus d'element. Per saber les coordenades d'aquest volum es guarden dos punts (els extrems del volum): un és el punt mínim dels tres eixos i l'altre és el punt màxim. Aquesta classe s'usa principalment per col·lisions i per determinacions de visibilitat.

Aquests volums són principalment tres: un rectangle en 3D, una esfera i un cilindre.

Els objectes d'aquesta classe ens han estat molt útils per crear el nostre mòdul de col·lisions.

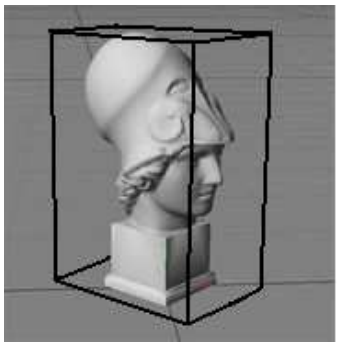


Fig 2.17- Un exemple de Bounding Box

2.3.1.5- La classe *SceneNode*

Els objectes de la classe *SceneNode* (figura 2.18) s'utilitzen per organitzar objectes dins d'una escena. Tenen les mateixes propietats jeràrquiques que els objectes de tipus *Node* genèric, però a més ofereixen la propietat de ser capaços d'afegir objectes a l'estructura i emmagatzemar-los en estructura d'arbre lligats al node que correspongui.

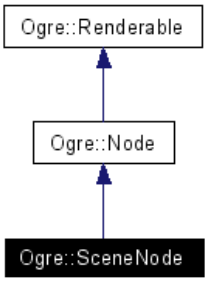


Fig 2.18- Classe SceneNode

Els nodes fills són continguts dins dels límits del seu node pare, donant molt de joc al procés.

2.3.1.6- La classe *RenderSystem*

La classe *RenderSystem* (figura 2.19) és una classe abstracta. Aquesta classe defineix la interfície de la API3D inferior. Això vol dir que les aplicacions han de ser visualitzades per una altre API específica, com per exemple Direct3D o OpenGL.

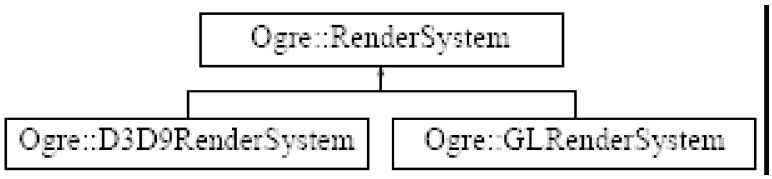


Fig 2.19- Classe RenderSystem

2.3.1.7- La classe *Càmera*

Els objectes d'aquesta classe (figura 2.20) són els encarregats de permetre'ns veure el que hi ha a l'escena. El que "veu" la càmera és el que s'envia a visualitzar, normalment a una finestra.

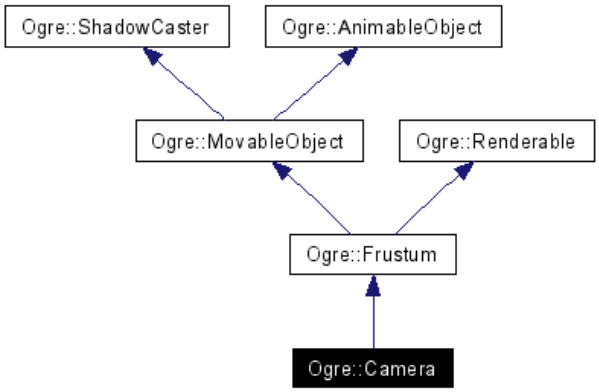


Fig 2.20- Classe Camera

Aquestes càmeres suporten projeccions ortogràfiques (l'objecte no canvia de mida a mesura que ens anem allunyant) i projeccions en perspectiva (a mesura que ens allunyem, l'objecte es fa més petit).

Hem de destacar que les càmeres es poden adjuntar a un *SceneNode*, cosa que farà que la càmera tingui la posició del node al qual està enganxada. Això és útil si volem que la càmera estigui sempre associada amb un objecte de l'escena, per exemple.

2.3.1.8- Les classes *BillboardSet* i *Billboard*

Aquestes dues classes (figura 2.21) són de les més importants degut al seu pes en aquest projecte.

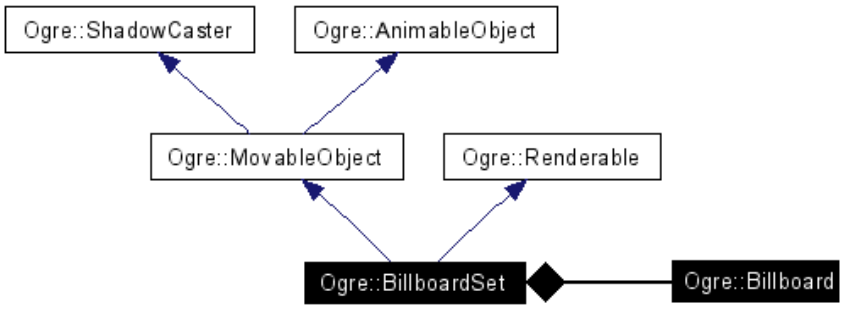


Fig 2.21- Classes *BillboardSet* i *Billboard*

Primer de tot expliquem què és un *Billboard*.

Un *billboard* és una primitiva (triangle, rectangle, etc.). En aquest cas seran dos triangles enganxats, formant un rectangle) que sempre està orientat a la càmera, sempre l'està mirant. No importa la posició on estigui, sempre estarà perpendicular a l'observador. Majoritàrament s'usen per crear efectes, tals com

partícules, fum, núvols...o persones. A aquests *billboards* podem assignar-los qualsevol textura. La classe *Billboard* és la que ho implementa.

Tots els *billboards* s'agrupen en *BillboardSets*.

Els objectes de la classe *BillboardSet* agrupen objectes *Billboard*. El motor OGRE dóna certs consells per l'òptim rendiment d'aquesta classe quan apareixen a escena centenars de *billboards*, tals com:

- Tots els *billboards* haurien de tenir la mateixa mida (així s'evita de fer càlculs innecessaris). Òbviament no és un requisit, i podem tenir *billboards* amb mides diferents (per exemple si volem crear núvols de fum).
- No crear mai *billboards* sense assignar-los a un *billboardSet*. És preferible tenir un *billboardSet* només amb un *billboard* que no pas un *billboard* sol sense cap *billboardSet* associat.

2.3.1.9- La classe *Image*

Com el nom indica, els objectes de la classe *Image* implementen la representació d'un fitxer d'imatge.

Normalment guarda les dades d'imatges sense compressió i és l'únic objecte que pot ser carregat en una textura. Aquests objectes són usats normalment quan volem carregar una imatge en una textura en l'escena o quan hem de fer algun preprocessament a alguna imatge abans de mostrar-la.

La imatge es sol guardar en una taula de *bytes*, i els píxels es guarden un darrere l'altre. Cada píxel ocupa com a mínim tres posicions de la taula, ja que ha de guardar informació per les tres components *red, green* i *blue*. L'estructura de la imatge en la taula es pot veure a la figura 2.22.

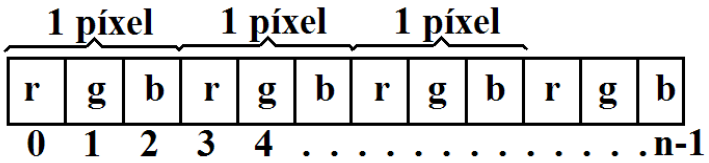


Fig 2.22- L'esquema de com es guarda a memòria una imatge en OGRE

On *n* és l'amplada de la imatge × l'alçada de la imatge × 3. Per exemple, una imatge en format *jpg* de 640x480 ocuparia una taula de 640*480*3 posicions = 921.600 posicions. Per tant, cada 640 píxels*3 posicions seria una fila de píxels de la imatge.

A memòria, aquesta imatge ocuparia 640*480*3*4 = 3.686.400 bytes = 3.687 KB.

2.3.1.10- La classe *FrameListener*

Quan visualitzem un entorn tridimensional en temps real, la sensació que tenim és la de que el procés és dinàmic, i per tant no percebem les imatges per separat, sinó com si veiéssim una seqüència de vídeo continuada. La *taxa* amb que les imatges són visualitzades es mesura en imatges per segon (*frames per second, fps*). El cervell humà es capaç de percebre com a imatges separades, freqüències de canvi d'imatge per sota dels 15 fps, aproximadament. Podem donar per vàlid que un entorn visualitzat a més de 15 fps és a temps real, i que l'usuari es concentrarà en l'acció i la reacció. Cal també tenir en compte que no té gaire sentit visualitzar per sobre dels 72 fps, ja que les diferències són imperceptibles, a part de malbaratar recursos.

La classe *FrameListener* (figura 2.23) diem que és una classe que "escolta", és a dir, que resta esperant senyals i events per poder cridar els mètodes corresponents sobre aquestes senyals. També és l'encarregada de visualitzar un fotograma (*frame*) a cada crida. Una pràctica habitual dels usuaris del motor OGRE és extendre aquesta classe base, és a dir, sobreescrivint els seus mètodes per tal d'aconseguir els objectius marcats. Aquests mètodes són dos:

- *Bool frameStarted (const FrameEvent &evt)*
- *Bool frameEnded (const FrameEvent &evt)*

El *frameStarted* es crida cada vegada abans de processar el frame i el *frameEnded* es crida tot just després d'haver-lo visualitzat. Tots dos retornen un booleà. D'aquesta manera es controla si es continua visualitzant o pel contrari s'ha d'aturar la visualització dels frames.

Hem de dir que aquests mètodes són els que van dins del bucle de visualització, així es van generant *frames* a cada volta, aconseguint el que comentàvem al principi: obtenir una continuitat en la visualització dels fotogrames donant interactivitat a la nostra aplicació. En aquest context, el *frame rate* aquí seria la quantitat de voltes (per tant la quantitat de vegades que es criden aquests mètodes) que pot fer el bucle de visualització cada segon.

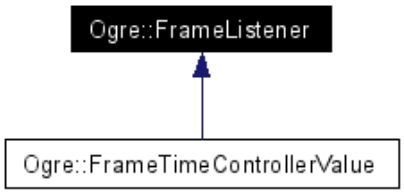


Fig 2.23- Classe FrameListener

Hem de comentar també que en el GdM només hem redefinit el *frameStarted*, ja que el *frameEnded* no l'hem necessitat a causa de les característiques que aviem d'implementar.

2.4- Els *shaders*

Des que va començar la "revolució 3D" en l'àmbit dels jocs d'ordinador a mitjans de la dècada dels 90, la tendència de la tecnologia aplicada a aquests temes ha estat traslladar la feina de processament dels gràfics tridimensionals, des de la CPU a la tarja de video. El que es volia aconseguir amb aquest canvi era descarregar la CPU de tasques purament orientades als gràfics, i d'aquesta manera, poder dedicar més temps i recursos a altres tipus de càlculs.

Al principi això es feia a molt baix nivell, en llenguatge assemblador. La veritat és que donava molt de poder al programador, però tenia una alta complexitat a l'hora d'escriure codi.

Per aconseguir traspassar a la GPU (unitat de processament gràfic) aquestes tasques de forma més pràctica i no tant feixuga es van crear els *shaders*.

Un *shader* es defineix com un conjunt d'instruccions escrites en un llenguatge d'alt nivell, que compilen i s'envien a la GPU per tal que aquesta les processi i realitzi les tasques donades per aquestes instruccions.

Com a llenguatges de *shader* d'alt nivell tenim el Cg, que pertany a la companyia Nvidia (significa "C" per gràfics). El GLSL pertany a la llibreria OpenGL. Hem de dir que els *shaders* creats pel GdM han estat creats amb el Cg.

2.4.1- Procés de visualització

En l'actualitat, les targes gràfiques consten de tres unitats que permeten incorporar elements programables a les etapes de transformació de vèrtexs i transformació final del color de cada píxel. Aquestes unitats són el *Vèrtex Shader*, el *Geometry Shader* i el *Píxel Shader* (anomenat també *fragment program*, més endavant veurem perquè).

Els *vertex shaders* s'encarreguen de la transformació dels vèrtexs d'una escena. Reben només un vertex cada vegada i només el poden modificar. No poden afegir-ne de nous ni eliminar-ne.

Els *píxel shaders* s'encarreguen de la transformació dels píxels de l'escena. Calculen el color de cada píxel individualment. Són usats bàsicament per calcular il·luminacions.

Els geometry shaders s'executen després dels vèrtexs shader. Reben sempre una primitiva, com per exemple triangles i, a poder ser, amb informació d'adjacència. Per exemple, si rep un triangle, l'entrada del *shader* seran els tres vèrtexs.

El fet de separar-los va proporcionar als programadors una major llibertat a l'hora de de dissenyar gràfics en tres dimensions, ja que poden tractar-se cada píxel, cada triangle i cada vèrtex per separat. D'aquesta manera, els efectes especials i la il·luminació es poden crear més detalladament, succeïnt el mateix amb la geometria dels objectes.

A la figura 2.24 es pot veure el gràfic del procés de visualització.

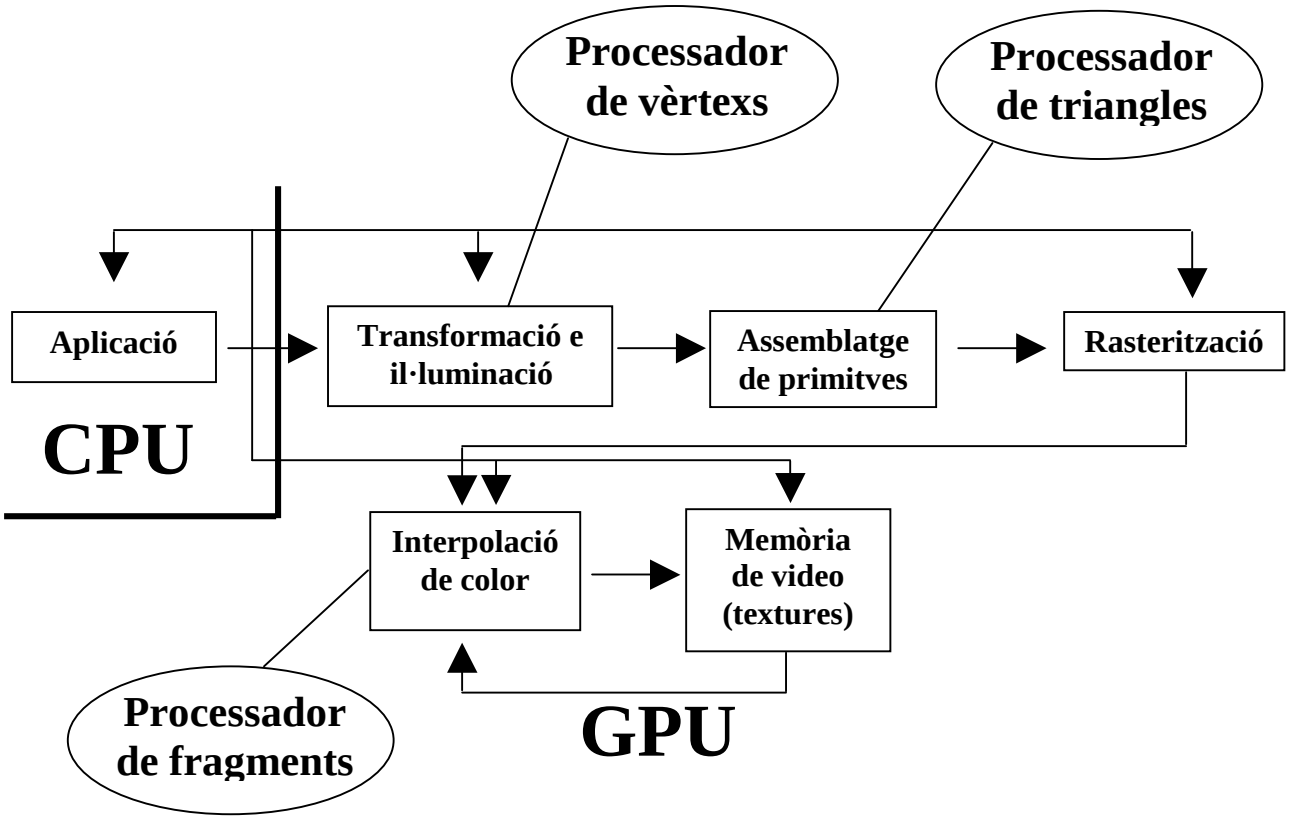


Fig 2.24- El procés de visualització

Els passos que se segueixen són els següents:

1. La CPU envia les instruccions (llenguatges de *shader* compilats) i la geometria a la GPU.
2. Dins el vertex *shader* es realitzen les operacions de transformació i eventualment de càlcul d'il·luminació.
3. Es transformen els vèrtexs de coordenades de món 3D a coordenades de pantalla 2D (fig 2.10).

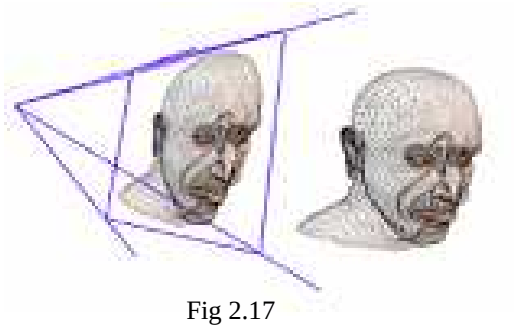


Fig 2.17

4. La geometria calculada (els vèrtexs) són combinats per formar primitives de visualització, normalment triangles, però també poden ser punts, línies o polígons. Cada primitiva és processada pel *Geometry Shader*.
5. Un cop obtingudes aquestes primitives, es realitza un procés de retallat, que consisteix en eliminar les primitives o part de primitives, que queden fora del *frustum*, que és l'àrea de l'espai tridimensional que podem observar a la vista de l'escena.
6. A continuació les primitives es rasteritzen i passen a ser píxels, amb un color associat (fig. 2.11).

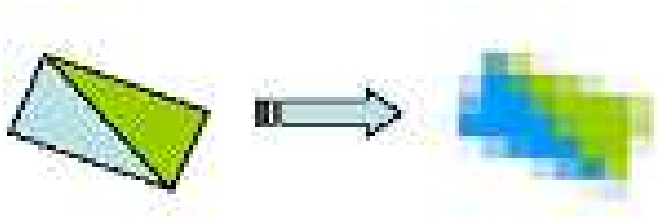


Fig 2.18

7. Durant el procés de rasterització és realitza la interpretació de les dades associades a cada vèrtex de la primitiva (per exemple, les coordenades de textura). Aquests píxels generats per la rasterització i les seves dades interpolades defineixen el que coneixem com a fragments. Un *fragment* és un tros de polígon amb

dimensions d'un píxel, candidat a ser mostrat per pantalla. Cada fragment és processat per la unitat programable *Pixel Shader*.

8. En el píxel shader és on es realitzen les transformacions de color sobre aquests píxels (fig. 2.12).

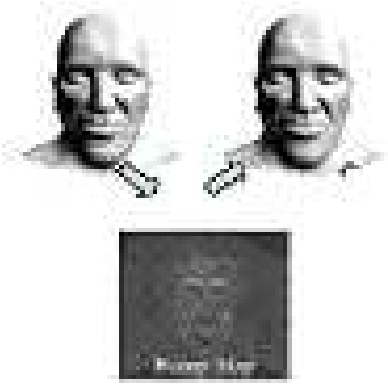


Fig 2.19

9. Un cop finalitzada la transformació del fragment, aquest és inclòs al *buffer* de pintat.

2.4.2- El llenguage de shader Cg

El Cg és un llenguatge d'alt nivell desenvolupat per Nvidia amb la col·laboració de Microsoft, per programar *Vèrtex Shaders, Geometry Shaders* i *Píxel Shaders*.

El Cg està basat en el llenguatge C i, tot i que comparteixen la mateixa sintaxi, algunes de les característiques de C es van modificar i es van afegir nous tipus de dades per fer el Cg més adequat per programar GPU's. El Cg només serveix per programar targes gràfiques, no substitueix a cap altre llenguatge general d'alt nivell.

Tipus de dades

Els tipus de dades bàsics en Cg són:

- float - a 32bit floating point number
- half - a 16bit floating point number
- int - a 32bit integer
- fixed - a 12bit fixed point number
- bool - a boolean variable
- sampler* - representa una textura

Cg també contempla els tipus vector i matrix, per exemple el float4x4 i el float3.

Operadors

En quant els operadors, podem dir que són els mateixos que en C i que en la majoria de llenguatges, afegint els operadors aritmètics per vectors i matrius.

Estructures de control i funcions

Cg comparteix les estructures bàsiques de control de C, com if/else, while i for. La manera de definir funcions és pràcticament la mateixa.

Llibreria de Cg

Com en C, Cg té una sèrie de funcions per realitzar tasques comunes en la programació de GPU's. Funcions com les matemàtiques tenen el seu equivalent en C, però d'altres són especialment dissenyades per ell, com per exemple les funcions per mapejar una textura, tex1D i tex2D.

Vegem un exemple per entendre aquest nou llenguatge:

```
// input vertex. Dades del vèrtex a transformar passades per
//l'aplicació
struct VertIn
{
      float4 pos   : POSITION;
      float4 color : COLOR0;
};

// output vertex. Vèrtex transformat i color pel vèrtex
//calculat
struct VertOut
{
    float4 pos   : POSITION;
    float4 color : COLOR0;
};


// vertex shader main
VertOut main(VertIn IN, uniform float4x4 modelViewProj)
{
      VertOut OUT;//variable per la sortida
      //calculem les coordenades finals
      OUT.pos     = mul(modelViewProj, IN.pos);
      //copiem el color d'entrada a la sortida
      OUT.color   = IN.color;
      //posem la component Blava del color a 1
      OUT.color.b = 1.0f;
      return OUT;
}
```

Com podem veure és pràcticament igual que el C, únicament canvien algunes operacions i alguns tipus de dades utilitzats.

Capítol 3: el Generador de Multituds

En els següents apartats explicarem les classes que hem creat per tal de desenvolupar el GdM. Parlarem de totes les classes que intervenen en el funcionament del programa.

Hem estructurat aquest capítol per mòduls, per a una millor comprensió: mòdul de generació de l'atles, mòdul de creació de l'escenari, mòdul de preprocés, mòdul de control de col·lisions, mòdul de generació de multituds i finalment el mòdul de la interfície d'usuari.

A la figura 3.1 es es mostra el diagrama de classes del GdM.

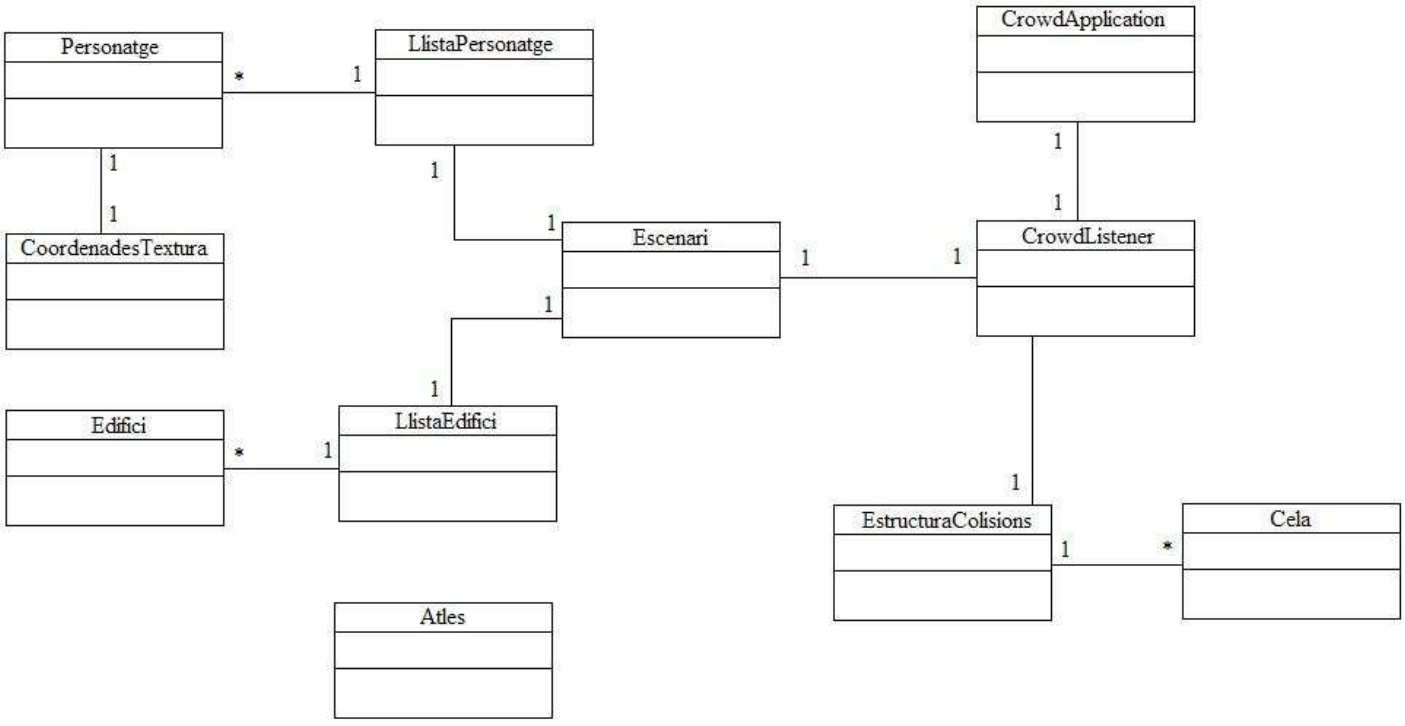


Fig 3.1- El diagrama de classes del sistema

Comencem, doncs, parlant de l'animació del personatge, de l'obtenció de les fotografies i de la generació de l'atles: el mòdul de la generació de l'atles.

