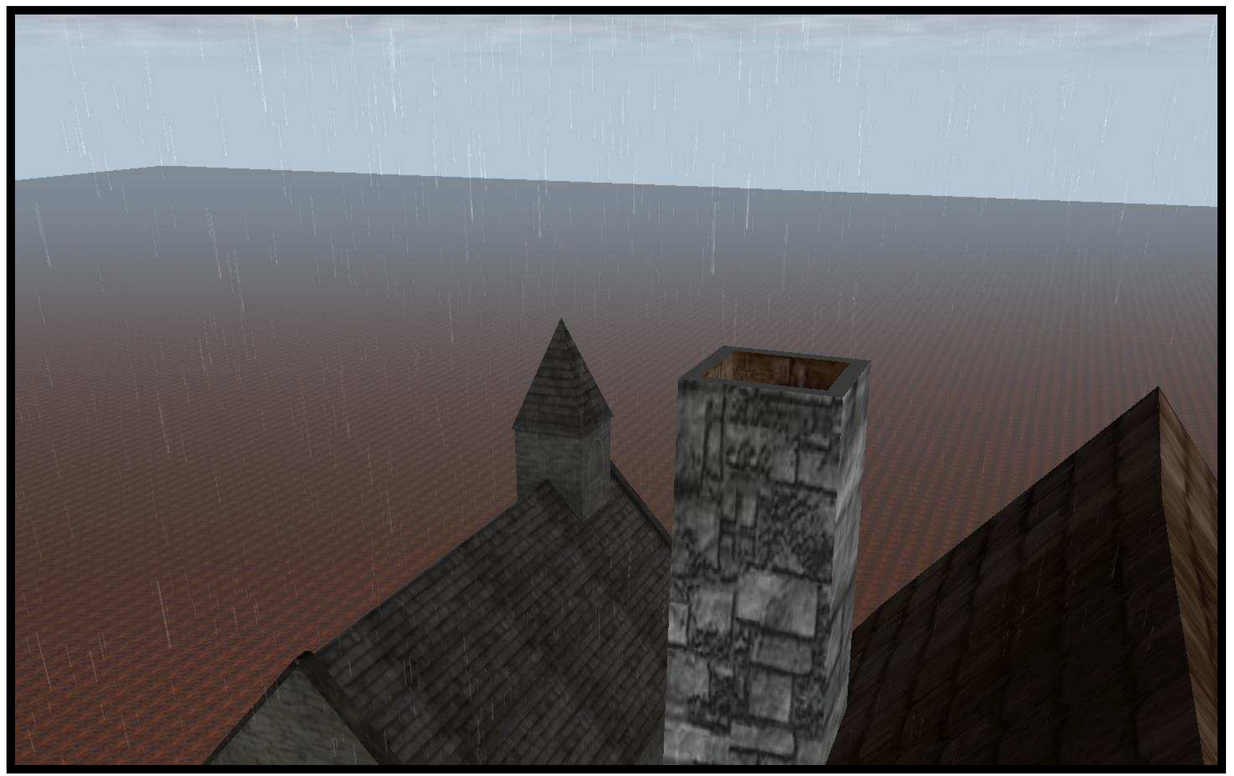
Projecte/Treball Fi de Carrera

Estudi: Enginyeria Informàtica. Pla 1997

Títol: Visualització de pluja realista en temps real



Document: Memòria

Alumne: Jordi Molist Pladevall

Director/Tutor: Gustavo Patow
Departament: Informàtica i Matemàtica Aplicada
Àrea: LSI

Convocatòria (mes/any): 09/2008

1 Capítol 1 -Introducció- 6
1.1 Presentació del problema 8
1.2 Motivació del projecte 9
1.3 Abast 9
1.4 Eines utilitzades 9
1.5 Planificació 10
1.6 Estructura de la documentació 12
2 Capítol 2 -Fonaments previs- 14
2.1 Conceptes de programació de la GPU 15
2.1.1 Descripció del procés de renderització 15
2.1.2 Detalls del procés de renderització 16
2.1.3 Llenguatges de programació per GPU 18
2.1.4 Anàlisis de motors gràfics 19
2.2 Estudi del motor gràfic Ogre3d 20
2.2.1 Objectes principals d'Ogre 20
2.2.2 Materials a l'Ogre 28
2.2.3 Sistemes de partícules d'Ogre 34
2.2.3.1 Atributs generals 34
2.2.3.2 Emitters 35
2.2.3.3 Affector 36
2.2.4 Simulació de pluja a l'Ogre 38
2.3 Física de la pluja 41
2.4 Funcions de densitat de probabilitat (PDF) i de distribució acumulatives (CDF) 43
2.5 Renderització de gotes d'aigua de pluja 45
2.5.1 Idea General 45
2.5.2 Motivació 45
2.5.3 Treballs Previs per la visualització de pluja 45
2.5.4 Aparença de les gotes de pluja 47
2.5.5 Base de dades les gotes de pluja 48
2.5.6 Algorisme de renderització de pluja 49
2.5.7 Captura de les gotes d'aigua reals 49
2.5.8 Base de dades de les gotes d'aigua renderitzades 50
3 Capítol 3 –Disseny i implementació- 52
3.1 Editor de pluja 54
3.1.1 Creació de les figures 54
3.1.1.1 Classe figura 55
3.1.1.1.1 Classe Cercle 55
3.1.1.1.2 Classe Rectangle 56
3.1.1.1.3 Classe Poli 57
3.1.1.2 Classe EstructuraFigures 57
3.1.2 Construcció de l'entorn de finestres 60
3.1.2.1 Classes creades 60
3.1.2.1.1 Classe MyApp 60
3.1.2.1.2 Classe MyFrame 61
3.1.2.1.3 MyCanvas 63
3.1.2.1.4 MyMiniFrame 66
3.1.2.1.5 MyMiniFrameDos 68
3.1.3 Processament del mapa de pluja 70
3.1.3.1 Difuminar les zones de pluja 70

3.1.3.2 Calcular les posicions de pluja 72
3.1.4 Generació d'atles 77
3.1.4.1 Generació de l'atles de gotes amb llum puntual................................ 78
3.1.4.2 Generació de l'atles de gotes amb llum ambiental 80
3.2 Pluja en temps real.. 83
3.2.1 Classes creades 85
3.2.1.1 Classe MyApplication 85
3.2.1.2 Classe MyFrameListener.. 87
3.2.1.3 Classe CalculProfunditat 94
3.2.1.4 Classe PramatresShader... 96
3.2.2 Materials 98
3.2.2.1 Profunditat 98
3.2.2.2 Pluja en temps real.. 100
3.3 Visualització de pluja realista.. 108
3.3.1 Transformació dels punts a billboards... 108
3.3.1.1 Tractament d'un billboard d'Ogre des de la GPU 110
3.3.2 Càlcul de les textures.. 116
3.3.2.1 Càlcul dels angles 116
3.3.2.2 Càlcul del centre del billboard des del vertex shader 118
3.3.2.3 Assignació de textures 122
3.3.3 Redimencionar el tamany dels billboards... 129
3.3.3.1 Fórmula per redimensionar la mida del billboard 130
3.3.4 Visualització amb diferents punts de llum a l'escenari 131
3.3.5 Aportació del píxel shader.. 133
3.3.5.1 Tractament de la visualització segons el tipus de punt de llum........ 138
4 Capítol 4 -Resultats i conclusions- 146
4.1 Resultats 148
4.2 Conclusions 161
5 Capítol 5 -Treballs futurs- 164
6 Bibliografia.. 166
7 Annex 169

Capítol 1

-Introducció-

1.1 Presentació del problema

La crescuda del nivell tecnològic de l'última dècada ha fet que cada vegada tinguem entre nosaltres més vídeo consoles al nostre voltant. Cada vegada més sovint les companyies de creació de consoles milloren els seus productes, per tant recíprocament s'han de crear nous vídeo jocs per als clients, els quals, segons nombrosos estudis, confirmen el gran interès que tenen pel món dels videojocs. Pràcticament a cada casa es disposa d'algun aparell per passar-hi unes hores entretingudes, i si no és així el regal més desitjat sempre sol ser una consola o un pc per poder jugar amb vídeo jocs. I és que el mercat de vídeo jocs cada vegada s'amplia més, ja que ara si que es pot dir que les consoles o els videojocs en general són tant pels grans com pels petits, ja que aquestes companyies han aconseguit crear videojocs per a tots els públics. Per tant, el que és evident, és que el mercat del videojoc en els últims anys ha fet una crescuda espectacular.

Figura 1.1, imatge d'un carrer on hi plou.

A causa d'aquesta gran crescuda de la demanda del videojoc, cada vegada els programadors o treballadors o artistes del videojoc intenten millorar el realisme dels seus jocs. Això sovint provoca que les CPUs quedin molt saturades. Per tant, cada cop més s'intenta repartir el treball de la CPU amb el hardware gràfic que permet ajudar a la CPU amb certes tasques.

La simulació de pluja en temps real és un tema de gran importància pel món dels videojocs i la realitat virtual, però fins ara només es limitava a una aproximació barroera mitjançant polígons semitransparents. Això trenca la sensació d'immersió del jugador. Clarament, es necessita una solució més realista basada en una simulació més precisa del fenomen de la pluja.

1.2 Motivació del projecte

L'objectiu d'aquest PFC és desenvolupar un sistema de pluja per a videojocs i aplicacions de realitat virtual que sigui acurat, tant en el sentit del realisme visual com el seu comportament.

A conseqüència de l'esmentat problema de la saturació de la CPU degut al gran nombre de processos que ha de gestionar, l'objectiu és que el comportament i la visualització d'aquest sistema de pluja sigui controlat totalment per hardware gràfic i no per software a la cpu repartint així les tasques a realitzar.

1.3 Abast

El sistema de pluja es desenvoluparà per a la seva incorporació en un conegut motor de videojocs (Ogre3D). A més a més, es pretén optimitzar la seva eficiència aprofitant les característiques del hardware gràfic actual.

Per això s'han desenvolupat dos subsistemes clarament diferenciats:

1. Sistema d'edició de pluja: aquest serà un programari que permetrà al dissenyador definir, per a una escena, les àrees on plourà i amb quina intensitat. Aquest mòdul generarà, doncs, la informació necessària per la segona part de l'aplicació, ja que proporcionarà a aquesta les posicions exactes on ha de deixar caure cada gota d'aigua.

2. Sistema de pluja en temps real: aquest és el mòdul que s'executarà al videojoc, permetent al jugador percebre una pluja realista. Per això es llegiran les dades del mòdul anterior i es processaran a la tarja gràfica en dos àrees diferents:

   a. Control del moviment de les gotes d'aigua mitjançant l'ús de vèrtex shaders.

   b. Control de l'aparença de les gotes mitjançant un píxel shader.

Tot això implica una programació extremadament eficient del hardware actual, la qual cosa exigeix optimitzar i fer servir les tècniques més extremes que es disposen a l'actualitat.

1.4 Eines utilitzades

Aquest projecte s'ha programat amb c++, s'ha creat un script de materials per Ogre3D que s'ha programat amb un llenguatge molt semblant al C++.

Eines utilitzades en el sistema d'edició de pluja:

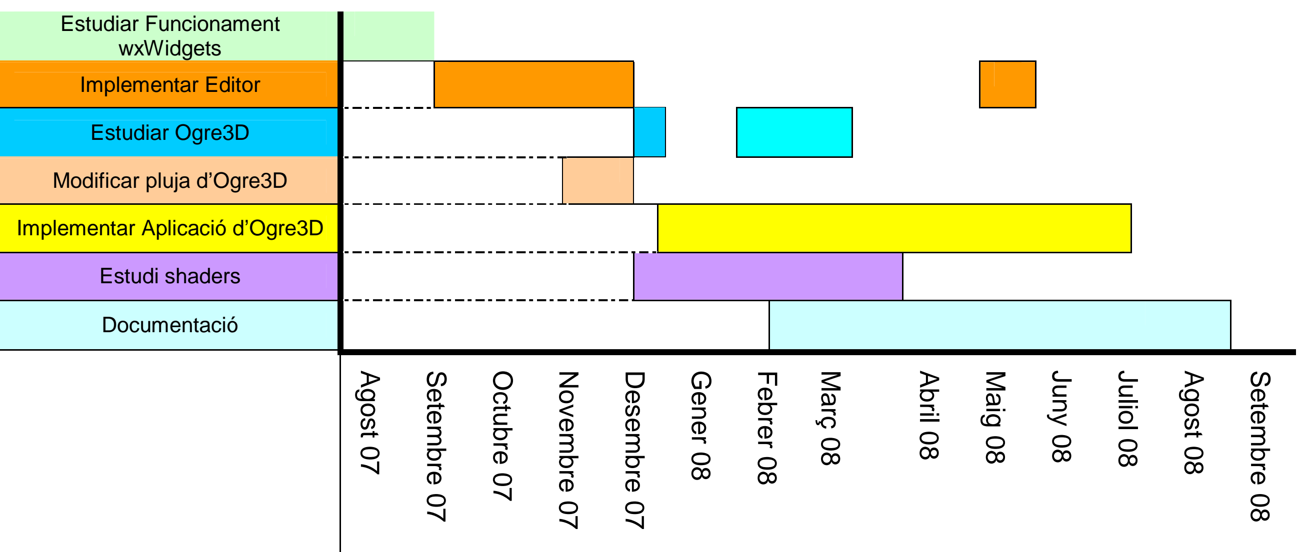
- Compilador Visual C++ 8.
- Llibreria wxWidgets 2.8.7 per crear el sistema de finestres.
- Llibreria Devil Image i FreeImage per a treballar amb imatges.
- Llibreria Mersenne Twister per a generar números aleatoris.