3.1- El mòdul de generació de l'atles

En aquest apartat explicarem tot el mòdul que hem implementat per l'obtenció de les fotografies del personatge i la posterior generació de l'atles mitjançant totes les fotografies obtingudes.

3.1.1- Animació del personatge

Abans de començar a pensar com fer la multitud, necessitem un personatge base per crear-la. Hem pensat que agafant un personatge ja creat ja n'hi ha prou per començar a treballar, ja que si l'haguéssim hagut de crear nosaltres hauríem perdut temps en detriment d'altres aspectes més constructius del projecte.

El personatge que hem animat s'ha extret d'una pàgina web que proporciona dissenys de Maya per lliure distribució (www.highend3d.com). De tots els que ens ofereix ens hem quedat amb el model d'una dona, ja que és el que més realisme dóna en comparació amb la resta de dissenys. A la figura 3.2 podem observar la postura inicial del personatge.

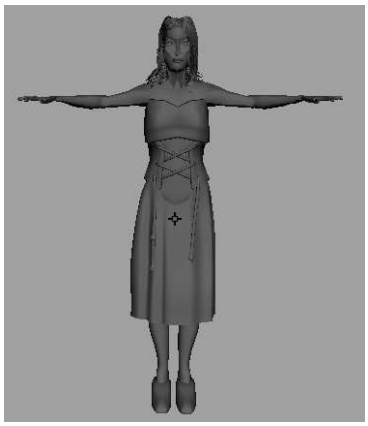


Fig 3.2- El personatge original

A partir d'aquí el que hem fet ha estat modificar-lo perquè pugui permetre moviments lliures a les seves extremitats, ja que en el model inicial les extremitats i el cos eren un sol objecte, i el que necessitem és que siguin objectes independents per la seva posterior animació.

El primer que s'ha fet ha estat separar cames i braços del cos donant-los a cadascuna una textura que doni més realisme al model. Un cop fet això ja tenim el model amb les parts separades i només ens falta unir-les de nou al tronc. El resultat final del personatge es mostra a la figura 3.3.

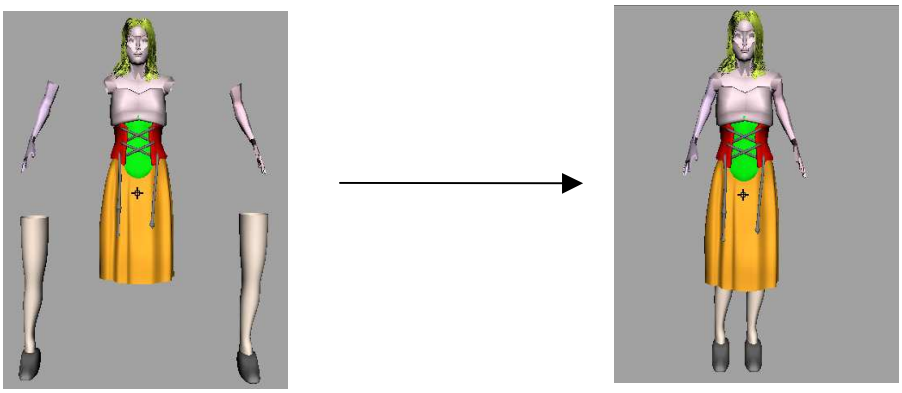


Fig 3.3- El nou personatge

Un cop tenim el personatge a punt procedim a animar-lo.

Hem pensat que la manera més fàcil d'animar-lo és l'animació per claus, o sigui, dient-li quines posicions hauria d'adoptar, i a quins instants, i deixar que el Maya interpolés les posicions intermitges (figura 3.4). També cal remarcar que per realitzar l'animació del moviment, un artista hagués pogut utilitzar la tècnica d'ossos, però aquesta està més enllà dels objectius del projecte.

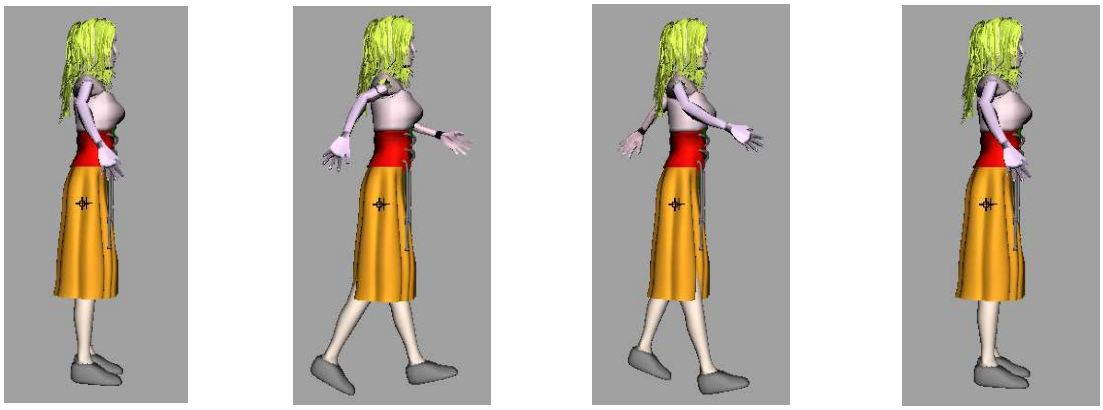


Fig 3.4- Instants del cicle de moviment del personatge

Al final ens va quedar el personatge amb un moviment prou realista per fer-lo servir en la multitud, com es podrà comprovar més endavant.

3.1.2– Obtenció de les fotografies per l'atles

Per la generació de l'atles necessitem obtenir moltes fotografies del personatge d'una forma que no sigui feixuga. Les imatges obtingudes han de ser del personatge caminant, obtingudes des del seu voltant i per cada estat del seu caminar. L'esquema usat és el de la figura 3.5.

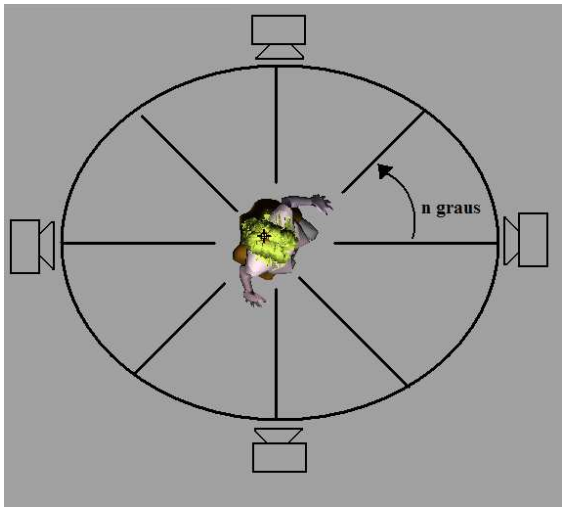


Fig 3.5- L'esquema de l'obtenció de les fotografies

Primer de tot es comencen les fotografies quan la noia està en repòs, o sigui, que encara no ha començat a caminar. Mentre això dura se li van fent fotos des de diferents angles. Per exemple, es comença per la foto quan la càmera està a la posició de 0º. Un cop feta la foto es rota la càmera *n* graus, sense canviar l'estat ni la posició de la noia. Es torna a fer la foto i es torna a rotar la càmera. Això es va fent fins que la càmera completa el cicle.

Ara, un cop feta la volta, cal canviar l'estat de la noia, i tornar a començar el procés. Quan la noia hagi arribat al moment en que l'obertura de les cames i braços és màxima (al final del moviment) obtindrem el que mostra la figura 3.6.

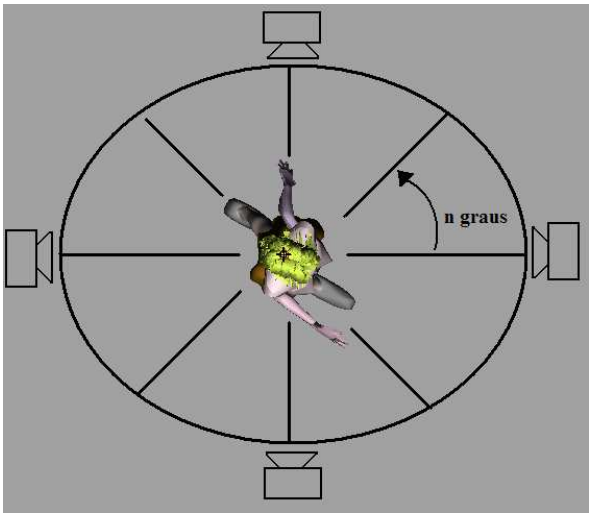


Fig 3.6- L'esquema de l'obtenció de les fotografies, amb un nou moviment

La idea és: per cada moviment de la noia, tirem *n* fotos durant tot el cicle d'animació de la noia. D'aquesta manera obtenim en una sèrie de fotografies tot el moviment complet de la noia des de *n* angles diferents.

Per programar això amb el Maya hem decidit implementar un petit *script* en MEL per automatitzar aquesta tasca i fer-la menys costosa.

Les funcions utilitzades en aquest *sript* ja estan explicades a l'apartat d'introducció al MEL, juntament amb la sintaxi d'aquest llenguatge de programació. Aquí simplement comentarem el que s'aconsegueix usant aquestes funcions en aquest context. Com a comentari final hem de dir que les fotografies s'han guardat en format PNG, ja que aquest conté informació de la transparència (així no tenim taques negres durant l'execució) i la imatge resultant és de bona qualitat. A més, la mida de les imatges és de 256 × 203 píxels, ja que si es fan a menys mida, durant l'execució del GdM no es veuen massa bé, i si es fan a més mida, l'atles de textura resultant és més gran de 4096 × 4096 píxels, cosa que el fa inviable perquè la nostra tarja gràfica no accepta resolucions majors que aquesta. Finalment, hem decidit fer 32 fotografies al personatge i avaluar a 20 instants de temps de la seva execució (32 fotografies per instant, en total obtenim 640 imatges del seu moviment). Vam triar aquests valors després de nombroses proves, comprovant la qualitat final de l'execució del GdM. A la figura 3.7 podem veure l'esquema final de l'atles.

L'script final queda d'aquesta forma:

```
setAttr "defaultRenderGlobals.imageFormat" 32;
//amb aquesta línia estem dient-li al Maya que les imatges guardades des
//de la vista de visualització s'han de guardar amb el format 32, que
//significa el format PNG.

renderIntoNewWindow render;
//amb aquesta línia fem que el Maya ens obri en la finestra de
//visualització l'escena actual.

string $desti;
//destí serveix per posar nom a les imatges. Simplement les anomenarem
//0.png, 1.png,..,n.png; per simplicitat.
string $aux = "llocOnGuardarImatges";


//32 fotos
//20 instants del personatge
//640 imatges en total

$h=0;
for($t=0;$t<100;$t=$t=$t+5)
{
      //aquest for controla els instants d'animació de la noia.
      currentTime -e $t;
      //posem el cicle de moviment del personatge a l'instant donat per t
      for($i=0;$i<32;$i++)
      {
```

```
            //aquest bucle controla la posició de la càmera.

            $desti=$aux+$h;
            //concatenem el path on guardar les imatges amb el nom (ja
            //tenim el path complet!)

            $h=$h+1;
            //actualitzem la variable, per no repetir noms

            renderWindowRenderCamera snapshot renderView persp;
            //diem que la càmera de la finestra de visualització faci una
            //captura de pantalla de l'escena, amb la càmera en mode
            //perspectiva

            renderWindowRender redoPreviousRender renderView;
            //tornem a rendritzar per no perdre la finestra

            renderWindowSaveImageCallback "renderView" $desti "PNG";
            //guardem com a fitxer amb extensió png la captura de càmera.
            //Ja tenim la fotografia!

            orbit -ha -11.25;
            //rotem la càmera 11.25 graus en sentit antihorari respecte
            //l'eix y. Ho fem de forma negativa per convenció.

            renderWindowRender redoPreviousRender renderView;
            //tornem a visualitzar per tenir a punt l'imatge de l'escena
            //per la propera fotografia
        }
}
```

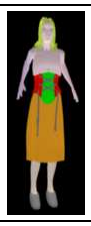
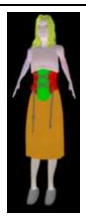
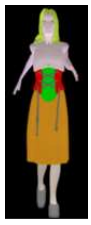
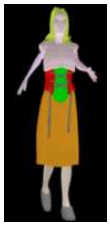
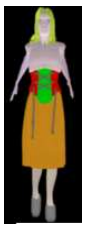
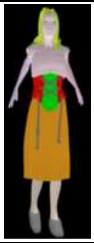
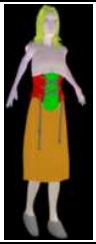
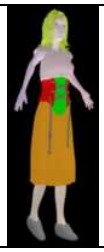
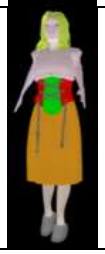
	0º	-11.25º	-22.5º	-33.75º	...	-348.75º
Instant 0						
Instant 20						
Instant 40						

Fig 3.7- L'esquema final de l'organització de l'atles

La utilitat d'obtenir les fotografies del personatge l'hem integrat a la interfície del Maya, fent-la accessible a tothom mitjançant la icona corresponent a la barra d'eines (figura 3.8).

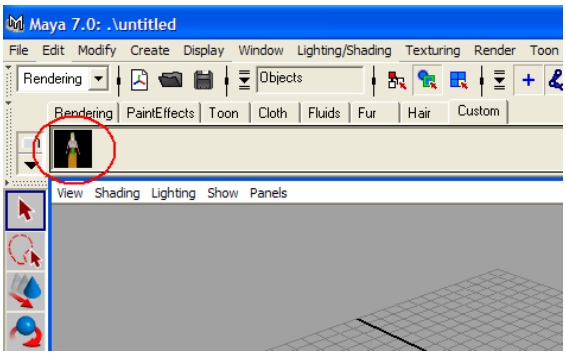


Fig 3.8- L'icona al menú del Maya del generador de fotografies

3.1.3– Generació de l'atles

Un cop ja tenim les fotografies del personatge guardades en una carpeta, hem de generar l'atles de textures.

En principi, el mòdul de generació de l'atles només s'ha d'executar una vegada, ja que un cop creat l'atles no cal tornar-lo a generar a cada execució, ja que el tenim emmagatzemat al disc dur. Per tant, l'atles ja es proporciona amb el programa. Expliquem, però, com ho hem fet per obtenir-lo.

3.1.3.1- La classe Atles

Per tal d'obtenir l'atles de textura ens hem creat la classe Atles. Aquesta classe (figura 3.9) només té un mètode: crear(). La seva funció és, a partir de les *n* imatges inicials del personatge, generar una única imatge final: l'atles de textura.

Per tal de fer més comprensible el mètode li hem afegit només un atribut: imatgeAtles. Aquest atribut és simplement una estructura *(struct)* que guarda un punter a una taula de *bytes,* que és com l'OGRE guarda la informació de la imatge a memòria. Aquest atribut s'anomena infoImatge.

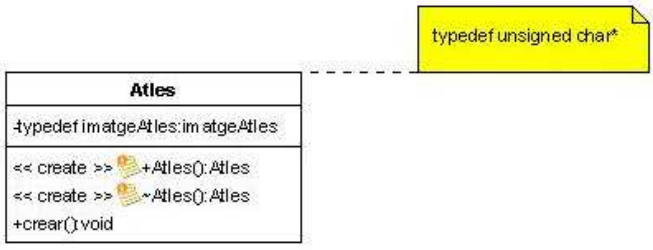


Fig 3.9- La classe Atles

3.1.3.2- El mètode crear()

Com hem comentat abans, és l'unic mètode de la classe, i l'encarregat de generar l'atles. L'atles té la següent forma: les columnes són els graus de la càmera, i les files són el moviment del personatge. Per aconseguir això el que fa el mètode és agafar les 32 primeres imatges (les fotografies del moviment al primer instant). De totes aquestes imatges agafa la primera fila de píxels i l'enganxa a la imatge final. Fa el mateix amb la segona fila de píxels, i continua fins l'última. Quan s'han acabat els píxels de les primeres 32 imatges agafa les 32 següents. I repeteix el procés, enganxant aquests píxels un darrere l'altre. Al final han de quedar a les columnes les diferents fotografies i a les files els diferents instants. Vegem el codi:

```
Vector_imatge_atles llistaImatges;
//hi guardarem les imatges obtingudes del Maya

Image i;
Image imatgeFinal;
```

numImatges,amplada i alçada es refereixen a les imatges guardades pel Maya.

```
for(i=0;i<numero d'imatges;i++);
{
      i.load("path");
      llistaImatges[p]=i
}
//ja tenim a memòria totes les imatges del Maya. Anem a crear
//l'atles
```

Per cada píxel, OGRE guarda 4 components: red, green, blue i alpha, ja que el format és PNG. Per tant, la imatge resultant s'haurà de guardar en una taula de *unsigned char* de la mida 4 * ampladaImatgeMaya * alçadaImatgeMaya * numImatgesMaya, per tant, 4 * 256 * 203 * 672 = 139.689.984 posicions.

```
index=0;

for(w=0;w<numImatges*amplada*alçada*4;w=w+amplada*32*4)
{
      //anirà indexant les posicions de la imatge final
      for(k=0;k<numImatgesAtles;k=k+32)
      {
            //recorrerà les imatges de 32 en 32
            for(a=0;a<alçada*amplada*4;a=amplada*4)
            {
                  //per recórrer cada cada píxel de la imatge
                  for(p=0;p<32;p++)
                  {
                        //de les 32 imatges que carreguem, una a
                        //una
```

```
                        for(h=0;h<amplada*4;h++)
                        {
                              //per anar de fila en fila d'una
                              //imatge
                              imatgeFinal[index]=llistaImatges[k+p]
                                   .infoImatge[a+h] ;
                              index++;
                        }
                  }
            }
      }
}
imatgeFinal.save("path final") ;
//a path final ja hi tenim el nostre atles de textura!
```

Aquest és l'encarregat de crear el nostre atles. Però tal com està escrit aquest codi ens passem de límit de resolució, ja que si fem les fotografies a 256x203 píxels i hi ha 32 imatges per instant, la resolució horitzontal de l'atles és de 256*32=8192, i la nostra tarja té com a límit imatges de 4096. A continuació explicarem com solucionar-ho.

3.1.3.3- El mètode crear() millorat

Si observem bé les imatges resultants (figura 3.10) veurem que hi ha molt d'espai que surt a la fotografia que en realitat no es fa servir.

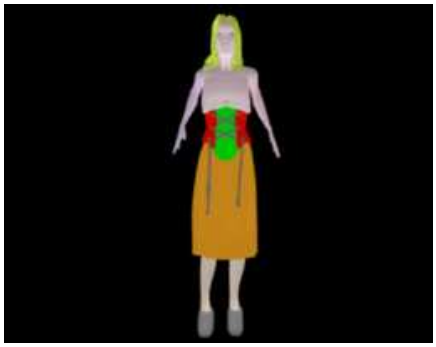


Fig 3.10- Una imatge de l'atles sense optimitzar

La idea és "retallar" aquestes imatges i, amb l'espai obtingut, fer que càpiguen en una resolució de menys de 4096. Per això podem pensar que, si com a màxim la imatge resultant ha de fer 4096, i tenim 32 imatges, això vol dir que per cada imatge hauríem d'ocupar com a màxim 128 píxels d'amplada. Si les que tenim ara fan 256 píxels d'amplada, vol dir que hem de reduïr a la meitat la seva amplada (l'alçada no l'hem de modificar perquè ja està dins els límits). Per tant, podem pensar que traient 64 píxels per cada costat ja perderíem aquests 128 píxels de sobres (figura 3.11).

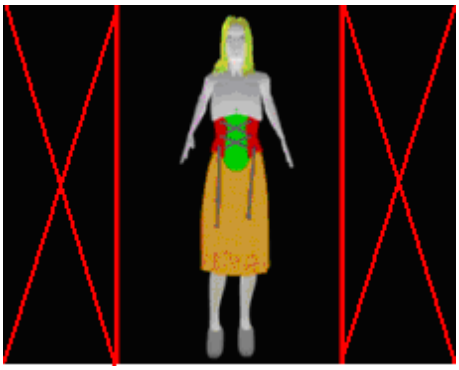


Fig 3.11- La mateixa imatge on es pot observar la mida final, traient l'espai sobrant

En vermell hem marcat com quedaria la imatge final si retalléssim 64 píxels per cada costat. Com podem veure ja ens va bé.

Al codi següent podem observar com amb unes petites modificacions a l'algoritme aconseguim un atles més optimitzat:

NOTA: només hem escrit el codi on hi ha modificacions.

```
index=0;

for(w=0;w<numImatges*(amplada-128)*alçada*4;w=w+(amplada-128)*32*4)
{
      //anirà indexant les posicions de la imatge final
      for(k=0;k<numImatgesAtles;k=k+32)
      {
            //recorrerà les imatges de 32 en 32
            for(a=0;a<alçada*amplada*4;a=amplada*4)
            {
                  //per recórrer cada cada píxel de la imatge
                  for(p=0;p<32;p++)
                  {
                        //de les 32 imatges que carreguem, una a
                        //una
                        for(h=0+64*4;h<(amplada-64)*4;h++)
                        {
                              //per anar de fila en fila d'una
                              //imatge
                              imatgeFinal[index]=llistaImatges[k+p]
                                    .infoImatge[a+h];
                              index++;
                        }
                  }
            }
      }
}
imatgeFinal.guardar("path final") ;
//a path final ja hi tenim el nostre atles de textura!
```

Com podem comprovar, gràcies a només tres canvis hem millorat el mètode.

3.1.3.4- L'atles final

Un cop feta la modificació de l'algoritme de crear l'atles, a la figura 3.12 es pot observar l'aspecte final que presentarà l'atles.

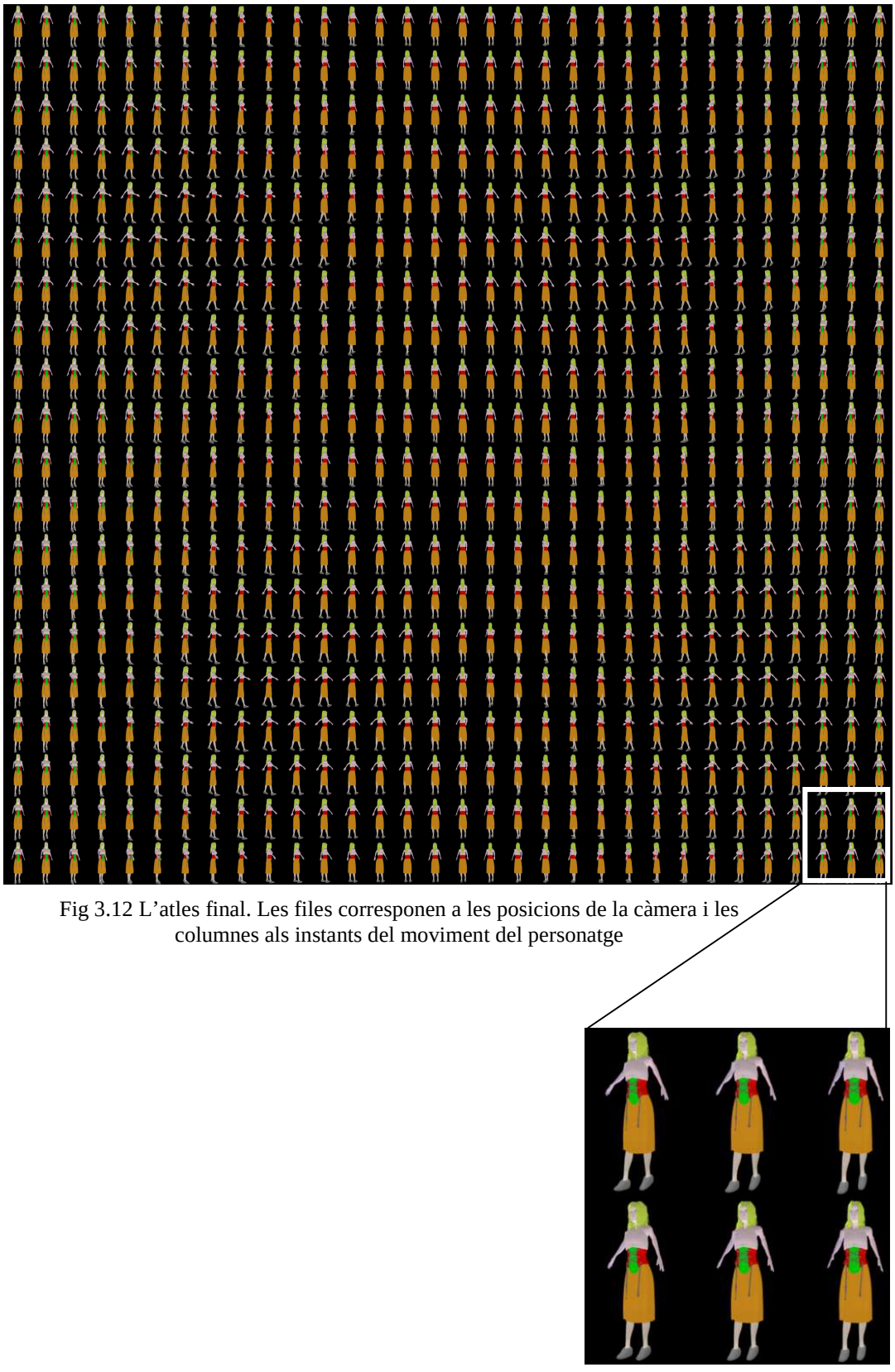


Fig 3.12 L'atles final. Les files corresponen a les posicions de la càmera i les columnes als instants del moviment del personatge

Les columnes representen les diferents fotografies que s'han fet al moviment del personatge. Les files veiem que representen el moviment del personatge en un cert instant de temps.