Eines utilitzades per a l'aplicació en temps real:

- Compilador Visual c++ 8.
- DirectX 9.0 SDK, la actualització del novembre del 2007.
- Cg 2.0 per poder compilar els vèrtex i píxels shaders.
- S'ha utilitzat el motor gràfic en 3D OGRE 1.4.7 Eihort.

1.5 Planificació

Aquesta és la planificació que s'ha seguit alhora d'elaborar aquest projecte:



- **Estudiar funcionament wxWidgets:** Com es pot veure al gràfic aquesta va ser la primera tasca a realitzar. Es van estudiar varis exemples de la qual disposa la llibreria per poder agafar alguna

idea interessant. Un cop es va tenir un coneixement prou ampli es va procedir a començar la implementació.

- **Implementar Editor:** Com es pot observar al gràfic es veuen dues parts a les quals es va treballar amb l'editor. La primera part, es va basar en aconseguir que a l'editor funcionessin totes les funcions bàsiques, és a dir: Carregar i mostrar una imatge per pantalla, guardar la imatge, poder dibuixar les figures desitjades per l'usuari i aconseguir-les guardar etc. Com es veu, ja a les acaballes del projecte es va tornar a modificar la versió de l'editor, ja que es va afegir la possibilitat de poder difuminar la zona de pluja i la de calcular la posició final de les gotes d'aigua a l'Ogre.

- **Estudiar L'OGRE:** Durant un llarg procés només es va realitzar l'aprenentatge del funcionament d'Ogre3D, des de la seva instal·lació per aconseguir el funcionament òptim de la llibreria fins a aconseguir dominar i/o entendre alguns aspectes d'aquest motor 3d. Un cop aconseguits els fonaments previs, es va entrar en un procés de proves i, un cop finalitzades, es va buscar informació per poder aplicar als nostres shaders a l'aplicació.

- **Modificar sistema de pluja de l'Ogre3D:** OGRE ja disposa dels seus sistemes de partícules, entre ells disposa del de pluja. Per contra, abans de començar a crear la nostra aplicació definitiva, es van observar els resultats que s'obtindrien al modificar el sistema de partícules d'Ogre3D per poder aconseguir fer caure les gotes d'aquest a una zona aproximada a la desitjada. Per tant, en aquest procés es van crear certs algorismes per aconseguir fer aquest sistema de partícules al lloc desitjat, aquests algorismes més tard podrien ser aprofitats (amb les modificacions pertinents) per la implementació final.

- **Estudi Shaders:** Un dels objectius d'aquest projecte era que la pluja estigués creada a partir de vèrtex i píxels shaders. A causa d' això primerament es va haver d'aprofundir en el concepte i amb el funcionament d'aquests. En concret, es va observar el funcionament del llenguatge Cg. Com es pot veure en el gràfic, s'ha compartit el temps d'estudi dels shaders en una zona en què es creava la implementació d'ogre i fins i tot en l'última part d'implementació de l'editor, i és que s'anaven fent proves de certs shaders amb l'Ogre3D o fins i tot amb les primeres versions del shader destinat a ser el shader de pluja (observant si, per exemple, les posicions generades per l'editor eren utilitzades realment i de forma correcta pels shaders que s'estaven començant a construir).

- **Implementar l'aplicació en temps real:** Un cop realitzades les proves al modificar o ampliar el sistema de partícules de l'Ogre,

es va poder començar a implementar l'aplicació. Cal dir que, en realitat, mai s'ha deixat de fer un estudi sobre el motor 3D, ja que en moltes ocasions, al anar veient certs problemes d'implementació, sempre s'aprenen coses. L'objectiu en aquell moment ja era aconseguir els objectius marcats inicialment.

- **Documentació:** Quan es va acabar la primera part de l'editor, es va iniciar a estructurar la documentació. Es van començar a explicar les classes creades en l'editor i els conceptes nous adquirits. Com es veu al gràfic, un cop explicada la part creada de l'editor, es va començar a documentar de manera paral·lela amb els estudis realitzats i a les implementacions realitzades.

1.6 Estructura de la documentació

Capítol 1. Introducció: En aquest primer capítol es presentarà el projecte, és a dir es presentarà la problemàtica que ens ha impulsat a crear el present projecte. Es presenten els objectius que s'han marcat, s'explica quin ha estat el treball realitzat en cada període de temps des de que es va començar a realitzar el treball fins al seu final, així com l'estructura del present document.

Capítol 2. Fonaments Previs: Es podrà observar conceptes de les actuals GPUs programables, per al que serveixen i com han afectat a les noves implementacions. Es podrà veure l'estudi dels diferents motors 3D existents i finalment en aquest segon capítol també es pot veure alguns temes que s'han destacat per poder entendre el funcionament bàsic del motor 3D escollit OGRE així com alguns dels objectes i classes més destacades, utilitzades i importants d'aquest motor. Gràcies a aquests fonaments, que es poden adquirir en aquest capítol, es facilitarà la comprensió del funcionament del projecte i el perquè de certs aspectes realitzats en l'aplicació en temps real que s'ha creat.

Capítol 3. Disseny i implementació: En aquest capítol de la documentació del projecte s'explicarà detalladament com s'han creat les aplicacions, és a dir les classes que s'han hagut de crear, els paràmetres d'aquestes, els mètodes, i els diferents algorismes que es van crear al llarg dels diferents documents de programació. S'explicarà els objectius que es compliran al realitzar cada classe i cada mètode implementat.

Capítol 4. Resultats i conclusions: S'explicarà i es mostrarà els resultats obtinguts de les aplicacions realitzades, a partir d'aquestes, s'exposaran les diferents conclusions a que s'ha arribat.

Capítol 5. Treballs futurs: En aquest darrer capítol es podrà observar els treballs futurs o millores que es poden realitzar sobre aquesta aplicació. Simplement s'exposaran les idees que es tenen relacionades en el tema, per en un futur poder actualitzar el treball realitzat.

Capítol 2

-Fonaments Previs-

En aquest capítol s'explicaran els principals treballs realitzats durant les primeres etapes de la realització d'aquest projecte, a més a més d'alguns conceptes necessaris per comprendre alguns capítols posteriors. A continuació es presentaran els conceptes de la programació de GPU de les targetes gràfiques actuals. També es mostrarà l'anàlisi de l'entorn de la programació gràfica conegut com motors gràfics i l'estudi del motor escollit, Ogre3D.

Finalment, ja pensant més concretament l'objectiu final del projecte, s'analitzarà el sistema de pluja el qual ja porta creat Ogre3D i també un estudi de la física de la pluja.

2.1 Conceptes de programació de la GPU

En aquest apartat explicarem les nocions bàsiques de la programació de la GPU (Graphics Processing Unit) de les targetes gràfiques modernes. Aquests conceptes bàsics seran importants a l'hora d'entendre com es creen i perquè es creen els vèrtexs i píxels shaders (concepte explicat en el pròxim apartat) per a la pluja en temps real.

2.1.1 Descripció del procés de renderització

En l'actualitat, les targetes gràfiques disposen de tres unitats que permeten incorporar elements programables a les etapes de transformació dels vèrtexs i de la transformació final del color de cada píxel *(divisió de la imatge més petita possible en la pantalla de l'ordinador)* rasteritzat a la pantalla.

El processador, el qual està capacitat d'executar les diferents transformacions programables de la geometria, s'anomena **Vèrtex Shader**. I el processador capacitat per executar les transformacions de colors programables als píxels abans de ser pintats en la pantalla se'ls anomena **PíxelShader,** nosaltres no treballarem amb els geometry shaders. Aquest conjunt de transformacions es codifiquen amb programes denominats shaders. A la figura 2.1 s'observa en quines etapes del procés de renderització són utilitzats el Vèrtex Shader i el Píxel Shader.

Durant el procés de renderització, la CPU és l'encarregada d'enviar la geometria i les textures a la GPU per realitzar les operacions de pintat dels polígons a la pantalla. La GPU s'encarrega de desempaquetar cada vèrtex, i a continuació el vèrtex shader realitza les operacions de transformació i càlculs de la il·luminació programades pel desenvolupador. Aquestes operacions són realitzades per a cada vèrtex i les seves dades associades (com les coordenades de textura) de tota la geometria de la escena.

A continuació, després de transformar els vèrtexs de coordenades de món a coordenades de pantalla, com es mostra a la figura 2.2 (a), els vèrtexs són combinats per formar primitives de visualització, convertint-se en punts, línies o polígons, segons especifiqui el desenvolupador.

Després de la generació de les primitives es realitza el procés de retallament (clipping), que consisteix en l'eliminació de parts d'una primitiva o grups de primitives que estan fora del volum de treball, anomenat també frustum.

Posteriorment, les primitives passen a ser rasteritzades, convertint-se en píxels amb un color associat de com es mostra a la figura 2.2(b). Durant el procés de rasterització es realitza la interpolació 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 es coneix amb el nom de fragments.

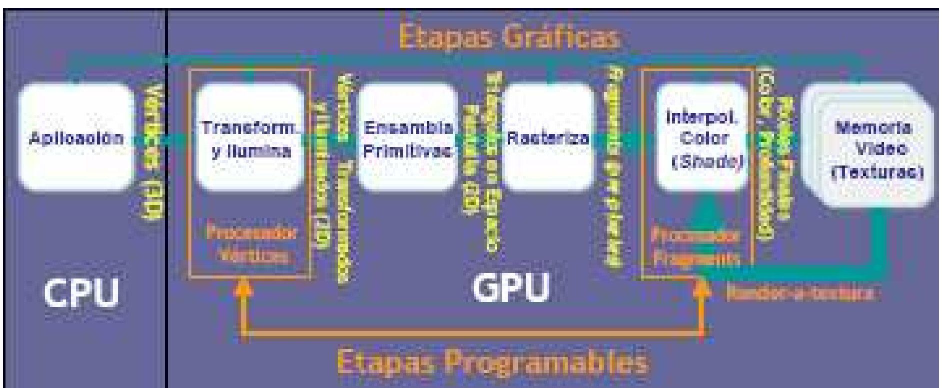


Figura 2.1, esquema general del procés de renderització.

Un fragment és un píxel amb opcions a ser pintat a la pantalla. Cada fragment és processat per la unitat programable anomenada Píxel Shader. El color de cada fragment pot ser modificat pel desenvolupador, programant les transformacions desitjades i utilitzant les dades interpolades associades al fragment i les textures especificades pel desenvolupador. A la figura 2.2 (c), es mostra les transformacions de color que es realitzen a les primitives de la cara del personatge. Aquestes transformacions han estat programades específicament per obtenir el resultat desitjat. Un cop finalitzada la transformació del fragment, aquest és pintat en el búffer de renderització (per exemple la pantalla).

Les noves capacitats de programació, incloses en dues de les etapes del procés de renderitzaciò, juntament amb l'enorme potència de les noves targetes gràfiques, proporciona una gran versatilitat, permetent així la implementació de noves tècniques gràfiques que permeten obtenir uns resultats gràfics molt més realistes i eficaços durant la visualització.

2.1.2 Detalls del procés de renderització

Durant el procés de renderització, els objectes geomètrics són processats per pintar-los a la pantalla. A l'apartat anterior es va fer una descripció de cada una de les etapes de la renderització. Es va comentar, sense entrar en detalls, que els vèrtexs de la geometria podien tenir informació associada.

A continuació detallarem quina informació i format poden tenir. Un model geomètric preparat per utilitzar-la en una visualització interactiva ha de disposar

obligatòriament per cada un dels seus vèrtexs de les coordenades de posició i el seu vector normal.