Aquest atles és el que ens permetrà aconseguir el moviment realista del personatge dins l'escena, associant al *billboard* corresponent la imatge del moviment en cada instant i en cada angle, simulant el seu moviment.

3.2- El mòdul de la creació de l'escenari

Com el nom indica és l'encarregat de preparar i generar l'escenari on els personatges es mouran. Bàsicament el que fa és crear una escena d'una mida concreta, en funció del número d'edificis que hi hauran. Un cop decidit quins elements s'hi posaran, es genera el pla on aniran els elements i els personatges. Per últim es genera un cel per tal de donar més realisme a l'escena. Els edificis escollits són els que porta l'OGRE per defecte, ja que donen bons resultats i són prou realistes per tal d'afegir-los a la nostra aplicació.

3.2.1- La classe Edifici

La classe Edifici (figura 3.13) és la classe encarregada de representar els edificis que instanciem a la nostra escena.

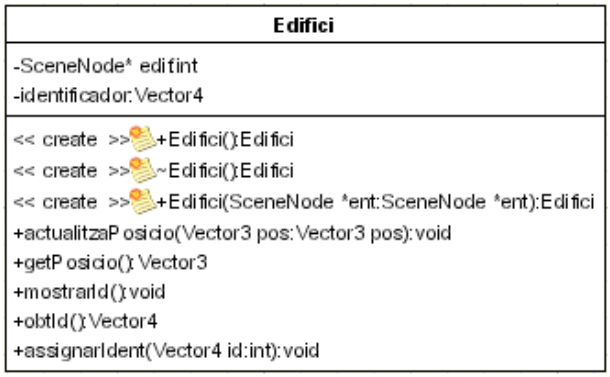


Fig 3.13- La classe Edifici

Té només dos atributs:

SceneNode* edif: aquest atribut guarda un punter a un SceneNode, que serà l'encarregat d'emmagatzemar el *mesh* de l'edifici.

Vector4 identificador: aquest atribut té sentit únicament en el mòdul de control de col·lisions i de preprocés. Podem dir que guarda un identificador de quatre valors, que serà l'encarregat d'identificar unívocament a l'edifici.

En ser una classe molt simple els mètodes que té són els típics per assignar i actualitzar els valors dels seus atributs.

3.2.2- La classe LlistaEdifici

Aquesta classe (figura 3.14) és l'encarregada de guardar en una estructura els edificis que hi han a l'escena per tal de portar un millor control i d'agilitzar la feina de gestionar-los.

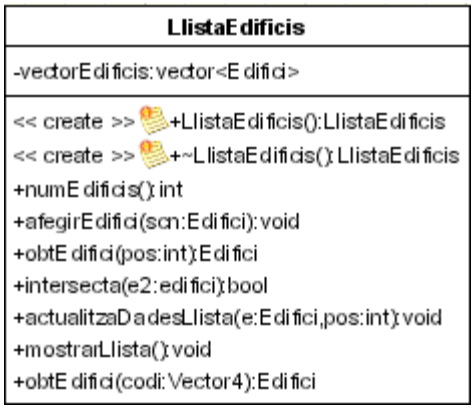


Fig 3.14- La classe LlistaEdificis

Només té un atribut:

Vector<Edifici> vectorEdificis: és un vector d'objectes Edifici.

A diferència de l'altra classe, aquesta posseeix uns mètodes bàsics pel bon funcionament del mòdul, d'entre els que destaquem:

bool interseca (Edifici e): aquest mètode és l'encarregat d'esbrinar si l'edifici passat com a paràmetre interseca (a l'escena direm que xoca) amb els edificis de la llista. Donem un cop d'ull al codi:

```
bool trobat=false
SceneNode s1
SceneNode s2 = e.obtenirSceneNode()
Enter i=0
Edifici aux
BoundingBox bb1,bb2
Bb2=s2->->_getWorldAABB()

while(i<obtenirNumEdificis() && !trobat)
{
      e = obtEdifici(i);
      sc1 = e.edif;

      ax = sc1->_getWorldAABB();

      if(ax.intersects(ax2))
            trob = true;
      else
            i++;
}
```

```
return trobat
```

Com podem observar, el que fa bàsicament el mètode és obtenir cada *BoundingBox* dels objectes de tipus Edifici i mirar si intersequen amb la *BoundingBox* de l'edifici passat per paràmetre.

Aquest mètode ens serà útil quan generem l'escenari, ja que així evitarem que hi hagi edificis que estiguin xocant entre ells.

3.2.3- La classe CoordenadesTextura

Aquesta classe (figura 3.15) ens serveix per fer més àgil el tractament de les coordenades de textura de l'atles. Bàsicament guardarà quatre coordenades que correspondran amb una imatge de l'atles.

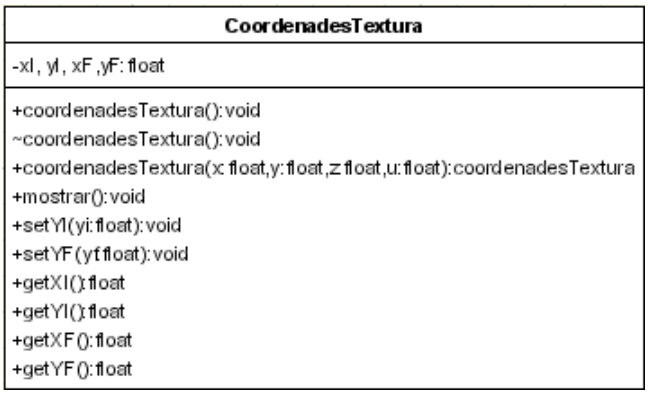


Fig 3.15- La classe CoordenadesTextura

Té quatre atributs:

***float* xI,xF,yI,yF:** cadascun d'ells guarda un valor que correspondrà a la primera i l'última coordenada en x i la primera i l'última en y. Amb això tenim les quatre coordenades que ens indicaran quina part de l'atles de textura seleccionem.

Igualment com la classe Edifici, els mètodes són els típics per actualitzar els valors dels atributs.

Hem de dir també que tots els personatges comencen tenint com a posició yInicial el valor 0 i la yFinal el valor "alçada de la fotografia". Les coordenades de l'atles van de 0 a 1 (en l'eix Y, el límit de dalt és el 0 i el de baix és l'1. En l'eix X, el límit per l'esquerre és el 0 i el de la dreta és l'1.), per tant, les coordenades de textura han de ser sempre en relació a aquest sistema de coordenades.

3.2.4- La classe Personatge

Una de les classes més importants del GdM (figura 3.16).

Com el nom indica és l'encarregada de representar els personatges que formaran la nostra multitud.

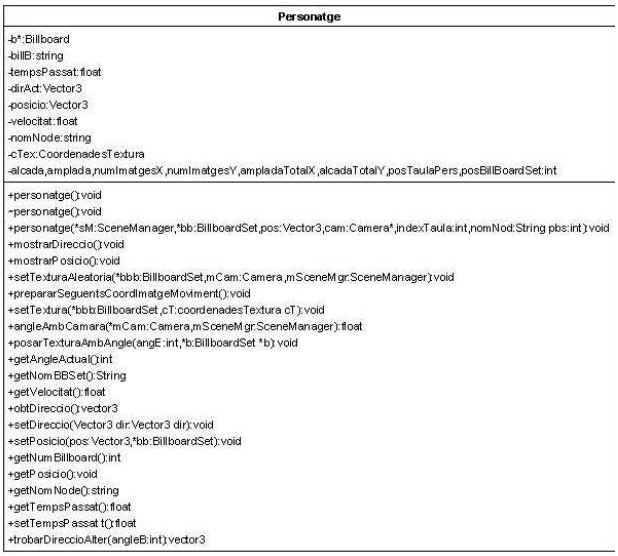


Fig 3.16- La classe Personatge

Vegem els seus atributs:

BillBoard* b: aquest atribut guarda un punter al *billBoard* usat per enganxar la imatge del personatge.

String billB: guarda el nom del *BillboardSet* al qual pertany el *billBoard* assignat al personatge.

Vector3 dirAct: és el vector que guarda la direcció actual que porta el personatge.

Vector3 posició: és el vector que guarda les coordenades on es troba el personatge dins l'escenari.

Float tempsPassat: aquest atribut guarda el temps que ha passat des que s'ha actualitzat la imatge del moviment per última vegada. Servirà per saber quan hem de posar la següent imatge del moviment.

Float velocitat: la velocitat que porta el personatge.

String nomNode: el nom del node al qual està enganxat el personatge. Útil per fer les actualitzacions de posició.

CoordenadesTextura cTex: ens servirà per controlar d'una forma més amena quina part de l'atles de textura agafar a l'hora d'enganxar la imatge del moviment corresponent. És del tipus CoordenadesTextura.

Real angle: guarda en tot moment l'angle que forma el vector dirActual amb el vector direcció de la càmera. Útil per saber quina imatge hem d'agafar de l'atles.

Int posTaulaPers, posBillBoardSet: ens diran quina posició ocupen dins la taula de personatges i quina posició ocupen dins el BillboardSet respectivament.

Alçada,amplada,numImatgesX,numImatgesY,ampladaTotalX,ampladaTotalY: tots són de tipus enter. Tenen a veure amb la informació de l'atles de textura. Alçada i amplada guarden la mida en píxels de les fotos obtingudes al personatge. NumImatgesX guarda el número de fotos que s'han fet al personatge per cada moviment. Recordem que en el nostre cas eren 32. NumImatgesY guarda el número de posicions diferents que adoptava el personatge. En el nostre cas eren 20. AmpladaTotalX i AmpladaTotalY guarden l'amplada i alçada de l'atles de textures respectivament.

Un cop presentats els atributs que formen aquesta classe aprofundim en els mètodes més importants que la formen:

angleAmbCamara(Camera* mCam,SceneManager* mSceneMgr): retorna un float. Aquest mètode s'encarrega de calcular l'angle α que formen el vector direcció del personatge P amb el vector direcció de la càmera δ. Servirà per escollir, de totes les imatges de l'atles, la que correspon amb l'angle format per aquests dos elements (figura 3.17).

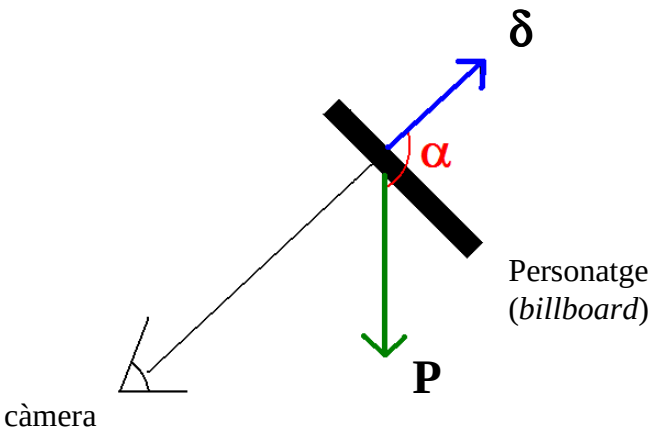


Fig 3.17- Diagrama del càlcul entre la càmera i el personatge

Vegem el que fa el mètode:

```
Vector3 posCam = camera->getPosition();
Vector3 aux2;
Vector3 aux=dirActual;
Vector3 posNoia;

posNoia = posicioActualNoia;
aux2 = (posNoia - posCam) ;
aux2.normalise();
//aux2 és el vector que va des de la càmera a la noia

float pScalar = aux.dotProduct(aux2) ;
//tenim el producte escalar

float ang = pScalar;//és pScalar sol perquè m1*m2 = 1.
//amb la fórmula obtenim l'angle en radiants entre els dos
//vectors
float angleRad = acos(ang);

Real pi = Ogre::Math::PI;
Real angleFi = (angleRad*180)/pi;
angle = angleFi;

//donem a l'atribut angle el valor en graus de l'angle entre
//la càmera i el personatge

Vector3 vPerp = dirAct.crossProduct(aux2) ;
vPerp.normalise();

//amb el producte vectorial esbrinem com estan col·locats els
//vectors

if(vPerp.y<0)
      angle = 180+angleFi;
else if(vPerp.y>0)
      angle=180-angleFi;
else//(vPerp.y=0)
{
      Vector3 res;
      res=aux-aux2;
      if(res= Vector3::ZERO)
            angle=180;
      else
            angle=0;
}
return angle;
```

posarTexturaAmbAngle(int angE,BillboardSet *b): hem dit que és molt important aquest angle trobat ja que és el responsable de indicar-nos quina imatge hem d'escollir de l'atles. Recordem que les columnes de l'atles estan graduades, en el nostre cas, $\Delta\alpha$=11.25 graus cadascuna. Això vol dir que l'angle entre el personatge i la càmera en el moment de la fotografia varia en increments de $\Delta\alpha$. Si aconseguim saber l'angle que formen el personatge i la càmera actual

podrem escollir, d'entre les columnes de l'atles, la que s'apropa més a l'angle obtingut. Per exemple, sabent que les columnes són, en el nostre cas, 0,11.25,22.5,33.75...348.75. Si obtenim que l'angle entre la càmera i el personatge és de α=20 graus, vol dir que hem d'escollir la columna de l'atles que té les fotografies quan el personatge forma α=22.5 graus amb la càmera. La forma amb que ho escollim és la següent:

Posició a escollir de l'atles = $\lceil(\alpha/\Delta\alpha)$, on $\lceil$ vol dir arrodoniment a l'alça.

Hem de comentar que per aconseguir un moviment sense que s'apreciïn els canvis d'angle necessitaríem moltes més imatges del personatge.

El codi del mètode segueix la següent forma:

```
//angE prové del mètode angleAmbCàmera
Enter posAtles = arrodonirAlça((angle/11.25)+0.5f);

if(posAtles = 32)
      posAtles = 0;//ens fa tota la volta...360º

float xI = (posAtles*amplada)/ampladaTotalX;

float yI= cTex.getYI();//la que ja teníem

float xF =(posAtles*amplada+amplada)/ampladaTotalX;

float yF= cTex.getYF();//la que ja teniem

cTex = coordenadesTextura(xI,yI,xF,yF) ;

b->getBillboard(posBillBoardSet)->setTexcoordRect(xI,yI,xF,yF)
//obtenim el billboard que pertany al personatge dins el
//billboardSet i li proporcionem les noves coordenades de
//textura
```

prepararSeguentsCoordImatgeMoviment(): aquesta funció no retorna cap valor, ja que treballa directament amb l'atribut que guarda les coordenades de textura. El seu objectiu és canviar la imatge que mostra el *billboard* per la següent imatge que toca. Podríem dir que s'encarrega de canviar les imatges del personatge per fer semblar que té moviment. Només modifica les coordenades en Y, ja que l'atles guarda en les columnes la seqüència del moviment del personatge:

```
//obtenim les coordenades en Y actuals
float yIAct=cTex.obtYI();
float yFAct=cTex.obtYF();

//calculem les noves coordenades en Y
float yI2 = yIAct + alcada/alcadaTotalY;
```

```
float yF2 = yFAct + alcada/alcadaTotalY;
//bàsicament el que hem fet ha estat baixar una fotografia en
//la columna de l'atles

if(yI2>1)
      yI2= 0;

if(yF2>1)
      yI2 = 0;
      yF2= alcada/alcadaTotalY;

//aquests condicionals volen dir que ens hem passat de rang,
//per tant tornem a començar des de dalt de la columna

cTex.setYI(yI2) ;
cTex.setYF(yF2) ;
//les actualitzem..i ja està
```

Vector3 trobarDireccioAlter(int angleB): aquesta funció, com hem comentat anteriorment, només pren sentit dins el mòdul de detecció de col·lisions. La seva funció és, donat l'angleB i la direcció actual, trobar una altra direcció obtinguda aleatòriament (fa que el personatge canviï de direcció per evitar una col·lisió). Vegem, però, el codi ja que no entranya dificultat alguna:

```
int angleFi;
angle = angle + angleB;

while(angle>360 || angle<-360)
{
      angle = random(-360,360);
}
//fem que l'angle estigui dins els limits

float angleRad = (angle*Ogre::Math::PI)/180.f;
//el passem a radiants

Matrix3 m3 (cos (angleRad) , 0 , -  sin(angleRad) , 0, 1, 0,
sin (angleRad), 0, cos (angleRad));
Vector3 aux = m3*dirAct;
//aquesta és una funció que mitjançant la matriu de rotació
//obté un nou vector rotat angle graus.
aux.normalise()
return aux;
```

3.2.5- La classe LlistaPersonatge

Com passa amb la classe llistaEdifici, la classe LlistaPersonatge (figura 3.18) simplement serveix per facilitar-nos la feina de gestionar els personatges que formaran la nostra multitud.

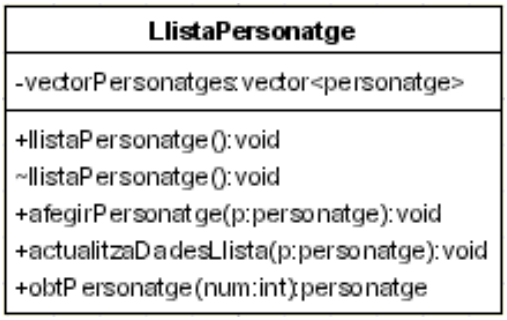


Fig 3.18- La classe LlistaPersonatge

Només té un atribut:

vector<personatge> vectorPersonatges: guarda un vector d'objectes Personatge.

Té els mètodes corresponents per afegir i actualitzar els valors de la llista de personatges.

3.2.6- La classe Escenari

La classe Escenari (figura3.19) és l'encarregada d'incialitzar tots els personatges i edifics de l'escena. També és l'encarregada de gestionar els objectes d'aquestes classes. Podríem dir que és l'encarregada de preparar els elements de l'escena per tal de començar a generar el moviment.

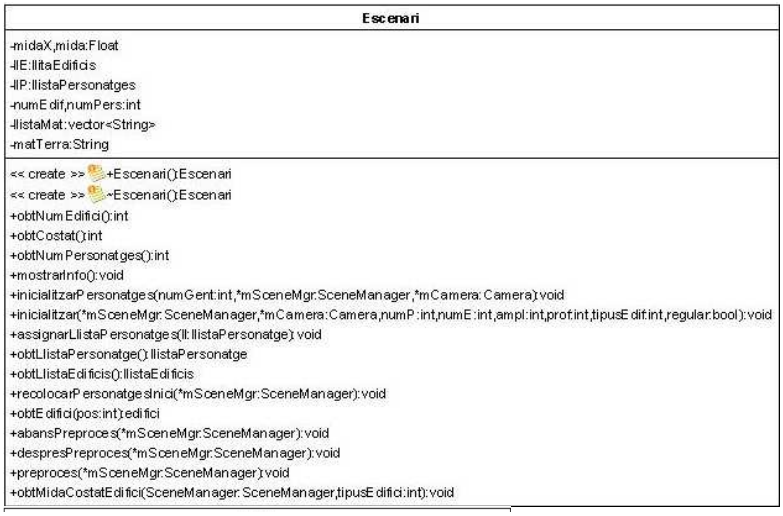


Fig 3.19- La classe Escenari

Té els següents atributs:

Real midaX, midaZ: aquests atributs de tipus Real són els encarregats d'emmagatzemar la mida de l'escenari. Hem de dir que els escenaris del GdM usualment seran quadrats, per simplificar els càlculs.

LlistaEdificis llE: és l'atribut que emmagatzema la llista de tots els edificis que composaran la nostra escena. És del tipus LlistaEdificis.

LlistaPersonatge llP: és l'atribut que emmagatzema la llista de tots els personatges que composaran la nostra escena. És del tipus LlistaPersonatge.

int numEdif: guarda el número d'edificis que hi han a l'escena.

int numPers: guarda el número de personatges que es representaran a l'escena.

vector<String> llistaMat: aquest atribut és un vector que emmagatzema objectes de tipus cadena de caràcters. La seva funció és la de saber en tot moment quin és el nom de la textura usada per cada edifici de l'escena. Veurem la seva utilitat en el mòdul de preprocés.

String matTerra: guarda unicament el nom de la textura usada pel pla de la nostra escena. Com l'anterior atribut, s'usa només al mòdul de preprocés.

Aquests dos últims atributs prenen importància en el mòdul de preprocés. A grans trets podem dir que serveixen per no perdre el valor de les textures dels edficis per tal de poder-les modificar.