Els programes preparats per transformar la geometria en el vèrtex shader, o aplicar transformació de color en el píxel shader, esperen rebre com a paràmetres d'entrada aquestes dades dels vèrtexs o les primitives geomètriques a renderitzar, respectivament.

Cada paràmetre d'entrada té una definició semàntica que especifica la informació que inclou el paràmetre en qüestió. En el quadre de la figura 2.3 es mostren algunes de les definicions de l'estil existent i la seva semàntica per cada tipus de dades incloses als vèrtexs de la geometria (*veure figura 2.3*). Aquests paràmetres són utilitzats per multitud d'operacions diferents, ampliant la possibilitat de la programació dels vèrtexs i píxels shaders.

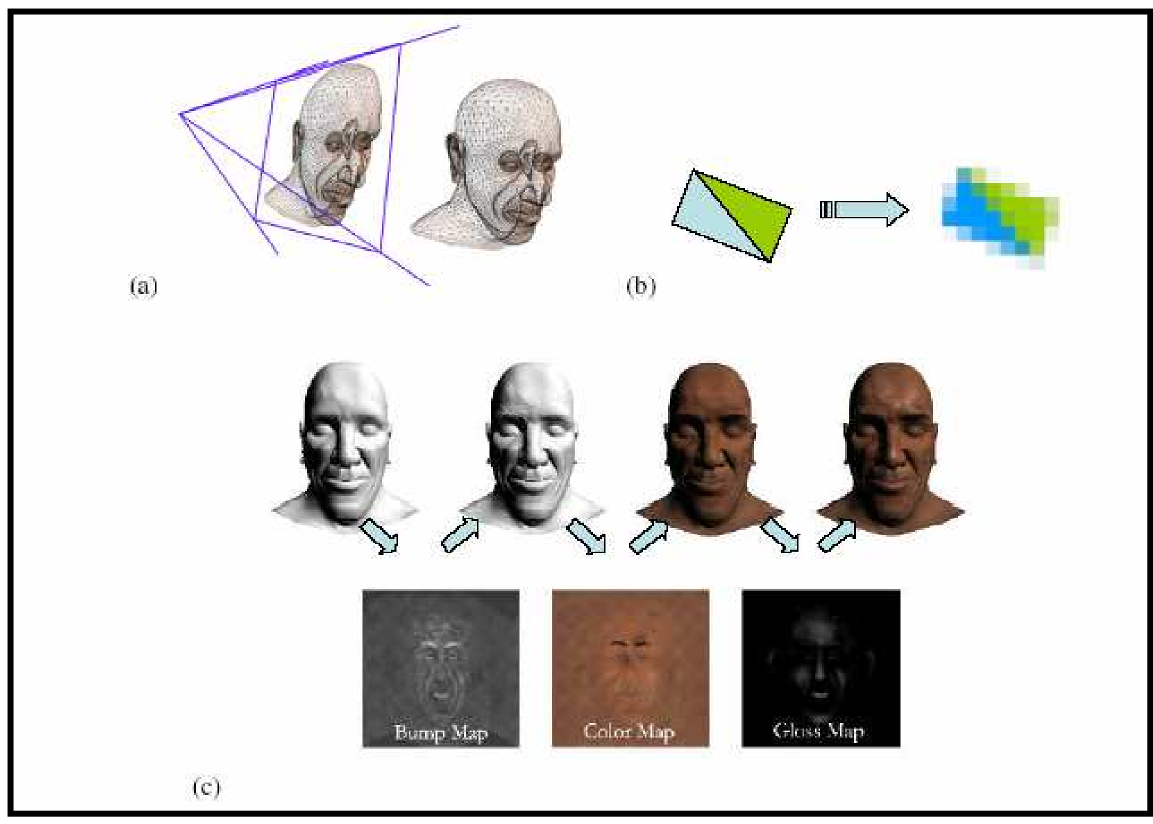


Figura 2.2. Representació dels processos de renderització.

A més a més d'aquests paràmetres que s'han vist, és possible passar paràmetres d'entrada uniformes (constants). El mateix valor del paràmetre serà utilitzat per la transformació de tots els vèrtexs o primitives de la geometria d'un objecte a l'escena.

Parámetros de entrada incluidos en la geometría			
Semántica	**Tipo**		**Descripción**
POSITION	float3	Obligatorio	Coordenadas de la posición del vértice.
NORMAL	float3	Obligatorio	Coordenadas del vector normal del vértice.
COLOR0	float3	Opcional	Color asignado al vértice.
COLOR*i*	float3	Opcional	Color i-ésimo del vértice.
COLOR7	float3	Opcional	...
TEXCOORD0	float4	Opcional	Coordenadas de textura del vértice.
TEXCOORD1	float4	Opcional	...
TEXCOORD*i*	float4	Opcional	Coordenadas de textura i-ésimo del vértice.
TEXCOORD7	float4	Opcional	...

Figura 2.3. Taula de paràmatres d'entrada a la GPU.

2.1.3 Llenguatges de programació per GPU

En l'actualitat existeixen diferents llenguatges de programació per GPU. Els més utilitzats són:

- **Cg** (C for Graphics): és un llenguatge semblant al llenguatge de programació c, amb la mateixa sintaxi per les declaracions, crides a funcions i la majoria dels seus tipus de dades. No obstant, té algunes millores i modificacions per facilitar la codificació de programes que es compilen amb codi assemblador de GPU optimitzat. Aquest llenguatge compatible amb les dos APIs gràfiques més importants OpenGl i Direct3D. Ha estat desenvolupat per Nvidia.

- **GLSL**: és el llenguatge de programació Standard de la API OpenGl Shanding Language, i a l'igual que el cg es basa en el llenguatge C. Ha estat desenvolupat per l'OpenGl ARB per a la programació gràfica per GPUs.

- **HLSL**: és el llenguatge de programació Standard de la API direct3D desenvolupat per microsoft, basat també en llenguatge C. És pràcticament idèntic al llenguatge Cg, a causa que Microsoft i Nvidia varen col·laborar per aconseguir la creació de Cg.

Aquests llenguatges de programació per GPUs tenen petites diferències entre ells. No obstant, tots presenten aproximadament el mateix tipus de funcionalitats. No existeixen grans avantatges o inconvenients per la utilització d'un o de l'altre. En aquest projecte de final de carrera s'ha treballat amb el

llenguatge Cg. En el capítol d'estudi de l'Ogre3D es veurà com es poden integrar els programes desenvolupats en Cg al motor 3D d'Ogre3D.

2.1.4 Anàlisis de motors gràfics

Per la programació de visualitzacions interactiva és necessari desenvolupar un conjunt de components que permetin gestionar una escena en tres dimensions formada per objectes geomètrics, llums i efectes d'animacions. Dintre de les operacions de gestió d'una visualització interactiva s'inclouen elements com, per exemple, el control de la navegació de l'usuari per l'escena, l'actualització de la posició dels objectes dins de l'escena, els càlculs de la seva il·luminació, i les operacions de renderització per pintar-los a la pantalla. El conjunt de components que realitzen aquestes tasques s'inclouen en un marc comú anomenat motor gràfic. Aquí veiem alguns dels motors gràfics més destacats:

- Nebula
- OpenSceneGraph
- Ogre3D
- Fly3D
- Crystal Space

Els motors gràfics amb tres dimensions més potents tècnicament són Nebula, OpenSceneGraph, Ogre3D i Fly3D. Nebula i Crystal Space s'aproximarien més al que s'entén com motor de videojocs, oferint altres aspectes tècnics generals com els events de so, i la física dels objectes amb tres dimensions de l'escena.

S'ha observat que Fly3D és un motor de jocs complert, fàcil d'utilitzar, però enfocat per desenvolupaments amb objectius educacionals, sense proporcionar la potència gràfica oferta per Ogre3D, Nebula i OpenSceneGraph. En canvi Nebula i Crystal Space han demostrat ser motors de jocs complerts, potents i amb llicencies compatibles. Però s'ha observat en aquests dos últims que el codi font era bastant desestructurat i amb escassa documentació.

S'ha observat que Ogre3D i OpenSceneGraph són els dos motors estrictament gràfics, i tècnicament amb les millors característiques per aprofitar les possibilitats de programació per GPU de les noves targetes gràfiques.

S'ha observat que tots dos estan ben estructurats. No obstant, el disseny del sistema de OpenSceneGraph és més complex, sense ser més complert, i menys documentat que el de Ogre3D. A més s'ha observat que les actualitzacions apareixen més sovint amb el motor Ogre3D. Finalment un dels factors claus per acabar escollint el motor Ogre3D, ha estat que aquest disposa d'una important comunitat d'usuaris amb els quals tens la possibilitat de discutir o simplement demana'ls-hi algun consell sobre algun aspecte d'aquest motor3D. A la figura 2.4 podem veure imatges de jocs, simulacions o simplement treballs realitzats amb Ogre3D:

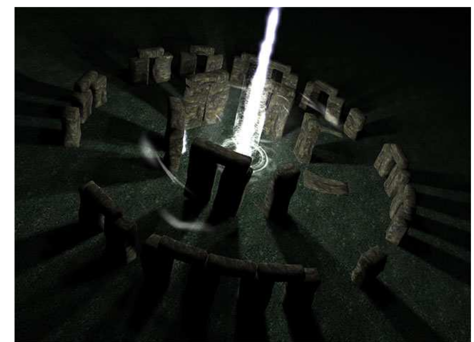


Figura 2.4, exemples de proyectes realitzats amb Ogre3D.

2.2 Estudi del motor gràfic Ogre3d

OGRE: Object Oriented Graphics Redering Engine, és un motor 3D escrit en c++, creat per ajudar als desenvolupadors a crear més fàcilment les seves aplicacions com videojocs, visualització d'arquitectures, simulacions, o qualsevol altra cosa que necessiti una visualització en 3 dimensions.

La biblioteca permet abstreure tots els detalls de les classes subjacents de Direct3D i OpenGL, i proporciona una interfície amb els objectes globals i altres classes necessàries per desenvolupar un videojoc.

OGRE és disponible sota la llicència de GNU Lesser General Public License (LGPL). Aquesta llicència a grans trets diu que tu pots obtenir de manera gratuïta i utilitzar el codi d'OGRE per qualsevol cosa, modificar-lo, afegir-hi codi etc, només amb la condició que sempre que el distribueixis facis públic aquell codi.

2.2.1 Objectes principals d'Ogre

Tot seguit es mostraran i s'explicaran els objectes més bàsics d'Ogre3D. Començarem amb un diagrama UML(*figura 2.5*):

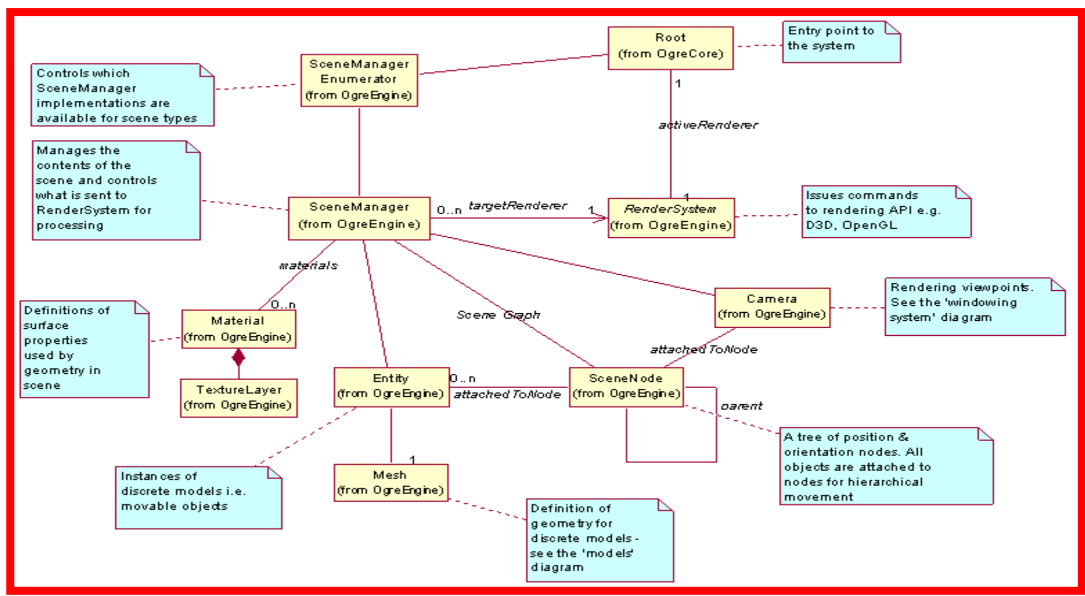


Figura 2.5. Diagrama UML de Ogre3D

Ara, amb més detall, s'explicarà cada un d'aquests objectes:

Objecte "root"

Root és el primer objecte que s'ha de crear, és el més important de tots, ja que sense ell és impossible fer res. Per tant aquest ha de ser el primer objecte a crear i l'últim a destruir. El més lògic de fer és fer una instància de root des d'un membre de l'objecte encarregat de posar en marxa l'Ogre3D, que per tant serà creat tant aviat com l'aplicació sigui executada, i serà destruït quan l'aplicació finalitzi.