Com a mètodes principals de la classe destaquem:

InicialitzarPersonatges(int nGent,SceneManager* mSceneMgr, Camera * mCamera): és l'encarregat d'inicialitzar tants personatges com *nGent* diu (*nGent* vindrà donat per l'usuari). El que fa el mètode primer de tot és crear el *BillboardSet*, que serà del que penjaran la resta de *Billboards* associats a cada personatge. Cal assignar-li també l'atles de textura per tal que tots els *billboard*s hi tinguin accés. Un cop fet això només ens queda instanciar els objectes personatge amb una posició, direcció i moviment incial aleatori, afegint-los a la llista de personatges.

```
BillboardSet*b=mSceneMgr->createBillboardSet("BBSet",numGent);
//creem el billboardSet

b->setBillboardType(BBT_ORIENTED_COMMON);
b->setCommonDirection(Vector3::UNIT_Y);
//així només roten en l'eix Y

b->setMaterialName("Examples/Atles");
//li assignem l'atles
Vector3 pos;
personatge p;
for(i=0;i<nGent;i++)
{
   pos = Vector3(random(-midaX/2,midaX/2),0,random(-midaZ/2,midaZ/2));
   //posició aleatoria. Es divideix per 2 perquè l'escenari està
   //centrat al (0,0,0). La coordenada Y és 0 perquè els personatges
   //no floten

   p = personatge(mSceneMgr,b,pos,mCamera,i,nomNode,i);
```

```
    //creem el personatge

    movInicial = random(0,20);
    for(j=0;j<movInicial;j++)
          p.prepararSeguentsCoordImatgeMoviment();

    //generem un número entre 0 i 20. Així fem que els personatges
    //comencin amb estats diferents del moviment de caminar. Sinó
    //anirien al mateix ritme i al mateix moment

    llisPers.afegirPersonatge(p);
    //afegim el personatge
}
```

recolocarPersonatgesInici(SceneManager *mSceneMgr): és un mètode molt important per tal d'obtenir una bona inicialització dels personatges de l'escena. El que fa aquest mètode és evitar que els personatges col·locats a la incialització estiguin dins d'algun edifici. Si això passés donaria problemes al mòdul de control de col·lisions, així com la reducció del realisme. Vegem-ne el codi:

```
int i, j;
Vector3 pos;
SceneNode *nAux;
AxisAlignedBox ax;
personatge pAux;
BillboardSet *b;

bool colisiona;
for(i=0;i<llP.vectorPersonatges.size();i++)//x tot personatge
{
      pAux=llP.obtPersonatge(i);
      b=mSceneMgr->getBillboardSet(pAux.getNomBBSet());
      //obtenim el seu billboard
      j=0;
      colisiona  = false;
      while(j<llE.vectorEdificis.size())//x tot edifici
      {
            nAux =  obtEdifici(j).edif;
            nAux->_update(true,false);
            ax = nAux->_getWorldAABB();
            //obtenim la bounding box de l'edifici
            while(ax.intersects(pAux.getPosicio()))
            {
                  //si intersecta...
                  colisiona=true;
                  pos   =   Vector3((Real)(Ogre::Math::RangeRandom(-
                  1*midaX/2,midaX/2)),0,(Real)(Ogre::Math::RangeRan
                  dom(-1*midaZ/2,midaZ/2)));
                  pAux.setPosicio(pos,b);
                  //trobem la posició alternativa
            }
            llP.actualitzaDadesLlista(pAux);
            if(!colisiona)
                  j++;
            else
```

```
            {
                  j=0;
                  colisiona=false;
            }
      }
}
```

inicialitzar(SceneManager *mSceneMgr, Càmera *mCamera,int numP,int numE,int ampl, int prof,int tipusEdif,bool regular): aquest és el mètode encarregat de cridar i parametritzar les anteriors classes per la inicialització final de l'escenari i els personatges. Degut a la seva extensió l'explicació del codi la farem per parts i només explicarem aquells trossos que tinguin un interès especial. Els paràmetres que porta la crida corresponen a l'sceneManager, a la càmera, al número de persones, al número d'edificis, a l'amplada i profunditat de l'escenari, al tipus d'edificis a representar i a la distribució adoptada. Arribats a aquest punt hem de comentar que l'usuari pot escollir dos tipus d'edificis: un serà el que porta l'OGRE per defecte, l'altre serà un cub. Hem escollit posar-hi més elements per donar més joc a l'aplicació. En quant a la distribució pot ser de dos tipus: o aleatòria o regular. Si és regular el GdM col·locarà els elements per files, altrament els col·locarà de forma aleatòria.

```
SceneNode *mNode;

mSceneMgr->destroyAllLights();
//les traiem pel mòdul de preprocés
numEdif=numE;
numPers=numP;

mSceneMgr->setSkyBox(true, "StormySkyBox");
//el cel

Plane plane;
plane.normal = Vector3::UNIT_Y;
plane.d = 100;
Ogre::MeshManager::getSingleton().crearPlà("Myplane",midaX,midaZ)
Entity* pPlaneEnt = mSceneMgr->crearEntitat("terra","Myplane");
pPlaneEnt->setMaterialName("Rockwall");
//pla creat

matTerra=pPlaneEnt->getMaterialName();
//guardem l'atribut per saber quin material tenia el pla creat

pPlaneEnt->setCastShadows(fals) ;
//les traiem pel mòdul de preprocés

mSceneMgr->getRootSceneNode()->createChildSceneNode ;
(Vector3(0,35,0))-> attachObject(pPlaneEnt)

Entity* pEnt;
```

En aquest punt ja tenim el que serà el terra de l'escenari i el cel. Anem a veure el codi per crear els edificis i els personatges.

```
Int tol = 400; //així els edificis estan tots a dins
String aux;
for(j=0;j<numE;j++)
{
      //per cada edifici
      mNodemSceneMgr->getRootSceneNode()->createChildSceneNode
(string(j)) ;
      if(tipusEdif==1)//cases tudor
      {
            pEnt = mSceneMgr->createEntity(sc.toString(j),
            "tudorhouse.mesh");
             mNode->setPosition(random((-midaX/2)+tol,(midaX/2)-
                  tol),470,random((-midaZ/2)+tol,(midaZ/2)-tol)) ;
            mNode->attachObject (pEnt) ;
      }
      else if(tipusEdif==2)//cubs
      {
            pEnt=mSceneMgr-
               >crearEntity(sc.toString(j),"cube.mesh");
            pEnt->setMaterialName("escac");
             mNode->setPosition(random((-midaX/2)+tol,(midaX/2)-
                  tol), 0,random((-midaZ/2)+tol,(midaZ/2)-tol)) ;
            mNode->attachObject (pEnt) ;
      }

      aux=pEnt->getMaterialName();
      llistaMat.push_back(aux) ;
      //guardem el nom del material de l'edifici a la taula //de
      materials

      Edifici e(mNode) ;//creem l'objecte Edifici
      while(llE.interseca(e))
      {
            //comprobem que no s'hagi col·locat xocant amb un altre
            //edifici
            if(tipusEdif==1)
                 e.actualitzaPosicio(Vector3(random((-
                        midaX/2)+tol,(midaX/2)-tol),470,random((-
                        midaZ/2)+tol,(midaZ/2)-tol)));

            altrament if(tipusEdif==2)
                 e.actualitzaPosicio(Vector3(random((-
                        midaX/2)+tol,(midaX/2)-tol), 0,random((-
                        midaZ/2)+tol,(midaZ/2)-tol))) ;
      }

      llE.afegirEdifici(Edifici(e)) ;//l'afegim a la llista dels
      //edificis
}
```

Faríem el mateix en el cas que la disposició no fos aleatoria, simplement canviarien les línies on posem la posició dels edificis.

Ara ja tenim els edificis col·locats i sense que es toquin els uns amb els altres. Hem de dir que s'ha posat una tolerància en el càlcul de les posicions (hi sumem o hi restem 400). Això ho fem per tal d'evitar que quedin edificis amb trossos fora de l'escenari. El que ens queda ara és crear els personatges i inicialitzar-los. Això ho fem amb els dos mètodes explicats anteriorment:

```
inicialitzarPersonatges(numP,mSceneMgr,mCamera);
recolocarPersonatgesInici(mSceneMgr);
```

Un cop cridats ja no ens queda res més per inicialitzar. Per tant, ja tenim l'escenari creat i inicialitzat per començar a generar la multitud. Vegem en la figura 3.20 el diagrama de seqüència del mòdul.

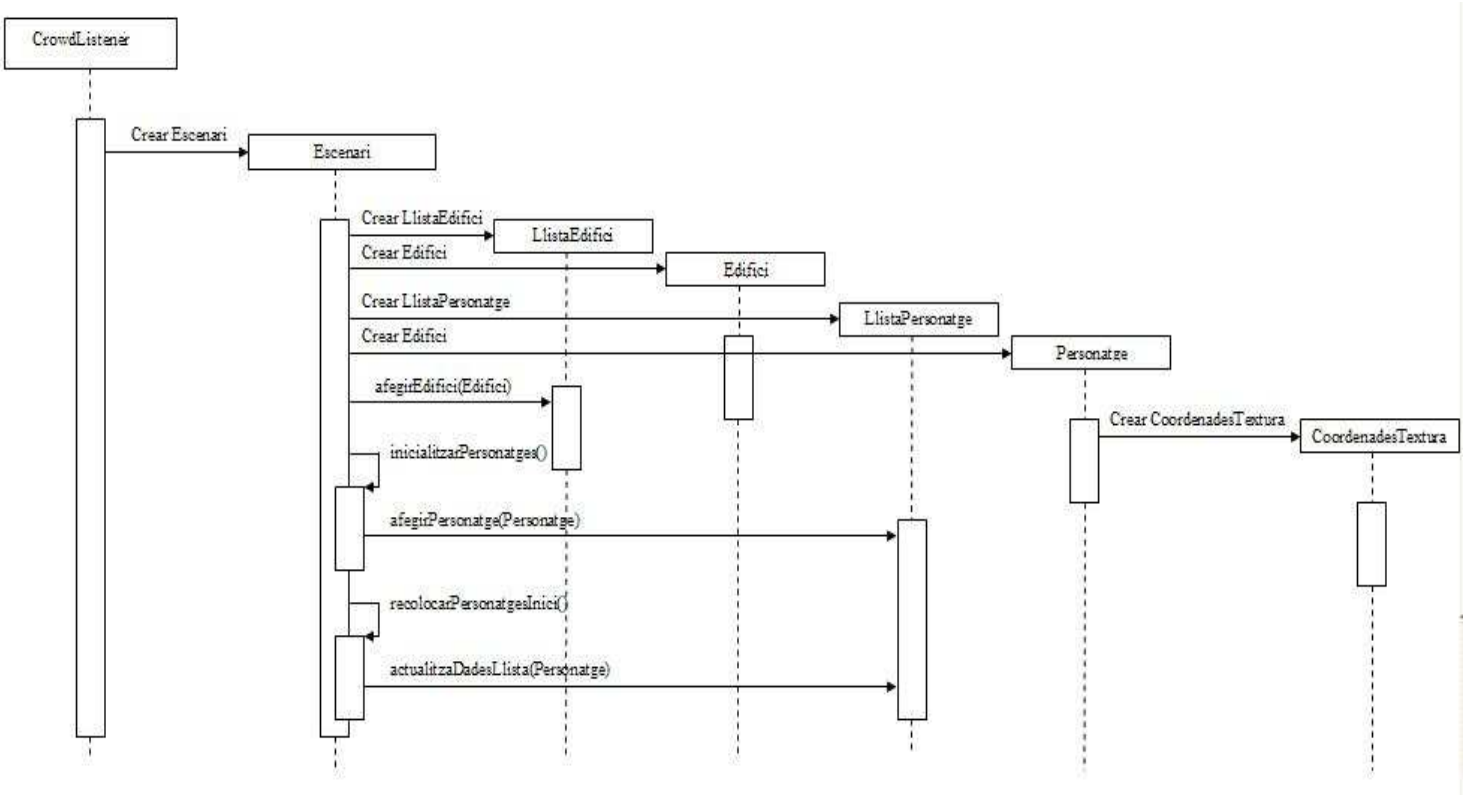


Fig 3.20- Diagrama de seqüència del mòdul de creació de l'escenari

Al digrama de seqüència anterior podem veure el procés de creació de l'Escenari: primer, l'objecte CrowdListener crida el constructor de l'Escenari. Després, i un cop creat aquest objecte, es crea la LlistaEdifici i s'emplena amb objectes Edifici. Es crea també la LlistaPersonatge i s'emplena amb objectes

Personatge, que a la vegada se'ls hi assigna un objecte CoordenadesTextura. Un cop estan creats, se'ls recoloca amb el mètode **recolocarPersonatgesInici**.

3.3- El mòdul de preprocés

Aquest mòdul és l'encarregat de fer un preprocessament a l'escena per tal de preparar-la perquè el mòdul de càlcul de col·lisions la pugui tractar.

L'objectiu d'aquest mòdul és el de preparar l'escena per obtenir de forma fàcil la informació d'aquesta, i emplenar després l'estructura de dades que usarà el mòdul de càlcul de col·lisions. La idea és tenir un mapa de com estan posicionats els edificis. Un cop obtingut aquest mapa, el mòdul de càlcul de col·lisions el farà servir per saber en tot moment quins edificis estan en posició de col·lisió amb els personatges. Expliquem, primer de tot, la idea general que volem implementar en aquest mòdul.

1. El primer que hem de fer és posicionar una càmera al zenit de l'escena per tal s'abarcar tot l'escenari (figura 3.21).

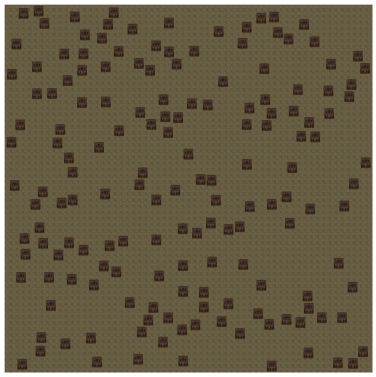


Fig 3.21- Imatge superior de l'escena sense el preprocés realitzat

2. Un cop tinguem la càmera posicionada, hem de canviar el color de cadascun dels edificis de l'escena, per aconseguir que cada edifici tingui un color diferent a un altre. Això ho fem així perquè farem que el color sigui l'identificador unívoc de cada edifici dins del mòdul de càlcul de col·lisions. És aquí on pren sentit l'atribut "identificador" de la classe Edifici. Aquest identificador serà un vector de quatre components on cadascuna d'elles serà el valor RGBA del color donat a l'edifici.

Un cop canviat el color dels edificis hem de canviar de color el pla, donant-li el color negre (0,0,0,1) i posant com a color de fons el blanc (1,1,1,1). La component alfa del color sempre la posem a 1, ja que no volem transparència. Hem d'amagar també els personatges perquè no interfereixin en el procés.

El fet de deixar el pla negre i el fons blanc ens servirà per informació de l'escena, per exemple, on comença i on acaba el pla (per tant l'escenari).

Un cop fet el preprocés la imatge que veuria la càmera seria la de la figura 3.22.

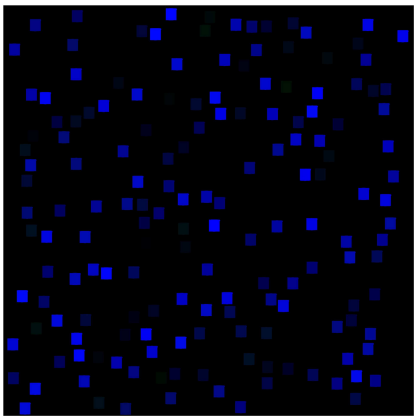


Fig 3.22- Vista superior de l'escena amb el preprocés realitzat

Com es pot comprovar ara els edificis han près un color diferent cadascun. Hem de dir també que el color negre no es dóna a cap edifici, ja que pertany al pla.

3. Un cop preparada l'escena ja es pot fer la fotografia de com ha quedat. Aquesta fotografia serà la que farà servir el mòdul de càlcul de col·lisions per emplenar les seves estructures de dades.
4. Feta la fotografia, hem de procedir a deixar tal com estava l'escena. És aquí on intervenen els atributs "llistaMat" i "matTerra" respectivament. El que hem de fer ara és anar recorrent la llista de materials de matTerra i assignar-los novament a l'edifici corresponent. D'aquesta manera tornen a quedar amb la textura original. Fem el mateix amb el pla.
5. Finalment, l'escena torna a quedar com al principi (figura 3.23):

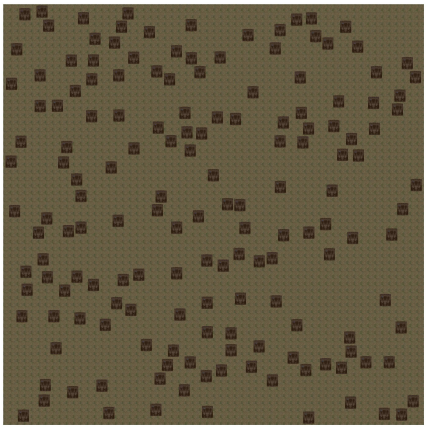


Fig 3.23- L'escena original

Ara que ja sabem el que hem de fer, anem a veure com ho hem fet i quines classes intervenen en el procés.

3.3.1- La classe Escenari (mòdul de preprocés)

Com la classe ja ha estat explicada en el punt 3.2.6, aquí només explicarem els mètodes que pertanyen al mòdul de preprocés.

void abansPreproces(SceneManager *mSceneMgr): aquest mètode el que fa és amagar tots els personatges, treure el cel i posar el terra de color negre.

```
BillboardSet *b = mSceneMgr->getBillboardSet("nomBBSet");
b->setVisible(false);
mSceneMgr->setSkyBox(false);
Entity *e = mSceneMgr->getEntity("terra");
e->setMaterialName("Examples/negre");
```

void despresPreproces(SceneManager *mSceneMgr): aquest mètode s'encarrega de restablir l'escena tal com estava: retorna el material original als edificis, recupera els personatges, recupera el cel i el terra.

```
Entity *e = mSceneMgr->getEntity("terra");
e->setMaterialName(matTerra);
BillboardSet *b = mSceneMgr->getBillboardSet("nomBBSet");
b->setVisible(cert);

mSceneMgr->setSkyBox(true);

Entity *e2;
for(i=0;i<llE.numEdificis();i++)
{
```

```
            //recorrem la llista d'edificis i els assignem el
            //material original
            e2 = mSceneMgr->getEntity(string(i));
            e2->setVisible(true)
            e2->setMaterialName(llistaMat[i]);
      }
```

void canviarColors(SceneManager *mSceneMgr): aquest és el mètode que s'encarrega de fer el canvi de color de cada edifici, i per tant, d'assignar un codi diferent a cada edifici.

```
Entity* pEnt
#define PARAMETRE_COLOR 1

float r,g,b;
Vector4 rgba;
int numEdif=0;
```

intervalCodiEdif és una variable global definida a la classe que actualment té el valor 5. Serveix per controlar l'increment dels números que donen el valor del color. Augmenten de 5 en 5. Si es deixés a 1, en certs ordinadors ens dóna problemes, ja que considera que, per exemple, el color (0,0,1,1) és el mateix que el (0,0,2,1). D'aquesta manera, la diferència entre els colors és més que suficient per tal que la tarja gràfica els pugui diferenciar.

```
int numEdif=0;
bool fi=false;
r=0;
while(r<256 && !fi)
{
 g=0;
       while(g<256 && !fi)
       {
             b=intervalCodifEdif;
             while(b<256 && !fi)
             {
                   pEnt = mSceneMgr-
                   >getEntity(sc.toString(numEdif));

                   rgba=
                   Vector4((float)(r/255.f),(float)(g/255.f),(float)
                   (b/255.f),(float)1.f);

                   pEnt->getSubEntity(0)-
                   >setCustomParameter(COLOR_MODULATE_PARAM_INDEX,rg
                   ba );


                  pEnt->setMaterialName("foto");

                  Edifici ef=llE.obtEdifici(numEdif);
                  ef.assignarIdent(rgba);
                  llE.actualitzaDadesLlista(ef,numEdif);
                  numEdif++;
```

```
                        if(numEdif==llE.numEdificis())
                              fi=true;
                        b=b+intervalCodifEdif;
                  }
                  g=g+intervalCodifEdif;
            }
            r=r+intervalCodifEdif;
      }
```

Vegem ara l'aspecte que té el *shader* creat per modificar el color dels edificis. Hem de dir que només s'ha creat un *Pixel shader* (o *fragment program*), ja que el vertex *shader* no és necessari perquè no estem modificant la geometria, sinó només el color dels píxels. Vam decidir implementar part del codi en *shader* ja que ens simplificava molt la feina del canvi de color dels edificis.

```
//Programa principal
// Rep un paràmetre uniform que és el color
// Com a sortida té un color. Com que la semántica és COLOR, vol dir que
// el que s'està generant és el color del fragment
void main_tex_mod_col_fp(out float4 color: COLOR,
      uniform float4 colorModulate)
{
      // Assignem el color del fragment
      color = colorModulate;
}
//"color" és el resultat de sortida. ColorModulate vé donat des de fora
//també, des de la linia de codi pEnt->
//setCustomParameter(PARAMETRE_COLOR,rgba).
//ColorModulate pren el valor de RGBA. Al declarar colorModulate com a
//uniform, estem dient que pren el valor d'una font externa.


//Indica que el fitxer conté un pixel shader
fragment_program TextureModColor_PS cg
{
      // El codi font del shader està al fitxer GameObjStandard.cg
      // amb format cg
      source GameObjStandard.cg //crida al main


      // Different entry point for pixel shader
      // El programa principal al fitxer té el nom de main_tex_mod_col_fp
      entry_point main_tex_mod_col_fp

      // El perfil de compilació és Píxel Shader 1.1
      profiles ps_1_1

      // No hi ha paràmetres per defecte
      default_params
      {
      }
}

//material que li assignem
material foto
{
```

```
	// La tècnica que s'executa té un sol pas
	technique
	{
		pass
		{
			// El fragment program que s'executa es
			// diu TextureModColor_PS
			fragment_program_ref TextureModColor_PS
			{
				param_named_auto colorModulate custom 1
				//li estem dient que agafi el paràmetre passat
				//(el color RGBA del codi)
			}
		}
	}
}
```

Un cop tenim el color canviat ja podem fer la foto. D'aquesta tasca se n'encarregarà un altre mètode de la classe nucli del GdM, que més endavant explicarem.

3.3.2- La classe CrowdListener (mòdul de preprocés)

Ara per ara no cal conèixer a fons la classe CrowdListener per entendre com fem la fotografia de l'escenari. A grans trets podem dir que la classe CrowdListener serà la que s'encarregarà de fer moure la multitud i controlar els events que hi tinguin lloc. Podem avançar que la classe CrowdListener és el nucli del nostre GdM. Quan parlem del mòdul de la generació de multituds aprofundirem més en aquesta important classe. Anem doncs a veure com es fa la fotografia amb el següent mètode:

void ferFoto(Escenari e): aquest mètode és l'encarregat de disparar la fotografia i guardar la informació per tal de fer-la accessible al mòdul de càlcul de col·lisions. El que fa és posar la càmera a l'altura necessària per abastar tot l'escenari i fer la fotografia. Entenem fer la fotografia com crear un altre objecte on visualitzar el que veu la càmera (d'aquesta manera no toquem el que fem servir normalment), fer una captura del que està veient aquesta càmera i guardar-ho en una taula de píxels (veure apartat 2.3.1.7). Aquesta taula és la que farà servir el mòdul de càlcul de col·lisions per emplenar les seves estructures de dades. Vegem els atributs de la classe CrowdListener que fa servir:

TexturePtr crowdTex: un punter a un objecte TexturePtr.

RenderTexture *crowdRendTex: punter a un RenderTarget.

PixelBox pixelBox: un objecte PixelBox que descriu una imatge a memòria.

HardwarePixelBufferSharedPtr pixelBuffer: un punter que serveix per accedir a un objecte PixelBox.

float* pDest: la taula on guardarem la fotografia.

Real textureResolution: ens indica la resolució que tindrà la nostra fotografia.

Un cop presentats els atributs vegem el mètode:

```
textureResolution= 3200;
//la resolució de la fotografia. En aquest cas és una fotografia
//quadrada, de 3200x3200


crowdTex=TextureManager::getSingleton().createManual("texFoto",

ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME,TEX_TYPE_2D,

textureResolution,textureResolution,0,PF_FLOAT32_RGBA,TU_RENDERTARG
ET);


crowdRendTex = crowdTex->getBuffer()->getRenderTarget();
rowdRendTex->setAutoUpdated(false);//evitem que s'actualitzi
//automàticament

Camera* camPrep = mSceneMgr->createCamera("camPrep");//creem la
//nova càmera

camPrep->setFixedYawAxis(false);//evitem que pugui rotar
camPrep->setProjectionType(PT_ORTHOGRAPHIC);//la posem en projecció
//otogràfica
enter posCam=esc.obtCostat()*1.5
//aquest valor surt de multiplicar el costat de l'escenari per una
//constant, en aquest cas 1.5. Ho fem així per trobar una altura
//que sigui proporcional a la mida de l'escenari per tal que es
//vegi tot

camPrep->setFOVy(Radian(45));//l'angle d'obertura de la //càmera
camPrep->setPosition(0,posCam,0);//la posem a l'altura trobada
camPrep->lookAt(0,0,0);//fem que miri a l'origen (cap a terra)

camPrep->setNearClipDistance(posCam-(posCam*.30))
camPrep->setAspectRatio(1);//perquè no es deformi la imatge

Ogre::Viewport* v =crowdRendTex->addViewport(camPrep) ;
v->setClearEveryFrame(true) ;
v->setBackgroundColour(Ogre::ColourValue::White) ;
v->setOverlaysEnabled (false) ;

pixelBuffer = crowdTex->obtenirBufer();
pDest = new float[textureResolution*textureResolution*4] ;
//creem una taula de floats que guardarà els valors de cada
//component del píxel
```

```
for(j=0;j<textureResolution*textureResolution*4;j=j+4)
{
      pDest[j]=0.0;
      pDest[j+1]=0.0;
      pDest[j+2]=0.0;
      pDest[j+3]=0.0;
}
//ja l'hem inicialitzat

pixelBox=PixelBox
(textureResolution,textureResolution,1,PF_FLOAT32_RGBA,pDest) ;

pixelBuffer->blitFromMemory(pixelBox) ;
```

En la figura 3.24 podem observar el diagrama de seqüència del mòdul.