Aquest objecte permet configurar com es vulgui que sigui el sistema. Bàsicament permet configurar el mateix que permet configurar la finestra que apareix al executar una aplicació d'OGRE (*figura2.6).* Aquesta finestra normalment s'executa si l'objecte root crida el mètode showConfigDialog() i permet configurar entre altres coses:

- La resolució de l'aplicació
- El color de fons
- Opció de pantalla complerta

A més a més, és un objecte que ens permetrà obtenir punters d'altres objectes com els de SceneManager i RenderSystem i altres de resource managers (*veure a baix*).

Figura 2.6, menú inicial d'Ogre3D

Root també té altres funcions més avançades, com per exemple la possibilitat d'actualitzar tots els objectes de renderització, molt utilitzat en

videojocs i demos, gràcies al mètode startRendering. Aquest mètode quan és cridat, entra en un bucle continu de renderització que només s'aturarà quan l'aplicació sigui aturada o quan, amb un frameListener, sigui aturat amb un mètode que ho permet.

Objecte RenderSystem

RenderSystem és una classe abstracta que defineix la interfície per sota de la API 3d. És l'encarregat d'enviar totes les renderitzacions a la API i també l'encarregat d'enviar totes les opcions d'interpretació a aquesta. La classe és abstracta, ja que totes les aplicacions són renderitzades per una API específica, com pot ser la Direct3D o bé l'OpenGl. Després que el sistema hagi estat inicialitzat per l'objecte root (root:: initialise), es podrà obtenir l'objecte RenderSystem amb el mètode root:: getSystemRenderSystem().

Objecte SceneManager

Apart de l'objecte root, aquest és segurament l'objecte més important de tots, l'objecte més utilitzat. Bàsicament s'encarrega de guardar i organitzar tots els objectes que després s'han d'enviar per renderitzar (llums, objectes (entitats), materials, serveix per crear i fer servir les càmeres, etc), és a dir que el que fa l'objecte SceneManager és controlar tots els objectes que han de representar l'escena del mon creat en Ogre. SceneManager guarda els noms de tots els objectes de l'escena (inclús l'escena i les càmeres) per així poder accedir més tard a ells i poder fer amb ells el que es desitja. Per exemple, es pot accedir a una càmera amb el mètode: getCamera.

SceneManager també s'encarrega d'enviar l'escena a l'objecte RenderSystem al moment en què aquesta s'ha de renderitzar.

Objecte ResourceManager

La classe ResourceManager simplement és una classe base per moltes altres. La seva funció és carregar recursos a altres classes. Aquests recursos poden ser dades (textures, meshes, mapes, etc, les quals són una subclasse de ResourceManager, anomenades per exemple TextureManager i MeshManager) que necessita l'Ogre per executar-se correctament.

ResourceManager assegura que els recursos només es carreguin una vegada i siguin compartits per totes les parts del motor d'Ogre. També permet buscar els recursos en un determinat número d'ubicacions, inclosos múltiples "path" i fins i tot arxius comprimits.

Per tant és lògic que normalment no s'interactuï gens amb aquest objecte, només es podrà interactuar amb ell quan es vulgui afegir alguna direcció perquè pugui buscar algun determinat recurs. Per aconseguir-ho hi ha l'opció de cridar el mètode estàtic: ResourceManager::addCommonSearchPath o ResourceManager::addCommonArchive, que farà que busqui en aquesta direcció i/o arxiu que s'acaba d'afegir a la llista.

RenderTarget

Finestra la qual té la possibilitat de rebre operacions de renderització de totes les classes target.

Sistema de coordenades d'Ogre3D

Per posicionar objectes al món 3D es necessita conèixer el sistema de coordenades amb el qual treballa Ogre3D, és aquest:

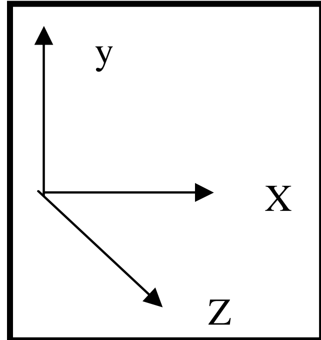


Figura 2.7, sistema de coordenades d'Ogre3d

Objecte Mesh

Un objecte Mesh representa un model 3D. S'utilitzen per representar objectes (entitats). Són organitzats des de la classe MeshManager i tenen el format .mesh.

Objecte Entitat (Entitie)

Una entitat és una instància a un objecte que està a l'escena. Pot ser un cotxe, una persona, una casa, una farola, un sistema de partícules, una càmera. Per tant no ha d'estar necessàriament en una posició fixa del mon que s'ha creat.

Les entitats estan basades en les meshes. Moltes entitats poden estar creades per la mateixa mesh, podem tenir tantes copies d'aquell objecte com es vulgui. Es podrà crear una entitat cridant el mètode createEntity de la classe SceneManager, al qual se li haurà de donar un nom per diferenciar de les altres, i la mesh en la qual estarà basada aquesta entitat (per exemple: casa.mesh).

Encara que s'hagi creat l'entitat no s'aconseguirà el que volem fins que no es lligui aquest objecte a la classe sceneNode, la qual és responsable de guardar les dades (vectors de posició, posició envers al mon etc.) de l'entitat que s'ha lligat a ella. Si es vol modificar la posició de l'entitat creada, s'haurà d'entrar al sceneNode per fer aquest canvi i no directament a la entitat, com podria semblar.

Objecte Càmera i Light

Sovint, quan s'explicava la utilitat dels anteriors objectes, ha sortit l'objecte càmera, ja que és un altre dels objectes clau en OGRE (i segurament també en tots els altres motors 3D). La seva funcionalitat és clara, permet veure tot el que s'ha creat a l'escena. L'objecte Càmera té moltes semblances amb l'objecte que s'ha esmentat explicant les entitats, SceneNode, ja que quan s'ha creat una càmera, aquest objecte disposa de mètodes com setPosition(), entre altres que també utilitzen a SceneNode. Tant és així que es podria lligar varis objectes SceneNode i Càmera entre ells, formant una estructura d'arbre, llavors la posició de cada objecte és sempre relativa a la del seu parent (veure figura 2.7).