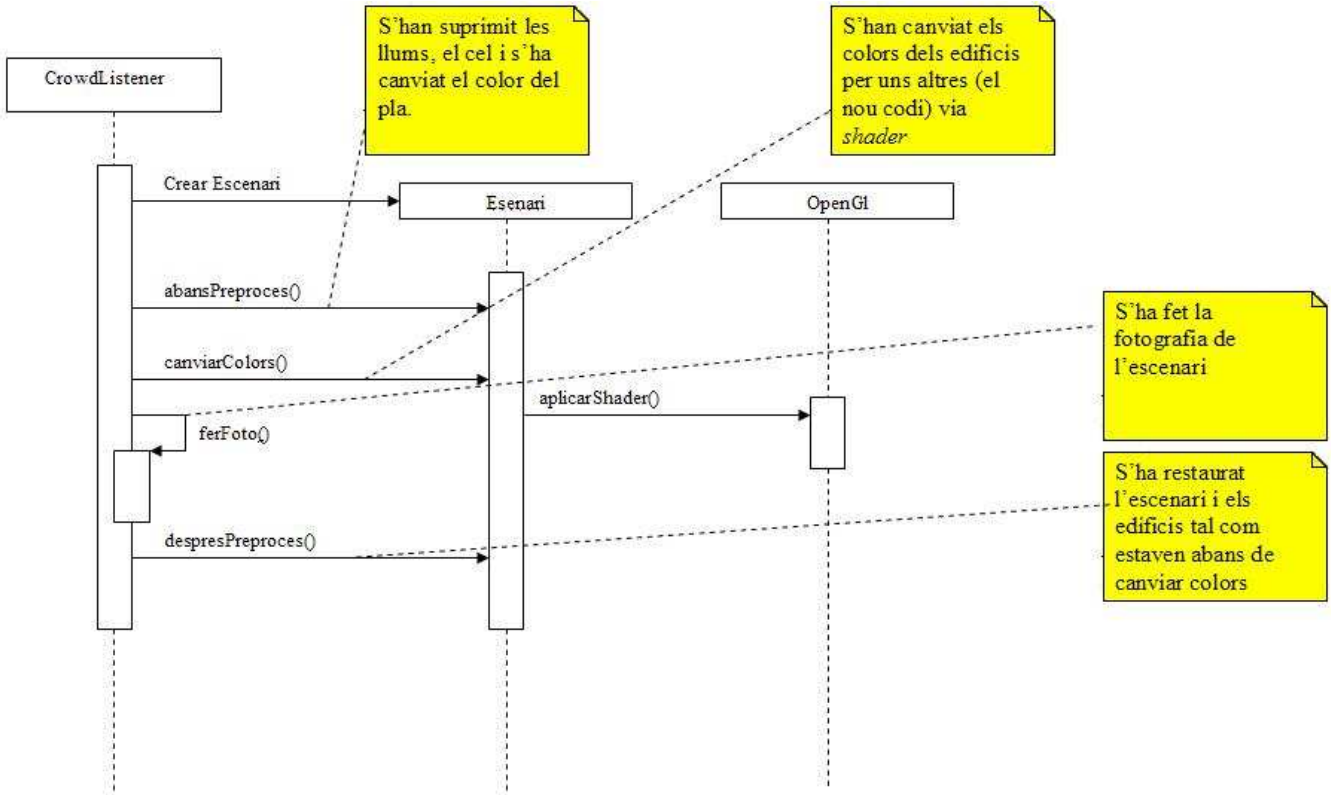


Fig 3.24- L'escena original

El diagrama anterior mostra com, un cop creat l'escenari, l'objecte de la classe CrowdListener crida els mètodes abansPreprocés (prepara l'escenari), canviarColors (crida al *shader* corresponent per canviar els colors dels edificis), ferFoto(que fa la foto aèria de l'escenari) per últim el desprésPreprocés (que restaura l'escenari tal com estava inicialment).

3.4- El mòdul de càlcul de col·lisions

Un cop tenim les dades emmagatzemades de com estan els edificis disposats a l'escena és el moment de fer intervenir el mòdul de càlcul de col·lisions.

Aquest mòdul es va idear per dotar als personatges d'una mica d'intel·ligència, i d'aquesta manera fer que el GdM fos una mica més realista.

L'objectiu d'aquest mòdul és posar a la disposició dels personatges una sèrie d'estructures de dades per tal de saber en tot moment amb quins edificis pot haver-hi alguna possibilitat de col·lisió, i si és així, actuar en conseqüència.

Aquest mòdul està basat en la idea de dividir l'escenari en cel·les, guardant cadascuna d'elles informació dels edificis que tenen a dins seu. Abans de començar a explicar les classes que formen aquest mòdul i els seus mètodes, vegem un exemple d'aquesta divisió en cel·les de l'escenari.

3.4.1- Dividint l'escenari en cel·les

L'objectiu d'aquesta divisió és simplificar l'escenari, organitzant millor els edificis que aquest conté i, gràcies a això, optimitzar el cost de calcular les col·lisions. D'entrada, sense aquesta organització, els personatges no saben els edificis que tenen al seu voltant, per tant, s'hauria de recórrer tots els edificis comprovant si hi ha o no col·lisió. La idea d'organitzar-los per cel·les és per aconseguir reduir aquestes cerques, ja que el personatge només haurà de mirar quins edificis hi ha a la seva cel·la.

Si fem aquesta divisió, hem de tenir en compte que ara ja sabrem quins edificis conté cada cel·la, i no haurem de mirar-los tots; cada personatge només mirarà els edificis que hi ha a la cel·la on es troba en aquell moment, per tant, el cost de mirar els edificis serà aproximadament constant:

Cost(càlculCol·lisió)=numPersonatges × numEdificisDinsCela = k

En quant a la mida d'aquestes cel·les hem pensat que, a major nombre d'edificis, menor mida de cel·la, i viceversa. Això ho fem així perquè en la fotografia des de dalt, si hi ha molts edificis aquests es veuran més petits i per tant, necessitarem més resolució per diferenciar les cel·les on pertanyen. En canvi si n'hi ha pocs, es veuran molt grans i, per tant, no té sentit tenir molta resolució per distingir a quina cel·la pertanyen. Amb la fórmula següent obtenim el número de cel·les que hi haurà en total a l'escenari:

NumCel·les=(costatEscenari*40/resolució)²

Aquest 40 és una constant que vam trobar mitjançant proves que fèiem per tal d'obtenir un bon número de cel·les en funció de la mida.

Observem l'exemple de la figura 3.25.

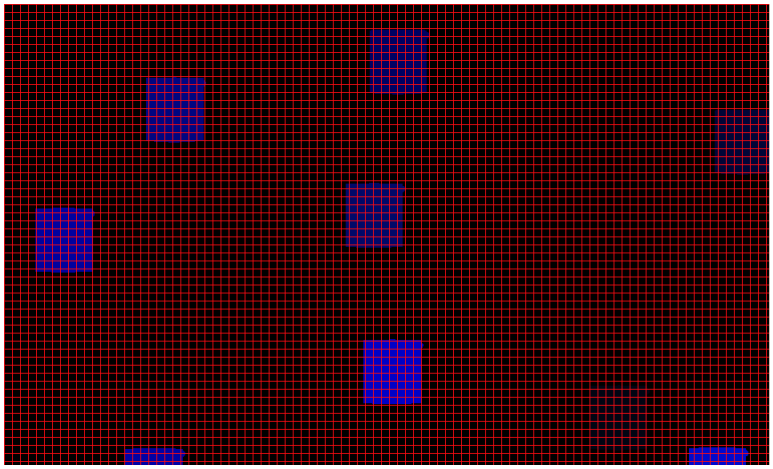


Fig 3.25- Una possible divisió en cel·les de l'escenari

Aquest és un escenari que fa 24000 unitats × 24000 unitats. Seguint la fórmula d'abans, tenim que el número de cel·les és (24000*40/3200)* (24000*40/3200) = 90000 cel·les en total. Per tant a cada fila hi ha 300 cel·les, que surten a uns 10 píxels per cel·la. Observem la figura 3.26 per un escenari petit.

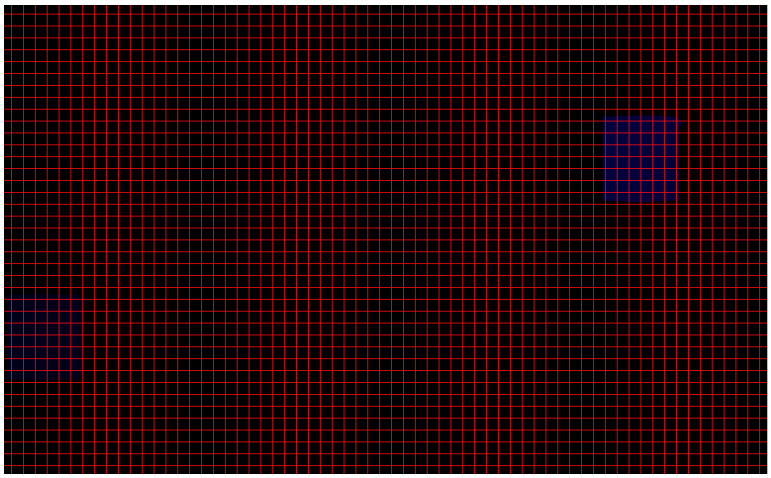


Fig 3.26- Una altre possible divisió en cel·les de l'escenari

En aquest cas l'escenari fa 9000 unitats × 9000 unitats. En total té (9000*40/3200)* (9000*40/3200) = 12657 cel·les. A cada fila hi ha 113 cel·les, amb uns 28 píxels cadascuna. Veiem doncs que és quasi tres vegades major que el cas anterior. De fet és el que volem, ja que no necessitem tanta precisió per saber a quina cel·la està cada edifici.

3.4.2- La classe Cela

Aquesta classe (figura 3.27) és la que farem servir per fer la divisió de l'escenari. Cada subdivisió serà un objecte cel·la. El conjunt de cel·les formaran l'estructura de dades de col·lisions.

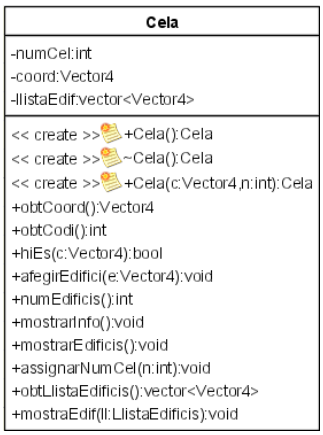


Fig 3.27- La classe Cela

Vegem els atributs que la formen:

int numCel: és l'identificador de la cel·la dins l'estructura de dades que les emmagatzema.

Vector4 coord: les quatre coordenades de la cel·la dins l'escenari. X,Y,Z,W equivalen respectivament a Xinicial, Xfinal, Zfinal,Zinicial.

Vector<Vector4> llistaEdif: un vector de vectors de quatre coordenades (que seran el codi de color dels edificis). Aquest vector guardarà la llista dels codis dels edificis que conté la cel·la.

Com a mètodes més importants tenim:

bool hiEs(Vector4 c): aquest mètode serveix per comprovar si el codi de color de l'edifici (c) està o no dins la llista de codis de color de la cel·la.

Un cop presentada la unitat mínima d'informació del mòdul de càlcul de col·lisions, anem a mostrar l'estructura formada per tot el conjunt de cel·les.

3.4.3- La classe EstructuraColisions

Aquesta classe (figura 3.28) és el nucli del mòdul de càlcul de col·lisions. És l'encarregada d'emplenar i proporcionar l'estructura de dades que faran servir els personatges de la multitud per comprovar si hi ha edificis amb els quals puguin

col·lisionar. Podem dir que està formada per objectes tipus Cel·la i a la vegada aquests objectes Cel·la es col·loquen per adaptar-se a la forma de l'escenari.

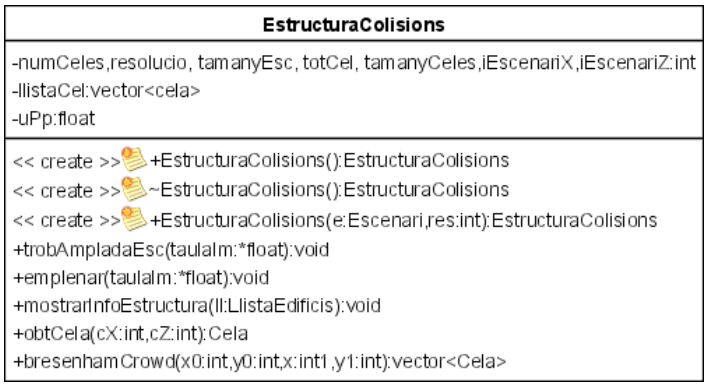


Fig 3.28- La classe EstructuraColisions

Vegem els atributs:

vector <Cela> llistaCel: un vector que guarda la llista de totes les cel·les que formaran l'estructura de dades.

int numCeles: el número de cel·les de l'estructura.

int resolucio: la resolució a la qual s'ha fet la fotografia de l'escenari.

int tamanyEsc: la mida de l'escenari.

int totCel: quantitat de cel·les que tenim a cada fila de l'escenari.

Aquests quatre últims atributs hem de dir que són de caire informatiu, ja que s'usen principalment per emplenar de forma més fàcil el fitxer de *log* de l'OGRE.

int tamanyCeles: la mida de cada cel·la en unitats de món

int iEscenariX,iEscenariZ: iEscenariX és el primer píxel que forma part de l'escenari (és pla) en X. IescenariZ és el primer píxel que forma part del pla en Z Tenim la coordenada en píxels de on comença el pla. Ens servirà com a punt de referència per transformar les coordenades en píxels dels edificis en coordenades de món de les cel·les i edificis. La figura 3.29 conté aquest esquema.

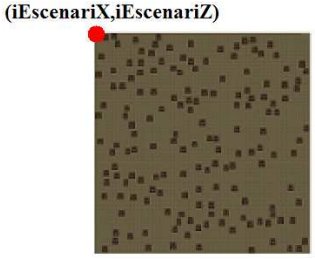


Fig 3.29- Ubicació dels píxels de control en la imatge de l'escenari

float uPp: un real que ens indica l'equivalència entre un píxel de la imatge i les unitats de món que li correponen, és a dir, quantes unitats de món són 1 píxel.

Ara que ja sabem amb quins atributs treballarem anem a analitzar els mètodes bàsics d'aquesta classe:

EstructuraColisions(Escenari e,int res): aquest és el constructor. El que fa és inicialitzar les cel·les i les deixa a punt per tal de poder assignar els codis dels edificis corresponents.

```
resolucio=res;
tamanyEsc=e.obtCostat();

totCel = arrodonirAlça((e.obtCostat()*40/resolucio);
numCeles=totCel*totCel;
tamanyCeles=Ogre::Math::Ceil(e.obtCostat()/totCel);//mida en
//unitats d'escenari de cada cel·la

//inicialitzem les cel·les. Anem d'esquerra a dreta i de dalt a baix
Vector4 coord;
int num=0;
float zP=(e.obtCostat()/2.f)*-1;//zP és z inicial
float zN= zP+tamanyCeles;//zN és z final
for(i=0;i<totCel+1;i++)
{
       float xN=(e.obtCostat()/2.f)*-1;//x negativ
       for(j=0;j<totCel+1;j++)
       {
             coord=Vector4(xN,xN+tamanyCeles,zN,zP);
             //Xinicial, Xfinal, Zfinal,Zinicial, les 4 coordenades
             //de la cel·la dins l'escenari

             Cela c(coord,num);

             llistaCel.afegir(c);
             xN = xN + tamanyCeles; //la coordenada X de la cel·la
             //es mou cap a la dreta

             num++;
       }

       zP = zP+tamanyCeles;
       zN=zP+tamanyCeles;
       //les coordenades Z de la cel·la es mouen cap avall
 }
```

Volem fer notar que afegim una cel·la pel costat dret i per baix. Això ho fem per temes d'arrodoniment: evitem que ens quedin trossos de pla sense estar dins a cap cel·la.

Cela obtCela(int cX,int cZ): aquest mètode serveix per obtenir la cel·la donada per la coordenada X i la coordenada Z de la mateixa, d'una forma directe.

```
int aux = -1*(tamanyEsc/2);
```

```
int aux2 = Ogre::Math::Ceil((aux-cZ)/tamanyCeles);
int posZ = (aux2*(totCel+1))*-1;

int auxx = -1*(tamanyEsc/2);
int aux4 = Ogre::Math::Ceil((auxx-cX)/tamanyCeles);
int posX = (aux4)*-1;

enter posFinal = posZ+posX;

Cela c = llistaCel[posFinal];

Retorna c
```

void trobAmpladaEsc(float* taulaIm): aquest mètode serveix per sel·leccionar dins la fotografia el pla, i per tant, delimitar l'escenari dins de la fotografia. El que fem és trobar els píxels de la imatge del principi de l'escenari i del final, i d'aquesta manera sabem quina amplada té, en píxels, l'escena. El que fem després és transformar aquesta amplada en píxels a mida d'escenari. Això ho farem servir per transformar a coordenades de món les coordenades en píxels d'on es troben els edificis (per exemple, si trobem un edifici al píxel de la imatge (1200,2000) això vol dir que en coordenades de món equivaldria, per exemple, al (1500,2500)).

```
int posPix=0
int pixFinal,pixInici
bool fi=false
int h=0;
//recorrem la taula de floats on està guardada la fotografia
while(h<resolucio && !fi)
{
	int j = 0;
	while(j<resolucio*4 && !fi)
	{
		r=taulaIm[posPix];
		g=taulaIm[posPix+1];
		b=taulaIm[posPix+2];
		w=1
		Vector4 codi(r,g,b,w)
		//obtenim el color del píxel i el guardem

		Vector4 bl(255/255.f,255/255.f,255/255.f,1)
		//el color blanc
		Vector4 ne(0,0,0,1)//el color negre


		if(codi!=bl)
		{
			//tenim pla
			pixInici=posPix//tenim el primer píxel del pla
			int posPixAux=posPix;

			Vector4 ne2=ne;
			Vector4 bl2(255/255.f,255/255.f,255/255.f,1);
			while(ne2!=bl2)
```

```
                        {
                              //ens saltem tot el pla. Hem de trobar
                              //altre cop el color blanc, que serà
                              //l'indicador que hem arribat al final del
                              //pla

                              r2=taulaIm[posPixAux];
                              g2=taulaIm[posPixAux+1];
                              b2=taulaIm[posPixAux+2];
                              w2=1;
                              ne2=Vector4(r2,g2,b2,w2);
                              posPixAux=posPixAux+4;
                        }

                        pixFinal=posPixAux-4;//tenim l'ultim píxel
                        fi=true;
                  }
                  else
                  {
                        //res. Primer ens saltem tots els píxels blancs
                  }
                  j=j+4;
                  posPix=posPix+4;
            }
            h++;
      }
      iEscenariX=(pixInici%resolucio)/4; //% dona la columna
      iEscenariZ=(pixInici/resolucio)/4; // / dona la fila
      uPp=tamanyEsc/((pixFinal-pixInici)/4.f); //per cada píxel, quantes
      //unitats d'escenari li pertoquen
}
```

Ara que ja sabem com passar de coordenades de píxel a coordenades d'escenari anem a veure el mètode que emplena l'estructura de dades amb els valors de la imatge.

void emplenar(float* taulaI): és l'encarregat d'emplenar les cel·les amb els codis de color corresponents donats per la imatge guardada. La idea és anar recorrent la imatge i cada cop que trobem un píxel que no sigui ni negre ni blanc (per tant un edifici) guardar a la cel·la corresponent aquest codi. Per tal de millorar l'eficiència del mètode, si el píxel llegit de l'edifici encara està a la mateixa cel·la (vol dir que encara estem llegint el mateix edifici i a la mateixa cel·la) no el processem i anem al següent. D'aquesta manera ens estalviem càlculs innecessaris.

```
trobAmpladaEsc(taulaI);//calculem l'amplada de l'escenari

float r,g,b,w;
enter posPix=0;
enter pixX,pixZ,difX,difZ;
float coordXEdif,coordZEdif;
float coordXFinal,coordZFinal;
Vector4 ant(0,0,0,1);
Vector4 bl(255/255.f,255/255.f,255/255.f,1);//blanc
```

```
Vector4 ne(0,0,0,1);//negre
//taulaI és la taula que guarda la imatge
Cela cAnt(Vector4(0,0,0,0),-1);
int h=0;
while(h<resolucio)
{
	int cl = 0;
	while(cl<resolucio*4)
	{
		r=taulaI[posPix];
		g=taulaI[posPix+1];
		b=taulaI[posPix+2];
		w=1;
		Vector4 codi(r,g,b,w);
		//llegim els valors RGBA de la imatge

		if(codi==bl || codi==ne)
		{
			//res
		}
		else//tenim edifici
		{
			ant=codi;//actualitzem el color llegit anterior

			/*la col i la fila del pix*/
			pixX=(posPix/4)%resolucio;
			pixZ=(posPix/4)/resolucio;


			difX=pixX-iEscenariX;
			//en píxels, la distancia entre inici escenari x i la
			//pos x del píxel llegit

			difZ=pixZ-iEscenariZ;
			//en pixels, la distancia entre inici escenari z i la
			//pos z del pixel llegit

			coordXEdif=difX*uPp;//en unitats d'escenari la coordX
			//(la distància que hi ha des del principi a aquest
			//punt)

			coordZEdif=difZ*uPp;//en unitats d'escenari la coordZ.
			//(la distància que hi ha des del principi a aquest
			//punt)

			enter meitat=tamanyEsc/2; //calculem el punt mig de
			//l'escenari, ja que està centrat a l'origen

			coordXFinal=coordXEdif-meitat;
			coordZFinal=coordZEdif-meitat;
			Cela c=obtCela(coordXFinal,coordZFinal);
			//ja tenim la cel·la on ha d'anar l'edifici

			if(cAnt.obtCodi()!=c.obtCodi())
			{
				//si la cel·la és nova, actualitzem la cel·la
				//anterior i seguim endavant
				cAnt.assignarNumCel(c.obtCodi());
				if(!c.hiEs(codi))
				{
					//si l'edifici no hi és l'hi posem
```

```
                                c.afegirEdifici(codi);
                                //actualitzem la cela
                                llistaCel[c.obtCodi()]=c;
                            }
                        }
                    }
                    posPix=posPix+4;
                    cl=cl+4;
                }
                h++;
            }
```

vector<Cela> bresenhamCrowd(int x0,int y0,int x1,int y1): és el mètode clau per l'execució del mòdul de càlcul de col·lisions. Aquest mètode retorna la llista de cel·les que passen per la recta que va des de la cel·la inicial (donada per X0,Y0) a la cel·la final (X1,Y1). És una adaptació de l'algorisme de Bresenham per dibuixar rectes.

L'algoritme de Bresenham (figura 3.30) per dibuixar rectes és un algoritme que determina, en una quadrícula (o *raster*), quins punts o cel·les s'haurien de pintar per aconseguir una aproximació a una línia recta entre dos punts donats. És un dels primers algoritmes que van aparèixer en la branca dels gràfics per ordinador.

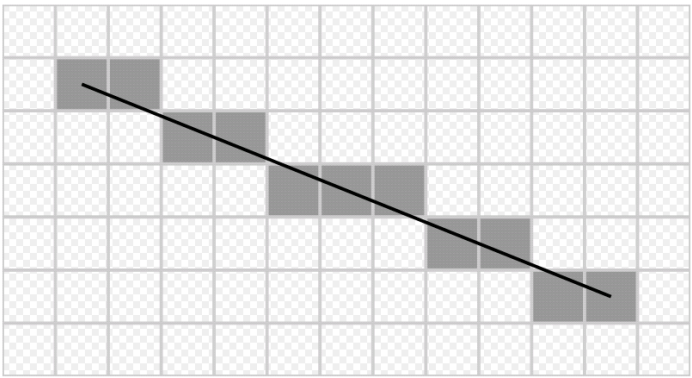


Fig 3.30- Diagrama del resultat de l'algoritme de Bresenham original

Com a convencions inicials tenia que la recta només podia ser dibuixada anant cap a la dreta i cap avall i que les coordenades dels píxels han de ser enteres. El que nosaltres hem fet és estudiar l'adaptació d'aquest algoritme de manera que les rectes puguin dibuixar-se en qualsevol direcció, tant sigui d'esquerra a dreta com de baix a dalt.

Vegem un exemple per entendre millor el motiu pel qual l'hem hagut d'implementar:

1. Suposem un escenari amb la següent distribució (figura 3.31).

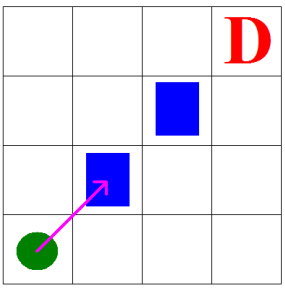


Fig 3.31- Situació inicial de l'escena

La rodona és el personatge, els rectangles blaus edificis i el vector lila indica la direcció del personatge.

2. Suposem ara que estem executant el GdM en una màquina ràpida, per tant, el temps que tarden a realitzar-se els càlculs són molt petits.
3. El personatge, per saber a quina posició ha d'anar a parar, fa servir la fórmula **posFinal = posInicial + velocitat*Δt**, ja que camina a velocitat constant. Si Δt és petit, i ho és ja que la màquina es potent, la posFinal no serà gaire lluny de la posInicial, donat que el personatge camina (per tant la velocitat és molt petita). En aquest cas, el personatge es trobarà que en la següent cel·la hi ha un edifici on, potser, col·lisionarà, i si ho fa, canviarà de trajectòria per evitar-lo.
4. Suposem ara que estem executant el GdM en una màquina poc potent. Seguint la mateixa fórmula que al punt 3, obtenim la nova posició destí del personatge (figura 3.32). Com en aquest cas Δt és molt gran (entre frame i frame ha passat molta estona), ens trobem que el personatge ha recorregut molt d'espai i ha arribat a la cel·la "D". Si haguéssim observat l'execució hauríem vist com el personatge feia un salt, mostrant un moviment no continuat característic d'un *framerate* baix:

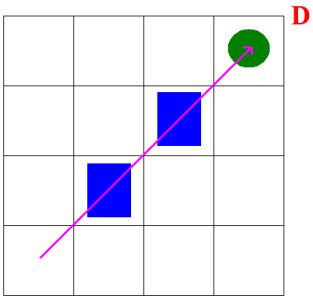


Fig 3.32- Situació final de l'escena

5. Si observem bé el camí que ha seguit, veurem que el personatge s'ha saltat dos cel·les de cop, on cada cel·la tenia un edifici. Per tant, aquest

fet, tot i no provocar cap error al GdM, demostra una falta de robustesa important, ja que el GdM hauria d'haver calculat que passava per unes cel·les on hi havia edificis i probablement hi col·lisionava (el personatge ha "travessat" els edificis).

6. El que hauria hagut de fer l'algorisme és sel·leccionar totes les cel·les per les quals passarà el personatge i mirar si hi ha algun edifici. Si és així, hauria de calcular si el personatge col·lisiona amb algun d'aquests edificis, i si és així, evitar-los (figura 3.33).

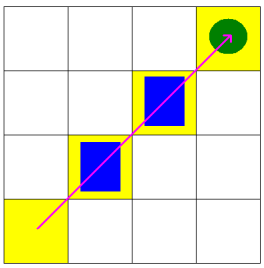


Fig 3.33- En groc, les cel·les obtingudes per Bresenham, els edificis de les quals hauriem de comprovar si col·lisionen o no amb el personatge

Anem a veure el codi del mètode:

```
vector<Cela> llistaCel;
Cela c;
enter x, y, dx, dy, p, incE, incNE, stepx, stepy; //stepx,stepy
són //l'increment que farem servir per moure'ns per la
quadrícula

dx = (x1 - x0);
dy = (y1 - y0);

if (dy < 0)
{
   //la recta baixa
   dy = -dy;
   stepy = -1;
}
else
      stepy = 1;//la recta puja

if (dx < 0)
{
   //la recta va cap a l'esquerra
   dx = -dx;
   stepx = -1;
}
else
   stepx = 1;//la recta va cap a la dreta

x = x0;
y = y0;

c=obtCela(x,y);
```

```
llistaCel.push_back(c);//la primera cel·la serà la inicial.
int codiCelAnt=c.obtCodi();//portem control de la cel·la
//anterior per no repetir


if(dx>dy)//el pendent <1,canvia la y
{
   p = 2*dy - dx;//calculem el pendent de la recta
   incE = 2*dy;
   incNE = 2*(dy-dx);
   while (x != x1)//mentre no arr al final
   {
         x = x + stepx;
         if (p < 0)
         {
               p = p + incE;
         }
         else
         {
               y = y + stepy;
               p = p + incNE;
         }

         c=obtCela(x,y);
         if(c.obtCodi() != codiCelAnt)
         {
               codiCelAnt=c.obtCodi();
               llistaCel.push_back(c);
         }
   }
}
else//pendent >=1,canvien les x
{
   p = 2*dx - dy;
   incE = 2*dx;
   incNE = 2*(dx-dy);
   while (y != y1)//mentre no arr al final
   {
         y = y + stepy;
         if (p < 0)
         {
               p = p + incE;
         }
         else
         {
               x = x + stepx;
               p = p + incNE;
         }
         c=obtCela(x,y);
         if(c.obtCodi() != codiCelAnt)
         {
               codiCelAnt=c.obtCodi();
               llistaCel.push_back(c);
         }
   }
}

return llistaCel;
```

Hem de recordar que aquesta implementació no és estrictament necessària, ja que en totes les màquines s'ha obtingut un bon rendiment per a un nombre raonable de personatges i edificis. L'hem implementat per dotar al GdM de més fortalesa davant possibles màquines amb un rendiment baix sense que això afecti al mòdul de col·lisions, evitant situacions anòmales, com per exemple que els personatges travessessin els edificis tot hi haver-hi una col·lisió.

En la figura 3.34 podem observar el diagrama de seqüència del mòdul.

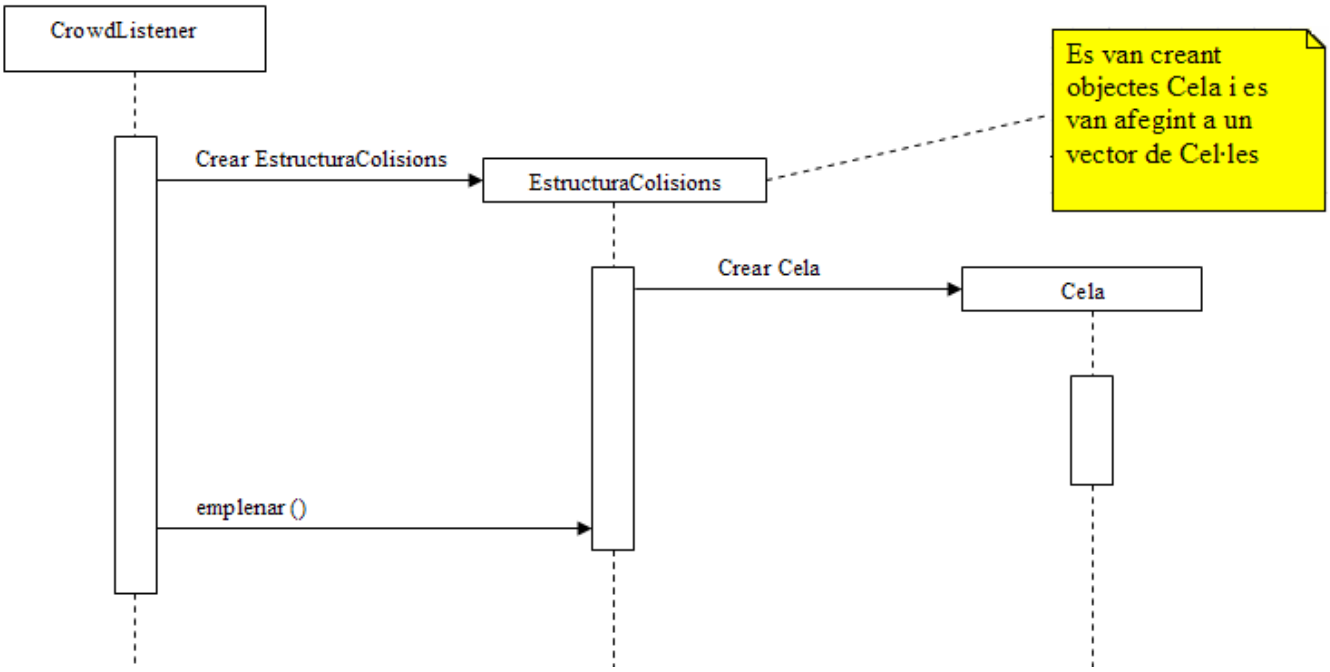


Fig 3.34 – Diagrama de seqüència del mòdul

El diagrama de seqüència anterior mostra com l'objecte de la classe CrowdListener crea un objecte de la classe EstructuraColisions. Aquest objecte, crea també objectes cel·la i, amb el mètode emplenar, les va inicialitzant amb la informació dels edificis de l'escenari.

3.5- El mòdul de la generació de multituds

Aquest podríem dir que és el nucli de la nostra aplicació. És l'encarregat d'organitzar i fer treballar conjuntament la resta de mòduls per tal d'aconseguir generar molts personatges movent-se per entre dels edificis, i, en última instància, evitar que els personatges col·lisionin amb aquests.

Aquest mòdul consta només de dues classes ben diferenciades: la classe CrowdGeneration i la CrowdListener. Les dues classes usen la característica de la

programació orientada a objectes de l'herència. En el primer cas, CrowdApplication hereta de la classe *ExampleApplication,* que és una classe base de l'OGRE pel desenvolupament d'aplicacions.

En el segon cas, CrowdListener hereta de la classe *ExampleFrameListener*, que a la vegada hereta de *FrameListener*. La classe *ExampleFrameListener* també és una classe base de l'OGRE per tal de facilitar la implementació d'aplicacions. També hereta de les classes *OIS:KeyListener* i *OIS:MouseListener*. Les dues són classes de l'OGRE que permeten escoltar els events que provenen de perifèrics, en el primer cas el teclat i en el segon la rata. Ens serviran per tal de poder gestionar correctament el mòdul de la interfíce d'usuari.

A grans trets podem dir que la classe CrowdGeneration serà l'encarregada d'inicar l'aplicació en sí. La classe CrowdListener serà la que s'encarregarà de fer moure la multitud controlant els events que succeixin i controlant la resta de mòduls que hi intervenen en temps real, així com d'instanciar la resta de les classes i fer-les interactuar entre elles.

Un cop presentades les dues classes base comencem doncs explicant la més senzilla: CrowdApplication.

3.5.1- La classe CrowdApplication

Principalment aquesta classe (figura 3.35) s'encarrega de la posta en marxa de l'aplicació. És a través seu que passen els paràmetres de configuració bàsica de l'OGRE.

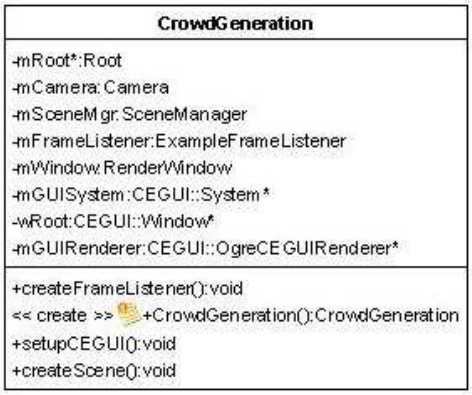


Fig 3.35 – Classe CrowdGeneration

Com hem comentat abans, aquesta classe hereta de la classe base de l'OGRE ExampleApplication. De per sí aquesta classe posseix els següents atributs:

Root *mRoot: és un punter a un objecte tipus Root, que és l'objecte del qual pengen totes les classe d'OGRE.

Camera* mCamera: un punter a un objecte càmera.

SceneManager* mSceneMgr: un punter a un objecte sceneManager.

ExampleFrameListener* mFrameListener: un punter a un objecte ExampleFrameListener.

RenderWindow* mWindow: un punter a un objecte renderWindow.

Tots aquests atributs ja estan creats i inicialitzats quan s'instancia la nostra classe CrowdApplication, pel sol fet d'heretar de la classe ExampleApplication. L'únic que hem de fer és modificar aquests atributs bàsics a les nostres necessitats.

Hem d'indicar també que hi hem afegit més atributs, però que pertanyen al mòdul de l'interfície d'usuari. Per tant és en aquest punt on els explicarem per tal d'obtenir una bona comprensió, ja que fora de context no l'obtindríem.

Com a mètode més important destaquem:

Void createFrameListener(void): aquest mètode és de la classe ExampleApplication, però en redefinir-lo aquí el que hem fet és posar-hi el nostre codi. La seva utilitat és la de crear un nou objecte frameListener. En el nostre cas serà un nou objecte CrowdListener:

```
mFrameListener = new CrowdListener(mWindow,mCamera,mSceneMgr,
     mCamera,mRoot,mGUIRenderer);

mRoot->addFrameListener(mFrameListener);
```

Bàsicament el que fem aquí és crear un nou objecte CrowdListener i assignar-lo a l'objecte root. Amb això fem que a partir d'ara el nou frameListener serà el nostre CrowdListener. Els paràmetres que es passen en la crida ja els explicarem quan parlem de la classe CrowdListener. Podem avançar que tots els paràmetres exceptuant el mGUIRenderer són els que crea la classe ExampleApplication per defecte. Aquest últim gestiona el mòdul de l'interfície d'usuari.

3.5.2- La classe CrowdListener

Podem dir, sense cap mena de dubte que aquesta és la classe (figura 3.36) més important conceptualment.

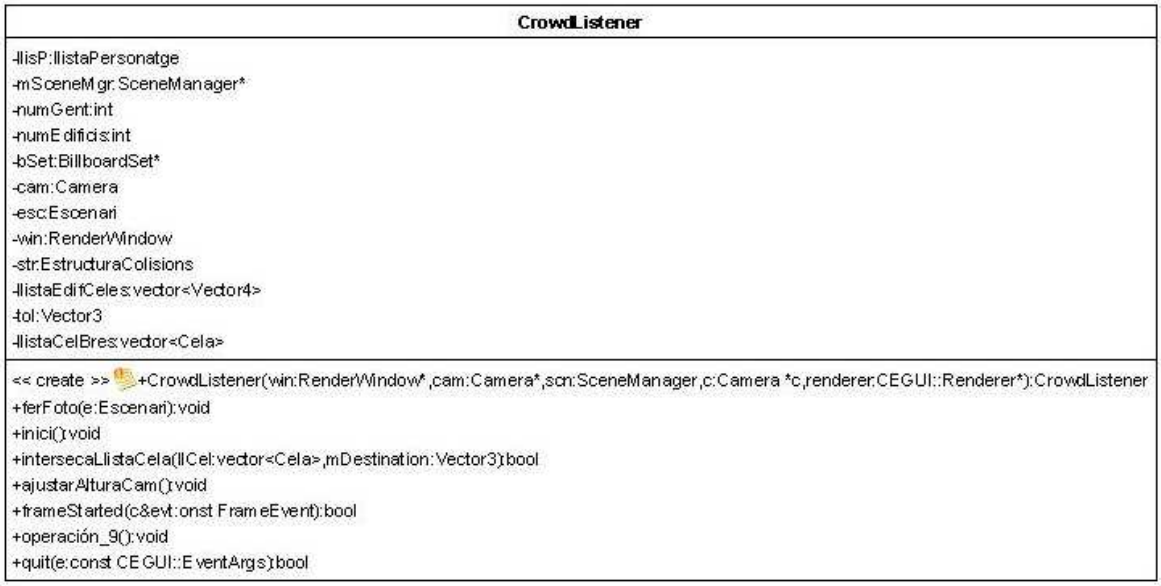


Fig 3.36 – Classe CrowdListener

La funció d'aquesta classe, a part de coordinar tots els mòduls que intervenen al GdM, és la de generar i animar la multitud a l'escenari, tot controlant les col·lisions. Per tant, tot el motor del GdM està dins aquesta classe. És per això que diem que és el nucli del GdM.

Anem a veure els atributs que conté:

llistaPersonatge llisP: la llista dels personatges de l'escenari.

SceneManager *mSceneMgr: el gestor d'escena.

int numGent,numEdificis: el número de persones que formaran la multitud i el número d'edificis que hi haurà a l'escena.

BillboardSet *bSet: el BillboardSet usat.

Camera *cam: la càmera de l'escena.

Escenari esc: l'objecte Escenari que s'instanciarà per la inicialització de l'escena.

RenderWindow *win: la finestra on es visualitzarà la nostra escena.

EstructuraColisions str: pertany al mòdul de càlcul de col·lisions. Guarda dins la seva estructura de dades informació de l'escenari disponible pels personatges per evitar col·lisionar amb els edificis.

vector<Vector4> llistaEdifCeles: pertany també al mòdul de càlcul de col·lisions. És un vector de objectes Vector4 (vectors de quatre coordenades). Guardaran els codis dels edificis anomenats al capítol on explicàvem la classe Edifici.

Vector3 tol: un vector que guardarà un valor concret, en el nostre cas (40,0,40). Només té sentit en el mòdul de col·lisions.

Vector<Cela> llistaCelBres: un vector que guarda objectes tipus Cel·la.

Com a atributs també tenim els ja explicats a l'apartat 3.3.2, del mòdul de preprocés, i tot un seguit de variables de tipus CEGUI, una llibreria de codi obert per generar interfícies d'usuari.

El funcionament general d'aquesta classe és força simple: Primer de tot, l'usuari ha d'escollir quin tipus d'element col·loca a l'escenari. Després ha de triar si els vol ubicats de forma aleatòria o de forma regular. Ha de triar també quantes persones han de formar la multitud i quants elements vol col·locar a l'escenari. Un cop iniciada l'execució, a cada *frame*, i per cada personatge, s'haurà d'evaluar on està i, en funció de la velocitat que porti i de la direcció on vagi, es calcularà la nova posició. Si en les cel·les per les quals ha de passar per arribar a la nova posició hi ha algun edifici, s'haurà de mirar si hi ha col·lisió, i si així és, s'haurà de canviar la direcció del personatge. Aquest procés es farà fins que l'usuari aturi l'execució.

Vegem els mètodes més importants:

void ferFoto(Escenari e): explicat a l'apartat 3.3.2, del mòdul de preprocés.

Void inici(): aquest mètode és l'encarregat de cridar els mètodes de la classe Escenari per inicialitzar-la. Obté també les dades que l'usuari ha escollit de la interfície i les passa l'Escenari per inicialitzar-lo. Inicialitza també la llavor per la generació de nombres aleatoris.

```
srand(time(NULL));
int tipusEdif;
if(ferCases)
      tipusEdif=1
altrament
      tipusEdif=2

costatEdifici =
Ogre::Math::Ceil(esc.obtMidaCostatEdifici(mSceneMgr,tipusEdif))
if(tipusEdif==1)
{
         if(numEdificis>=250)
         {
            esc.inicialitzar(mSceneMgr,mCamera,numGent,numEdificis,costat
            Edifici*numEdificis/6,costatEdifici*numEdificis/6,tipusEdif,t
            ipusPos) ;
         }
         else
```

```
         {
            int aux = (int)(numEdificis/40);
            int cnst=1;

            if(aux!=0)
                  cnst=3+aux*0.5;

            esc.inicialitzar(mSceneMgr,mCamera,numGent,numEdificis,(int)c
            ostatEdifici*numEdificis/cnst,(int)costatEdifici*numEdificis/
            cnst,tipusEdif,tipusPos);
         }
}
else
{
      if(numEdificis<=10)
         esc.inicialitzar(mSceneMgr,mCamera,numGent,numEdificis,costatEdi
         fici*numEdificis*3,costatEdifici*numEdificis*3,tipusEdif,tipusPo
         s) ;
      if(10<numEdificis && numEdificis<=20)
         esc.inicialitzar(mSceneMgr,mCamera,numGent,numEdificis,costatEdi
         fici*numEdificis*2,costatEdifici*numEdificis*2,tipusEdif,tipusPo
         s) ;
      if(20<numEdificis && numEdificis<80)
         esc.inicialitzar(mSceneMgr,mCamera,numGent,numEdificis,costatEdi
         fici*numEdificis/1.5,costatEdifici*numEdificis/1.5,tipusEdif,tip
         usPos) ;
      if(80<=numEdificis && numEdificis<120)
         esc.inicialitzar(mSceneMgr,mCamera,numGent,numEdificis,costatEdi
         fici*numEdificis/3,costatEdifici*numEdificis/3,tipusEdif,tipusPo
         s) ;
      if(120<=numEdificis)
         esc.inicialitzar(mSceneMgr,mCamera,numGent,numEdificis,costatEdi
         fici*numEdificis/4,costatEdifici*numEdificis/4,tipusEdif,tipusPo
         s) ;
}
llisP = esc.obtLlistaPersonatge();
```

bool intersecaLlistaCela(vector<Cela> llCel , Vector3 mDestination): aquest és un dels mètodes més importants de la classe, ja que és l'encarregat de decidir si un personatge col·lisiona o no amb algun edifici en el seu camí a seguir fins la posició (o cel·la) de destí. La llista de cel·les ve donada pel mètode "bresenhamCrowd" de la classe EstructuraColisions.

```
SceneNode *scn;
bool fi=false;
Cela c;
AxisAlignedBox ax;
LlistaEdifici llEd;
while(i<llCel.mida() && !fi)
{
      //per totes les cel·les mirem si col·lisiona amb algun dels
      //edificis que té a dins
      c=llCel[i] ;//obtenim la cel·la
      llistaEdifCeles=c.obtLlistaEdificis();//obtenim els edificis que té
      enter pos=0;
      booleà trob=false;
```

```
	Edifici e;

	if(llistaEdifCeles.size()==0)
	{
		//no té edificis
	}
	else
	{
		while(!trob && pos<llistaEdifCeles.size())
		{
			llEd= esc.obtLlistaEdificis();
			e=llEd. obtEdifici(llistaEdifCeles[pos]) ;
			ax= _getWorldAABB()
			if(ax.intersects(mDestination))
			{
				//tenim col·lisió
				trob=true;
				fi=true;
			}
			else
				pos++;
		}
	}
	i++;
}

return fi;
```

bool frameStarted(const FrameEvent &evt): el mètode més important de la classe, i també, de tot el GdM. És l'encarregat de coordinar tots els mòduls abans esmentats per tal que treballin conjuntament. Fa que els personatges es moguin i que no col·lisionin amb els edificis, generant la multitud.

El codi el mostrarem per parts degut a la seva extensió.

```
//capturem els moviments de la rata i el teclat.
mKeyboard->capture();
mMouse->capture();

//aquí hi ha les funcions de CEGUI per tal que l'usuari tingui la rata i
//el teclat disponible per fer anar en el mòdul de la interfície gràfica.

if(comencar)
{
	if(preproces)
	{
		//aquí fem desaparèixer part de la interfície gràfica i fem
		//aparèixer altres elements per mostrar en quin estat es
		//troba el preprocés

		inici();//crida al mètode inici, per inicialitzar l'escenari
		esc.abansPreproces(mSceneMgr);//el preparem pel preprocés
		esc.canviarColors(mSceneMgr);
		//canviem els colors dels edificis
		ferFoto(esc) ;//fem el preprocessament de l'escenari

		crowdRendTex->update();//dibuixa l'ecena
```

```
            pixelBuffer->blitToMemory(pixelBox) ;//emplenem la taula amb
            //la fotografia

            preproces=false;//ja no cal que entrem a fer el preprocés
            esc.despresPreproces(mSceneMgr) ;//restaurem l'escenari

            //creem i emplenem l'estructura de col·lisions
            str = EstructuraColisions(esc,textureResolution) ;
            str.emplenar(pDest) ;

            //alliberem la memòria
            TextureManager::getSingleton().remove("texFoto");
            crowdRendTex->removeAllListeners();
            crowdRendTex->removeAllViewports();

            //posicionem la càmera
            mCamera->setPosition(Vector3(0,50,0)) ;
            mCamera->setDirection( Vector3::NEGATIVE_UNIT_Z ) ;
      }
```

En aquest troç de codi es mostra com es crea l'escenari i s'inicialitza, així com la manera de cridar als diferents mètodes de les classes per realitzar el preprocessament i la posterior creació de l'estructura de col·lisions pels personatges. Vegem ara com es fa per aconseguir que es moguin i que no xoquin:

```
if(gui)
{
      Vector3 dir;
      bool fi;
      for(i = 0;i<esc.obtNumPersonatges();i++)
      {
            //per cada personatge de la llista
            pers = llisP.obtPersonatge(i) ;
            pers.setTempsPassat(pers.getTempsPassat() +
                  evt.timeSinceLastFrame) ;//actualitzem el temps que ha
                  passat des de l'ultim frame

            //calculem la distància que ha recorregut
            float move = pers.getVelocitat() * evt.timeSinceLastFrame;

            //mDestination guarda la posició on ha arribat
            mDestination = pers.getPosicio() + move*pers.obtDireccio();

            while(!esc.dinsEscenari(mDestination))
            {
                  mDestination = (mDestination + pers.getPosicio())/2;
            }
```

Aquest mentre serveix per posar mDestination dins els límits de l'escenari, ja que hi ha vegades, en les màquines lentes, que la diferència de temps es molt gran i fa que el personatge recorri molta distància, sortint dels límits de l'escena. Ho fem abans d'obtenir les cel·les amb Bresenham ja que sinó, amb una posició errònia, el mètode donaria error.

```
llistaCelBres=str.bresenhamCrowd(pers.getPosicio().x,pers.get
      Posicio().z,mDestination.x,mDestination.z) ;
//ja tenim la llista de cel·les per on haurà de passar
```

El bucle següent és el més important. Calcula dues coses: que el personatge no surti de l'escenari i que no xoqui amb cap dels edificis de les cel·les per les quals ha de passar. Notar que s'hi multiplica també una constant TOL. És una tolerància que serveix per fer que els personatges canviïn de direcció abans en el cas que es trobin obstacles.

```
fi=false;

while(!esc.dinsEscenari(mDestination+pers.obtDireccio()*tol)
||
intersecaLlistaCela(llistaCelBres,mDestination+pers.obtDirecc
io()*tol))
{
      bool fer=false;
      dir=pers.trobarDireccioAlter(aletori(-360,360)) ;
      pers.setDireccio(dir) ;
      //el personatge prova una altra direcció
      //recalculem la nova posició de destí
       mDestination = pers.getPosicio() +
           move*pers.obtDireccio();

      //tornem a arreglar mDestination per si el GdM corre
      //sobre màquines poc potents
      while(!esc.dinsEscenari(mDestination))
      {
            mDestination=(mDestination +
                 pers.getPosicio())/2;
      }

      //tornem a obtenir la llista de cel·les per les quals
       //passarà el personatge
      llistaCelBres=str.bresenhamCrowd(pers.getPosicio().x,pe
            rs.getPosicio().z,mDestination.x,mDestination.z);
}
```

Ja estem fora del mentre. El personatge ha evitat col·lidir i ja pot quedar-se a la nova posició on ja arribat. Obtenim el billboard assignat al personatge i li canviem la posició.

```
bSet=mSceneMgr->getBillboardSet(pers.getNomBBSet());
pers.setPosicio(mDestination,bSet) ;

//ara hem de mirar si al personatge se li ha de canviar la
//imatge del moviment.
if(pers.getTempsPassat()>=(80/pers.getVelocitat())*.035)
{
      //li posem la següent imatge del moviment.
      pers.prepararSeguentsCoordImatgeMoviment();
      //tornem a posar el seu temps entre frame i frame a 0
      pers.setTempsPassat(0) ;
}
//triem quina imatge li posem en funció de l'angle amb la
//càmera
```

```
            pers.posarTexturaAmbAngle(pers.angleAmbCamara(cam,mSceneMgr),
                  bSet) ;
            //finalment actualitzem el personatge a la llista de
            //personatges
            llisP.actualitzaDadesLlista(pers) ;
      }
}
gui=true;
}//tanca el if(començar)

//es va retornant cert fins que l'usuari prem "ESC". En aquest moment
//acaba l'execució del GdM
return ExampleFrameListener::frameStarted(evt) ;
```

Per tal de veure de forma general com actua el GdM hem dibuixat el diagrama d'activitat de l'aplicació general (fig 3.37).

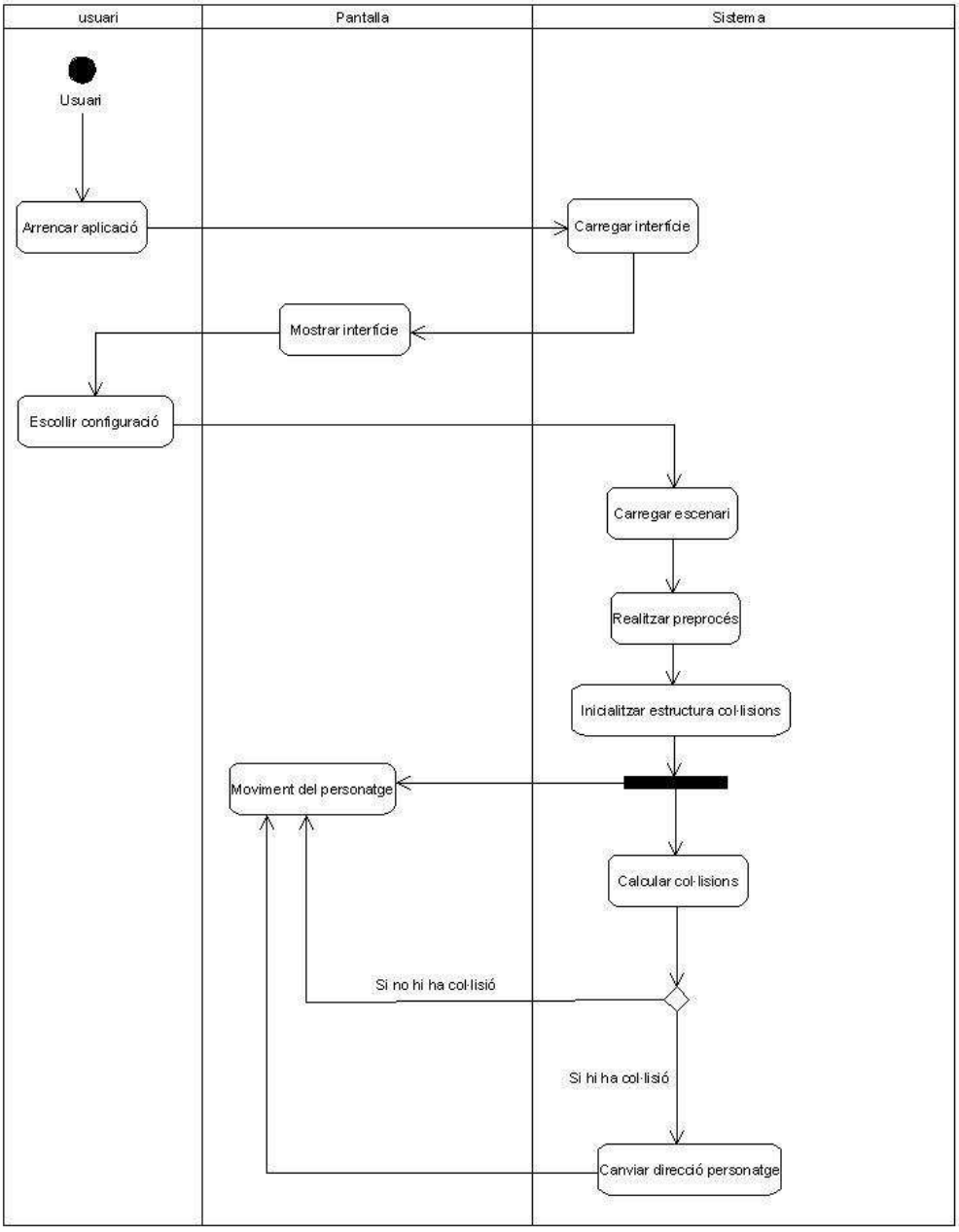


Fig 3.37- Diagrama d'activitat del sistema

El primer que fa l'usuari és iniciar l'aplicació. Quan s'inicia l'usuari veu la interfície gràfica que el sistema ha carregat. Mitjançant aquesta interfície, l'usuari escull quina configuració vol donar al GdM, i, un cop acceptada, el sistema procedeix a crear l'escenari, preprocessar-lo per crear l'estructura de col·lisions i finalment mostra per pantalla el moviment dels personatges. Si aquests estan a punt de col·lidir, el sistema és l'encarregat de canviar-los la direcció per evitar la col·lisió.

3.6- El mòdul de la interfície d'usuari

Tot programa orientat a l'usuari ha de proporcionar una interfície gràfica que permeti a l'usuari interactuar amb l'aplicació de forma fàcil i intuïtiva.

Un cop finalitzat el GdM hem pensat en una interfície molt simple que permetés a l'usuari triar quins tipus d'edificis representar, en quina quantitat, i quantes persones crear. Ens referim a si vol 20 o 500 personatges, si vol 20 o 100 edificis, si vol cases nomals o cubs o, finalment, si vol que es col·loquin de forma aleatòria o de forma regular (formant línies de cases).

Aquesta interfície s'ha fet usant la llibreria de codi obert CEGUI (*Crazy Eddies Graphic User Interface*), degut a que hi ha una gran comunitat d'usuaris que la mantenen, una molt bona documentació a internet i uns fòrums prou accessibles per la possible resolució de dubtes. És també la llibreria que porta l'OGRE per defecte.

En els següents apartats veurem, de forma molt superficial, ja que l'explicació de CEGUI no és objectiu del projecte, les característiques d'aquesta llibreria i les diferents classes de la CEGUI que hem fet servir.

3.6.1- La interfície final

Primer de tot vegem en la figura 3.38 l'aspecte final de la interfície d'usuari dissenyada.

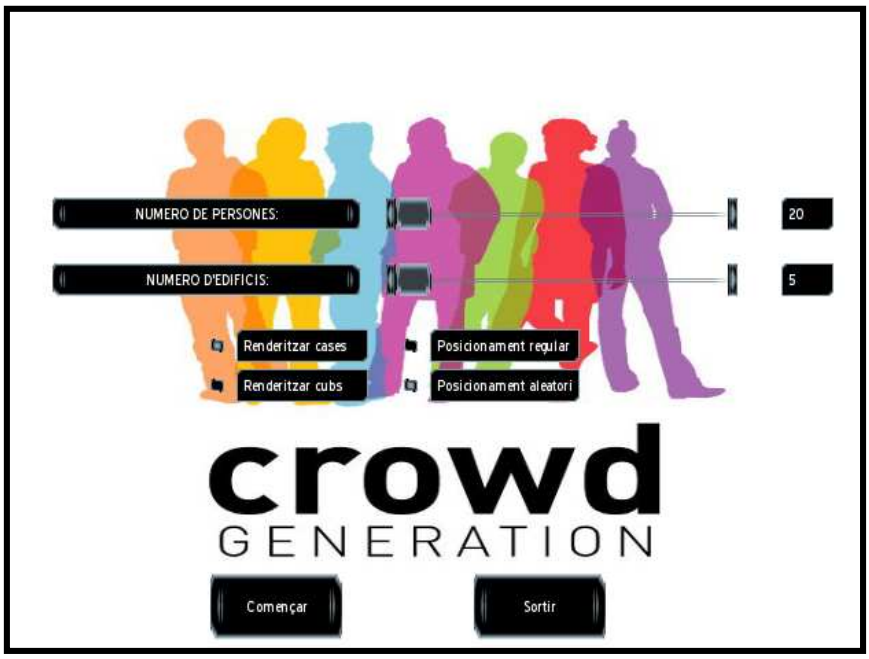


Fig 3.38- Aquestes etiquetes ens informen del procés de preparació del GdM

Consta de dues barres de desplaçament que controlen el número d'edificis i el número de persones.

Tenim també quatre botons d'opció, que ens permeten escollir entre si la visualització de cases o cubs, així com si els volem posicionats seguint un patró o de forma aleatòria.

Per últim tenim dos botons: el de començar i el de sortir. Tenim de fons el logotip creat pel GdM.

Cal recordar que és poc probable obtenir un bon rendiment del GdM si posem els dos controls al màxim, per potent que sigui la màquina. Així doncs el millor rendiment s'obté amb les quantitats equilibrades (molts personatges, pocs edificis o bé molts edificis i pocs personatges).

Hem creat també cinc quadres de text calibrats de l'1 al 5 i que es van alternant en funció de quin punt del preprocés es trobi el GdM (figura 3.39).

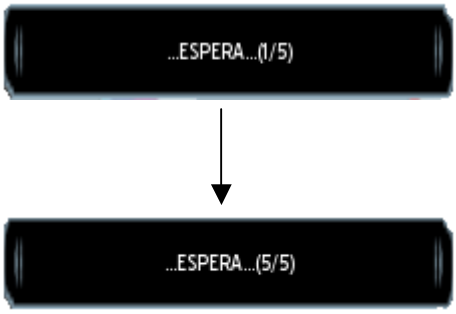


Fig 3.39- Aquestes etiquetes ens informen del procés de preparació del GdM

En els següents apartats es mostren les classes que hem fet servir per la creació de la interfície, així com certes parts del codi. Aquest codi és el definitiu de CEGUI.

3.6.2- Classes usades

Com hem comentat anteriorment, hem usat la llibreria CEGUI.

Una particularitat d'aquesta llibreria, és que tots els elements, ja siguin barres de desplaçament com botons, hereten de la classe CEGUI::Window. El que fa que agafin una forma o una altra és l'aparença que adopten mitjançant un fitxer de text xml que defineix el comportament d'aquests elements i el *casting* que se li aplica. Vegem un exemple:

```
CEGUI::Checkbox* cCas; //definim un checkBox
CCas=(CEGUI::Checkbox*)CEGUI::WindowManager::getSingleton().createWi
    ndow("TaharezLook/Checkbox", (CEGUI::utf8*)"cCas");
```

El que fem aquí és instanciar un nou objecte Finestra, el que passa que al fer el càsting el convertim a checkBox. El fitxer TaharezLook.looknfeel, a l'apartat de CheckBox, ens diu com ha de ser aquest checkBox

Hem de dir també que abans de res, s'ha de crear una finestra arrel, de la qual penjaran totes les altres, i un sistema GUI, que serà l'encarregat d'organitzar totes les finestres i de pintar-les per pantalla. Com a apunt important, quan es crea una finestra se li ha de donar un nom únic, d'aquesta manera ens hi podrem referir de forma unívoca. Vegem el procés:

```
mGUIRenderer = new CEGUI::OgreCEGUIRenderer(mWindow,
    Ogre::RENDER_QUEUE_OVERLAY, false, 3000,mSceneMgr);
mGUISystem = new CEGUI::System(mGUIRenderer);
//ja tenim el sistema GUI a punt per començar a crear finestres
wRoot=CEGUI::WindowManager::getSingleton().createWindow((CEGUI::utf8
    *)"DefaultWindow", (CEGUI::utf8*)"root");//"root" és el nom
mGUISystem->setGUISheet(wRoot);
//creem la primera finestra i la assignem com a arrel

//definim també el tipus de ratolí que es veurà per pantalla
mGUISystem->setDefaultMouseCursor((CEGUI::utf8*)"TaharezLook",
    (CEGUI::utf8*)"MouseArrow");//"MouseArrow" és el nom
```

Un cop inicialitzat el sistema CEGUI, vegem com hem creat la resta d'elements. Cal notar que en els mètodes per canviar la posició i la mida, els paràmetres estan posats en coordenades CEGUI, que significa per exemple que l'amplada i l'alçada de la finestra estan entre 0 i 1, com a molts altres sistemes. Vegem les classes que hem usat:

CEGUI::PushButton: serveix per crear botons (figura 3.40).

Fig 3.40- Aspecte d'un botó de la llibreria CEGUI

```
CEGUI::Window*sort=CEGUI::WindowManager::getSingleton().createWindow
    ("TaharezLook/Button","surt");//creem la nova instància
sort->setText("Sortir");//li posem el text
sort->setSize(CEGUI::UVector2(CEGUI::UDim(0.15,                0),
    CEGUI::UDim(0.10, 0)));//li canviem la mida
sort->setPosition(CEGUI::UVector2(cegui_reldim(0.55f),  cegui_reldim(
    0.87f)) );//li canviem la posició
wRoot->addChildWindow(sort);//l'afegim a la finestra root
```

CEGUI::Scrollbar: serveix per crear barres de desplaçament (figura 3.41).

Fig 3.41- Aspecte d'una barra de desplaçament de la llibreria CEGUI

```
scroll=static_cast<CEGUI::Scrollbar*>(CEGUI::WindowManager::getSingl
    eton().createWindow("TaharezLook/HorizontalScrollbar",(CEGUI::ut
    f8*)"scroll"));
scroll->setPosition(CEGUI::UVector2(cegui_reldim(0.45f),
    cegui_reldim( 0.30f)) );
scroll->setSize(CEGUI::UVector2(cegui_reldim(.4f),
    cegui_reldim(0.05f)) );
scroll->setDocumentSize(6000);//el límit
scroll->setPageSize(10);//quant avança la barra (imatge) a cada pas
scroll->setStepSize(10);//quant  avancem  (valor  que  representa  la
    barra, en el nostre cas edificis i persones) a cada pas
scroll->setScrollPosition(0);//comença a l'esquerra de tot
wRoot->addChildWindow(scroll); //l'afegim a la finestra root
```

CEGUI::Checkbox: serveix per crear botons d'opció (figura 3.42).

Fig 3.42- Aspecte dels botons d'opció (activat i no activat) de la llibreria CEGUI

```
cCas=(CEGUI::Checkbox*)CEGUI::WindowManager::getSingleton().createWi
    ndow("TaharezLook/Checkbox", (CEGUI::utf8*)"cCas");
cCas->setPosition(CEGUI::UVector2(cegui_reldim(0.25f),  cegui_reldim(
    0.50f)) );//triem la posició
cCas->setSize(CEGUI::UVector2(cegui_reldim(.05f),
    cegui_reldim(.05f)) ); //triem la mida
cCas->setSelected(true);//per defecte que estigui sel·leccionat
wRoot->addChildWindow(cCas); //l'afegim a la finestra root
```

CEGUI::Textbox: serveix per crear text (figura 3.43). En el nostre cas no l'hem usat, sinó que hem fet servir l'aparença dels botons i hem posat etiquetes de text.

Fig 3.43- Aspecte d'un quadre de text de la llibreria CEGUI

```
titNum=CEGUI::PushButton*)CEGUI::WindowManager::getSingleton().creat
    eWindow("TaharezLook/Button", (CEGUI::utf8*)"titNum");
titNum->setPosition(CEGUI::UVector2(cegui_reldim(0.07f),
    cegui_reldim( 0.30f)) );//canviem la posició
titNum->setSize(CEGUI::UVector2(cegui_reldim(0.35f),   cegui_reldim(
    0.05f)) );//canviem la mida
titNum->setText("NUMERO DE PERSONES:");//li posem el text
wRoot->addChildWindow(titNum); //l'afegim a la finestra root
```

Ara que ja tenim els elements bàsics per permetre a l'usuari entrar les seves preferències vegem com es fa per dotar a aquests elements de capacitat per captar els events que succeixin.

3.6.3- Events

A molt alt nivell, podem definir event com una acció que passa dins la interfície del programa o dins el programa mateix, provocada per l'usuari. Per exemple seria un event el fet de que l'usuari cliqui un botó, tanqui una finestra, entri un número en un quadre de text o simplement que mogui la rata per desplaçar-se fins al botó. A tots aquets events se'ls hi pot programar una resposta.

Tots els mètodes que "escolten" els dispositius s'han de posar dins la classe CrowdListener, ja que, si recordem, l'hem fet heretar de les classes OIS::KeyListener i de la OIS:MouseListener. Aquestes dues classes són les que ens permetran saber l'estat del teclat i de la rata, i d'aquesta manera saber si s'ha apretat una tecla o si hem fet clic amb la rata.

Només mostrarem dos exemples ben representatius del que és programar events en CEGUI, ja que la majoria es fan igual i tampoc és objectiu el projecte ensenyar CEGUI:

Fer que quan cliquem el botó sortir, la finestra de configuració del GdM es tanqui i que el programa s'acabi: en aquest cas hem de programar una resposta a l'event que es produeix quan la rata clica sobre el botó de sortir. Primer de tot l'hem de programar:

```
sort->subscribeEvent(CEGUI::PushButton::EventClicked,
    CEGUI::Event::Subscriber(&CrowdListener::quit, this));
//estem dient que quan el botó s'apreti, que se'n vagi al mètode
//"quit" de la classe CrowdListener.
```

Un cop programat, anem a veure el mètode "quit" què fa:

```
bool quit(const CEGUI::EventArgs &e)
{
        mContinue = false;
        return true;
}
```

Si ens hi fixem bé, l'únic que fa és dir que mContinue és fals. Amb el que això conseguim és que el mètode *frameStarted* retorni fals, i per tant, que el GdM s'aturi sense haver-se executat encara.

Fer que quan cliquem al botó dret de la barra de desplaçament, el comptador puji: com en el cas anterior, hem de fer que quan es cliqui al botó

dret de la barra de desplaçament augmentin el número de persones. Programem l'event:

```
bmes->subscribeEvent(CEGUI::PushButton::EventClicked,
     CEGUI::Event::Subscriber(&CrowdListener::pujar, this));
//bmes correspon al botó d'augmentar de la primera barra de
//desplaçament
```

Vegem el mètode "pujar" què fa:

```
bool pujar(const CEGUI::EventArgs &e)
{
   if(numGent<6000)
   {
      numGent=numGent+10;
      text->setText(CEGUI::PropertyHelper::intToString(numGent));
   }
   return true;
}
```

El que fa és cada cop que s'apreta el botó, augmenta en una el número de persones que s'han de visualitzar. També escriu en un quadre de text aquest número. Si es passa de 6000 ja no pujarà més.

Un altre mètode important és el *keyPressed*, que gestiona els events de teclat (el fem servir per controlar durant l'execució del GdM si l'usuari ha apretat la tecla "ESC" i, per conseqüent, hem de sortir).

Capítol 4: anàlisis dels resultats

En aquest capítol mostrarem els resultats que hem obtingut del nostre GdM, així com també valorarem el rendiment que aquest obté en funció de certs paràmetres d'entrada. El rendiment el valorarem en funció dels FPS (*frames per second*) que ens dóna l'OGRE en l'execució.

4.1- Resultats del GdM

A continuació mostrarem captures de pantalla de diferents moments de les execucions del GdM. Hem de recalcar que amb les imatges no es pot veure com els personatges col·lideixen amb els edificis ni tampoc la manera realista que tenen de moure's. Així doncs, és visualitzant una execució on realment es poden apreciar les virtuts del GdM.

10 cases amb posició aleatòria, 20 personatges (figura 4.1)

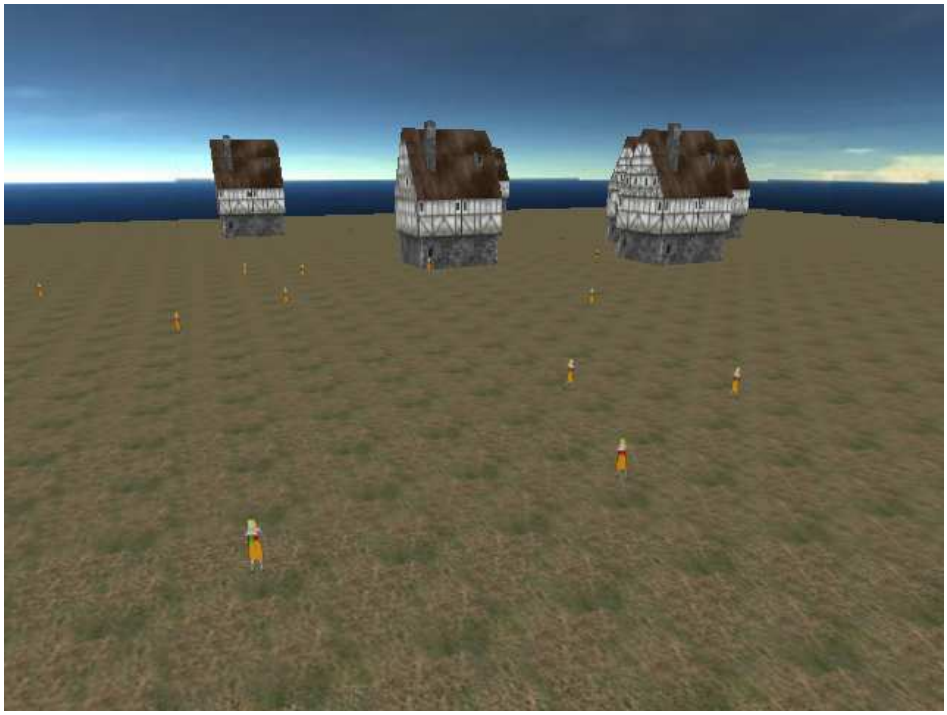


Fig 4.1- Execució amb molt pocs personatges

10 cases amb posició aleatòria, 500 personatges (figura 4.2)

Fig 4.2- Execució amb un número baix de personatges

10 cases amb posició aleatòria, 3500 personatges (figura 4.3)

Fig 4.3- Execució amb un número alt de personatges

50 cubs amb posició aleatòria, 500 personatges (figura 4.4)

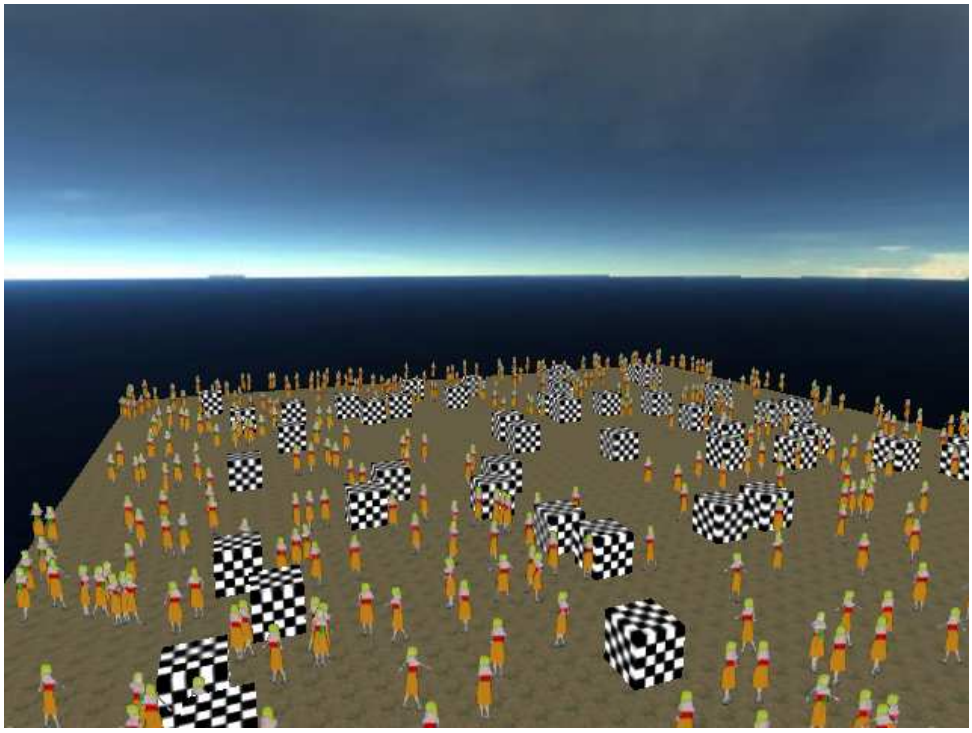


Fig 4.4- Execució amb un número baix de personatges

100 cubs amb posició aleatòria, 3500 personatges (figura 4.5)

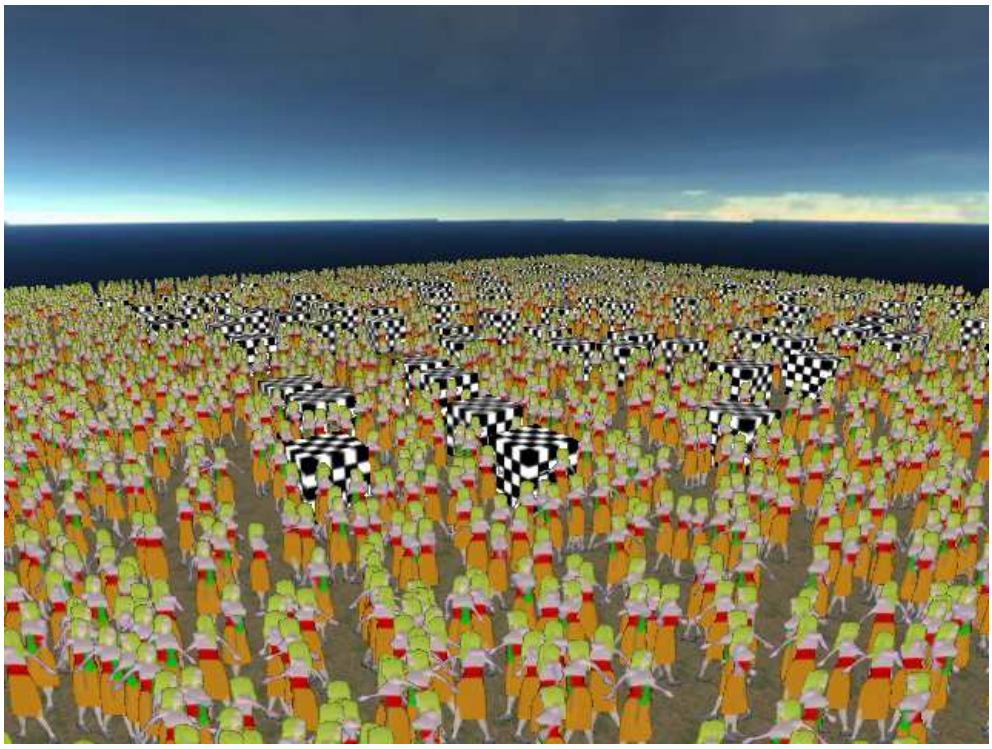


Fig 4.5- Execució amb un número alt de personatges

300 cubs amb posició regular, 6000 personatges (figura 4.6)

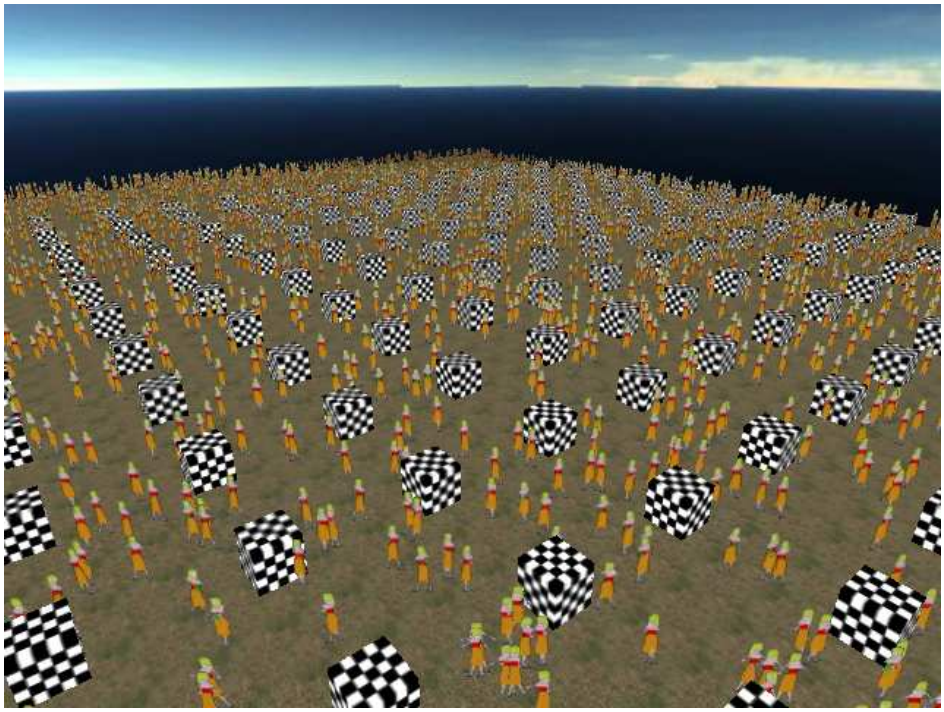


Fig 4.6- Execució amb un número molt alt de personatges

Càlcul de les col·lisions (figura 4.7)

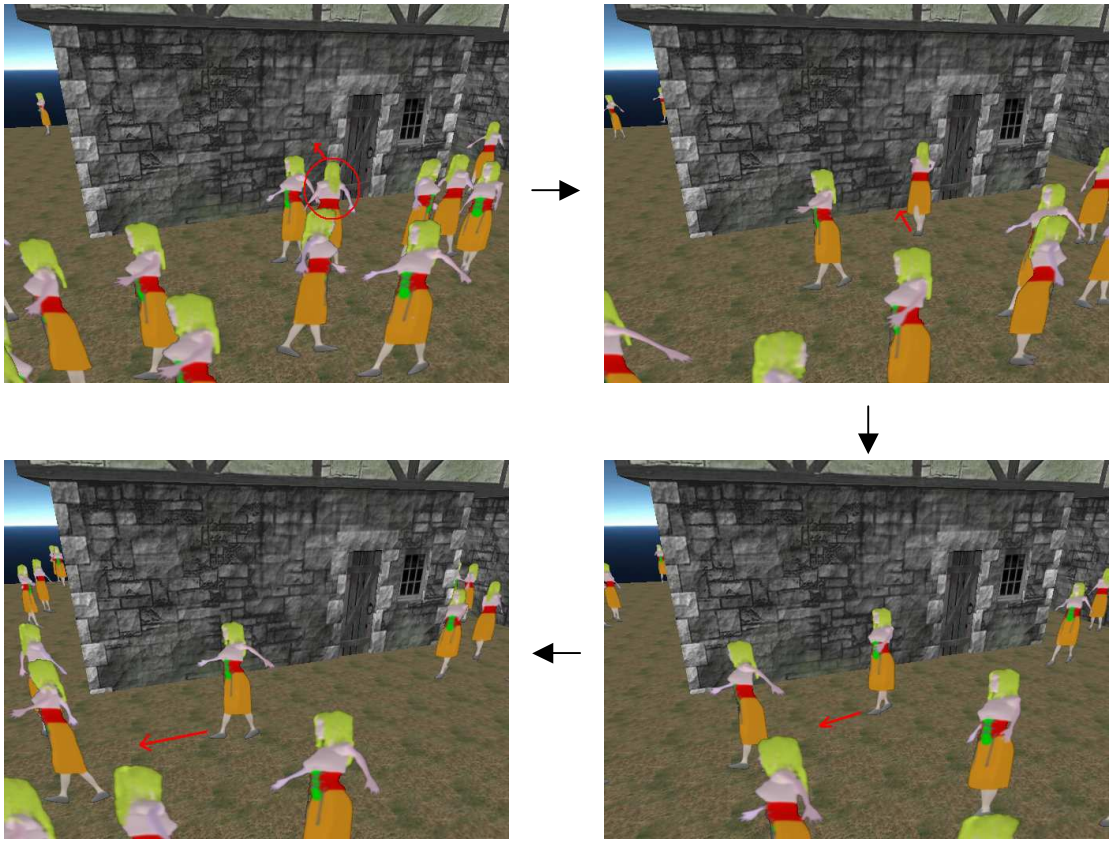


Fig 4.7- Exemple del personatge evitant la col·lisió amb l'edifici

4.2- Anàlisi del rendiment

Abans de res, i com a idea general, hem de dir que els resultats obtinguts a la taula de la figura 4.8 canviaran si les proves les féssim en una màquina més potent. Vist el que es pot arribar a obtenir en el punt anterior, vegem ara quin cost, en FPS, té aconseguir les escenes anteriors. Observant les execucions realitzades fins al moment, si posem molts edificis i molt pocs personatges, aquests personatges no notaran la baixada de rendiment i portaran un moviment prou realista. Si pel contrari posem molts personatges i pocs edificis, aquests personatges si que notaran la baixada de rendiment i, per tant, el seu moviment no serà del tot realista. Altrament, si posem el màxim de personatges i el màxim d'edificis, el rendiment del GdM no serà molt bo. Recordem que si la màquina fos més potent, el que aconseguiríem és, amb les mateixes dades de la taula, fer augmentar els valors de FPS.

El que volem dir amb tot això és que, en triar les execucions, hem de pensar que si volem molts personatges el moviment pot no ser del tot ràpid, en canvi si posem pocs personatges, aquests segur que tindran un bon moviment.

A continuació es mostra la taula obtinguda (figura 4.8) amb el tipus, el número d'edificis, de persones i els FPS obtinguts. Algunes execucions són potser una mica exagerades, però són només per obtenir dels FPS en casos extrems.

(NOTA: el fet de que per pantalla surti la informació dels FPS fa baixar una mica el rendiment. En aquesta taula es mostren els FPS amb aquest quadre de text visualitzant-se)

NÚMERO D'EDIFICIS	NÚMERO DE PERSONES	FPS cases	FPS cubs
10	20	**351**	**352**
40	200	**330**	**331**
60	500	**317**	**317**
80	1000	**171**	**171**
100	1500	**98**	**99**
125	2500	**41**	**43**
150	3500	**23**	**23**
200	4500	**14**	**14.75**

300	6000	**7.20**	**7.30**
5	6000	**8.43**	**8.90**

Fig 4.8- Taula de rendiments 1

75 EDIFICIS	NÚMERO DE PERSONES	FPS cases	FPS cubs
	100	**345**	**347**
	200	**321**	**324**
	500	**317**	**318**
	1000	**171**	**171**
	1500	**100**	**102**
	2500	**41**	**41**
	3500	**23**	**23.75**
	4500	**14**	**14.75**
	5500	**9.93**	**10**
	6000	**7.43**	**7.60**

Fig 4.9- Taula de rendiments 2, amb el número d'edificis constant

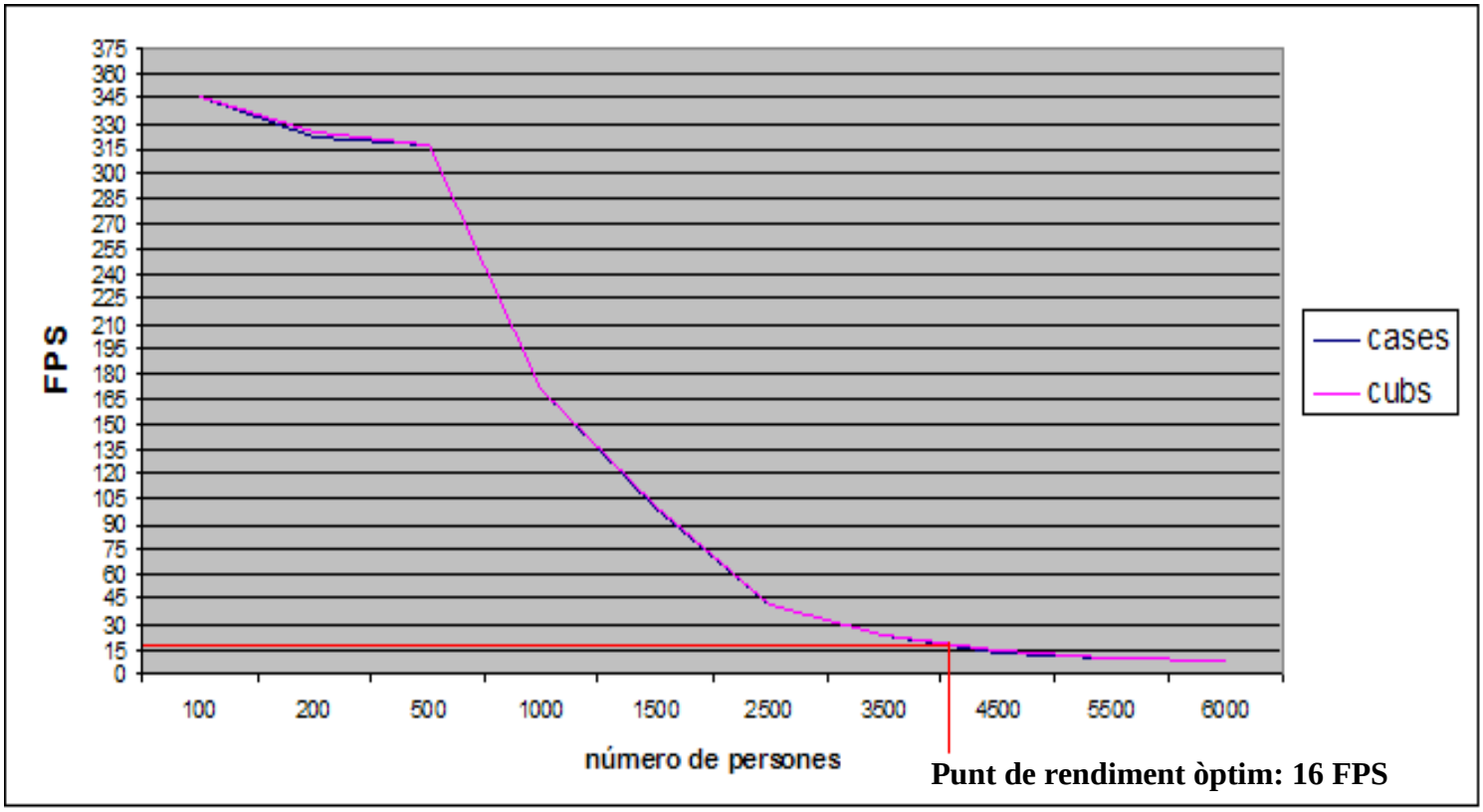


Fig 4.10- Gràfica de la taula 4.9, amb el número d'edificis constant (75)

Observant les taules anteriors, podem extreure vàries conclusions, algunes d'elles evidents:

- Com ja haviem vaticinat més amunt, com més elements hagin de visualitzar-se, pitjor serà el rendiment del GdM. Els pitjors FPS s'han obtingut quan el GdM funcionava amb les dades d'edificis i persones al màxim.
- Altrament, el millor rendiment s'ha obtingut quan teníem molt pocs edificis en pantalla i el mínim de personatges.
- Si ens fixem en valors mitjans, ni molts edificis ni molts personatges (60 edificis per 500 persones), tenim que els FPS són molt bons.
- Proporcionalment, i a partir del llindar marcat a la gràfica, si anem augmentat els elements, el rendiment comença a baixar, en part perquè ja tenim un bon nombre de personatges a l'escena, i per conseqüent, els càlculs de col·lisions i el cost de processar els vèrtexs augmenten Podem estimar que el nombre màxim de personatges que aproximadament podem tenir sense que ens baixi molt el rendiment és 4100.
- Finalment, i continuant el punt anterior, podem afirmar que la variable que té més pes dins el rendiment final del GdM és el número de personatges. Fixem-nos que, amb el màxim de personatges, en passar del mínim d'edificis al màxim, el rendiment és pràcticament el mateix. Si en canvi deixem els edificis al màxim, i passem del mínim de personatges al màxim, veurem com el rendiment baixa notablement.
- Si en comptes de visualitzar cases renderitzem cubs, pràcticament tenim el mateix resultat, lleugerament millor gràcies a que els cubs tenen una geometria molt més simple que la casa.
- El fet que a major número de personatges pitjor rendiment, pot ser degut a que s'han de calcular moltes més col·lisions. A més, i si ens fixem en el codi del mètode *frameStarted*, de la classe crowdListener, el que fem és per cada personatge de l'escena calcular la posició i la col·lisió. És normal, llavors, que al augmentar el número de personatges augmenti també el temps d'espera per processar tots els personatges (i per conseqüent el número de vèrtexs).

Si ens fixem només en punts de la implementació, podem plantejar les següents conclusions:

- Recordant l'apartat 3.4.1, podem dir que el fet de dividir l'escenari en cel·les ens ha significat una millora pel que fa al cost de calcular les col·lisions.
- Usant impostors, i fixant-nos en el cas pitjor, fem un càlcul aproximat dels vèrtexs a tractar i el que ocupen a memòria:

  Si fem servir impostors ocupem 6000 personatges * 4 vèrtexs (1 per cada extrem del *billboard*) * 4 bytes per vèrtex = 96 KB. A més, en total tindrem 4*6000 = 24.000 vèrtexs. Si en canvi utilitzem el model original ocuparem = 6000 personatges * 2355 vèrtexs (tots els del model en Maya) * 4 bytes per vèrtex = 56.52 MB, i en total tindrem 2355*6000 = 14.130.000 vèrtexs. Observant aquests números podem dir que fent servir els models originals s'han de processar molts més triangles i vèrtexs que no pas fent servir impostors, per tant, el cost de processament és més baix en el cas d'aquests últims, i per conseqüent, millorem la velocitat.

  Amb aquests resultats, podem plantejar encara una última idea: fixem-nos que, amb uns 2500 personatges, obtenim un *framerate* acceptable (figura 4.9). Aquests personatges utilitzen 2500*4 = 10.000 vèrtexs. Si recordem, un sol model del personatge ja està format per 2355 vèrtexs. Suposem que aquest model el simplifiquem fins a quedar-nos només amb 1000 vèrtexs. El que podem dir, és que per obtenir un *framerate* com si tinguéssim 2500 personatges, com a màxim podríem visualitzar 10 personatges a l'escena, cosa totalment inviable ja que no assolim els objectius. És gràcies a aquesta comparació per la qual podem afirmar que usant impostors hem millorat considerablement el rendiment de visualitzar centenars de personatges, complint així un dels objectius del Projecte.
- A l'apartat 3.4.3 explicàvem que el personatge, per saber quina quantitat d'espai havia recorregut, feia servir la fórmula **posFinal = posInicial + velocitat * Δt**. Inicialment havíem pensat fer servir la idea que el personatge es movia una quantitat fixa a cada *frame* (per exmple 0.5 unitats d'escenari per *frame*). Aquesta idea és equivocada,

perquè aquest valor no seria constant, és a dir, si el GdM s'executava en una màquina ràpida, el personatge avançava més ràpid que en una màquina lenta, ja que en la lenta hi havia menys *frames* per segon que en la ràpida. És per aquest motiu que vam escollir la fórmula abans explicada.

Com a apunt final, podem dir que si les proves s'han efectuat en un processador Intel Pentium IV a 3.0 GHz amb 1 GB de Ram i la tarja és una Nvidia Fx 5500. Si aquestes proves s'haguessin efectuat amb una màquina més potent i amb una millor tarja gràfica, hauríem augmentat els valors de les taules, ja siguin el nombre d'elements com els *frames* obtinguts (les corbes de la gràfica de la figura 4.10 s'haurien desplaçat cap amunt).

Capítol 5: conclusions

En aquest últim capítol explicarem a quines conclusions hem arribat després d'implementar el GdM i analitzar els resultats obtinguts.

5.1- Conclusions

Gràcies a aquest projecte, hem demostrat que usant impostors (en el nostre cas els *billboard*s) es poden generar grans multituds amb un cost computacional millor que generant-les sense usar impostors (visualitzant el *mesh* directament).

També s'ha pogut comprovar que amb pocs recursos es poden obtenir uns resultats acceptables en quant a qualitat de visualització i com a quantitat de personatges visualitzats (s'han aconseguit generar unes 4000 persones a temps real sense que el rendiment del sistema hagi baixat molt).

Hem de dir també que si haguéssim volgut generar aquesta quantitat de persones utilitzant el model original hagués resultat pràcticament impossible, ja que s'haurien hagut de tractar una quantitat de vèrtexs molt superior a la capacitat del sistema. És per això que podem afirmar que usant impostors és com s'aconsegueixen uns resultats prou bons, assolint així un bon equilibri entre qualitat de visualització i quantitat de persones visualitzades.

5.2- Projectes futurs

Gràcies a l'entorn obert del nostre projecte i la facilitat que té per poder adaptar-hi mòduls, al GdM se li obre un gran ventall de possibilitats per explorar, de les que destaquem:

- **Col·lisions entre personatges:** aquest mòdul seria l'encarregat d'evitar que els personatges xoquessin entre ells.
- **Moviment per afinitats:** seria interessant degut a que els personatges es mourien en funció de les afinitats que tindrien els uns amb els altres, arribant a crear moviments de grups de persones en comptes de persones soles caminant sense anar a cap objectiu marcat.
- **Millora de la visualització del personatge d'aprop:** com s'ha pogut comprovar, quan el personatge està molt a prop, la qualitat

d'aquest baixa molt. Es podria sol·lucionar canviant el *billboard* pel *mesh* del personatge.

- **Creació de ciutats reals:** es podria arribar a dissenyar una gran ciutat real.
- **Més intel·ligència:** es podria dotar als personatges de major intel·ligència, decidint ells a on volen anar (possibles edificis, per exemple), canviar la velocitat i inclús parar-se, etc..
- **Fenòmens meteorològics:** es podria fer més realista l'escena amb aquests nous efectes.
- **Afegir fotografies del personatge variant l'alçada de la càmera:** ara només tenim fotografies del voltant del personatge amb l'altura de la càmera sense variar. Si anéssim pujant la càmera i fent les fotografies tindríem un nou nivell de realisme que ens permetria visualitzar correctament el personatge si la càmera de l'observador pugés en alçada (ara si la càmera puja el personatge es veu "pla", sense volum).

Hem escollit aquests, però en queden molts altres. És per això que, tot i que el GdM encara està en fase incial, es podria aconseguir una gran aplicació de simulació de persones en temps real, dotades d'una bona intel·ligència artificial.

Bibliografia

Ogre manual. 2006. http://www.ogre3d.org/docs/manual

Van Verth, James, Lars M. Bishop. *Essential mathematics for games and interactive applications: A programmer's guide*. Morgan Kaufmann Publishers, 2004.

Alias Learning Tools. *Learning Maya 7: foundation*. Sybex, 2005.

Wilkins, Mark, Chris Kazmier. *MEL Scripting for Maya Animators, Second Edition*. Morgan Kaufmann Publishers, 2005.

Junker, Gregory. *Pro OGRE 3D Programming*. Apress, 2006.

Fernando, Rajima. Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Professional, 2003.

Edwards, C. H., Penney, David E. *Cálculo con geometría analítica*. Prentice-Hall, 1996.

Stroustrup, Bjarne. El *Lenguaje de programación C++ (2ª ed.)*. Addison-Wesley Iberoamericana, 1993.

Highend3d. 2008. *Highend3d*. http://www.highend3d.com

Learning-Maya.com. 2008. *Maya Tutorials Database*. http://www.learning-maya.com/index.php

OGRE Forums. 2008. *OGRE Forums*. http://www.ogre3d.org/phpBB2/index.php

Crazy Eddie's GUI System. 2008. *Crazy Eddie's GUI System*. http://www.cegui.org.uk/wiki/index.php/Main_Page

Crazy Eddie's GUI System – Fòrums. 2008. *Crazy Eddie's GUI System – Forums*. http://www.cegui.org.uk/phpBB2/index.php

Nvidia Developer Fòrums. 2008. Nvidia Developer Fòrums. http://developer.nvidia.com/fòrums/index.php?

Bresenham's line algorithm. 2008. *Wikipedia*. http://en.wikipedia.org/wiki/Bresenham's_line_algorithm

Algoritmo basado en la ecuación de la recta. 2008. *Hèctor E. Medellín Anaya*. http://galia.fc.uaslp.mx/~medellin/Applets/LineasRectas/Recta.htm

Annex: instal·lació del GdM

En aquest apartat donarem les pautes a seguir per tal d'instal·lar correctament el GdM i fer-lo funcionar

El GdM consta d'un executable i de varis arxius de materials i d'imatges que fan possible el funcionament correcte del programa. Com a fitxers d'imatge tenim:

- Atles.png
- Negre.png
- Esc.png

Com a fitxers de material tenim:

- Atles.material
- Escac.material
- Example.material
- Foto.material

I finalment el *shader*:

- GameObjStandard.cg

Abans d'explicar on han d'anar aquests fitxers expliquem primer com instal·lar el motor gràfic OGRE:

- Executar l'instal·lador de l'OGRE 1.4.5 SDK (la que s'ha fet servir per crear el GdM).
- Seguir les passes indicades per instal·lar-lo.

Al finalitzar el procés l'OGRE ja estarà instal·lat al nostre ordinador.

Un cop ja tenim el motor, vegem on col·locar els fitxers esmentats anteriorment:

- A la ubicació "C:\OgreSDK\media\materials\textures" hem de posar-hi els fitxers d'imatge.
- A la ubicació "C:\OgreSDK\media\materials\scripts" hem de posar-hi els fitxers de material.
- A la ubicació "C:\OgreSDK\media\materials\programs" hem de posar-hi el shader.

Ara que ja tenim els fitxers a la ubicació adequada ja podem executar el GdM!