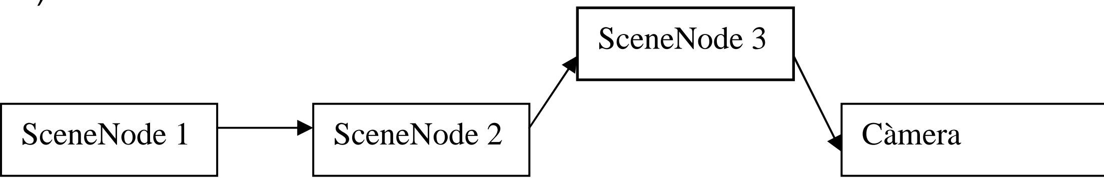


Figura 2.7

Per contra del que es podria pensar relacionant l'objecte càmera a una càmera real, és que no es crearan vàries càmeres, per exemple una per veure una zona de l'escena i una altra càmera per veure l'altra zona. El que es fa és que es crearien diferents SceneNodes, i que els col·locaríem a les posicions desitjades i faríem servir una única càmera. Quan es vulgui veure una zona de l'escena, la càmera estaria a una posició en concret i quan es volgués veure una altra part de l'escena la lligaríem al SceneNode adequada. Per aconseguir triar l'SceneNode al que es vol lligar la nostra càmera, es farà gràcies al FrameListener, que es veurà després.

L'objecte llum també es comporta de manera molt semblant a les càmeres, i per tant, es pot crear de la mateixa manera i també col·locar-ho allà on es desitgi o també lligar-ho amb un SceneNode. A l'Ogre hi ha 3 tipus diferents de llums:

1- Punt de llum (LT_POINT), el qual com el seu nom indica es posarà un punt de llum a l'espai i il·luminar uniformement en totes direccions(veure figura 2.8).

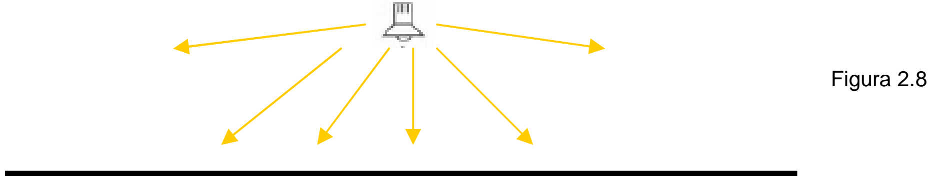


Figura 2.8

2- Spotlight (LT_SPOTLIGHT), un spotlight per exemple es podria comparar amb la llum que fa un lot, que al centre té una intensitat de llum molt forta, però que a mida que ens distanciem del centre, perd intensitat (veure figura 2.9).

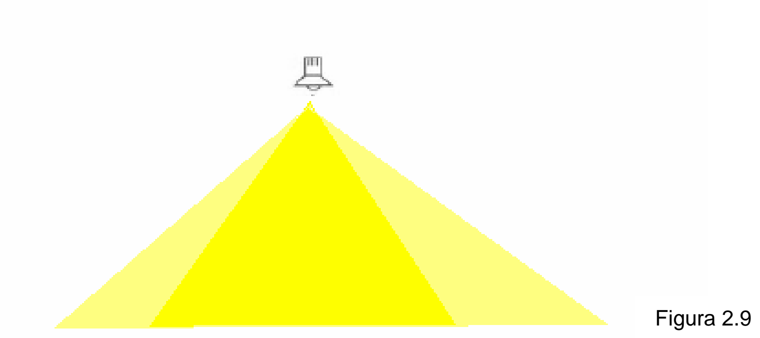


Figura 2.9

3- Direccional (LT_DIRECTIONAL), com indica el seu nom, es tracta un llum que defineix una direcció definida (veure figura 2.10).

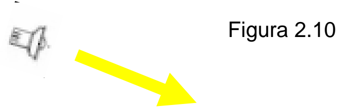


Figura 2.10

FrameListeners

FrameListener es diu que és una classe “escoltadora”. Aquestes tipus de classes el que fan és esperar rebre un senyal per després cridar els seus mètodes. Per tant el FrameListener és una classe que crida als mètodes quan es produeixi alguna senyal de frame. Normalment, el que sol fer és crear una subclasse del FrameListener per després modificar els mètodes de que disposa la classe per aconseguir la fita desitjada. Llavors per què s’utilitzi aquesta classe s’haurà de cridar el mètode de l’objecte root::addFrameListener, passant com a paràmetre un objecte de la subclasse creada.
La classe FrameListener només disposa de dos mètodes:

- bool frameStarted (const FrameEvent &evt);
- bool frameEnded(const FrameEvent &evt);

Com indica el nom, frameStarted és cridat cada vegada que el frame s’està apunt de renderitzar. Retorna un booleà, on cert voldrà dir que es continuï renderitzant, i fals voldrà dir que s’aturi la renderització i, per tant, el bucle de renderització. FrameEnded és cridat cada vegada que ha acabat la renderització d’un frame. Si retorna cert, es prepararà per renderitzar el pròxim frame, si retorna fals, s’acabarà el bucle de renderització.

Per acabar aquest apartat, a continuació es mostra un exemple basat, sobretot, en la utilització dels frameListeners, però en el que es mostra com s'utilitzen els diferents objectes explicats anteriorment.

Veiem el codi del següent exemple:

```
class TutorialFrameListener : public ExampleFrameListener
 {
 public:
     TutorialFrameListener( RenderWindow* win, Camera* cam, SceneManager
*sceneMgr )
         : ExampleFrameListener(win, cam, false, false)
     {
         mMouseDown = false;
         mToggle = 0.0;

         mCamNode = cam->getParentSceneNode( )->getParentSceneNode( );
         mSceneMgr = sceneMgr;

         mRotate = 0.13;
         mMove = 250;
     }
...
```

Es crea la subclasse, heretant de ExampleFrameListener que és una classe que hereta de frameListener i keyListener.

```
     bool frameStarted(const FrameEvent &evt)
     {
         mInputDevice->capture();

         bool currMouse = mInputDevice->getMouseButton( 0 );
```

Començament del mètode frameStarted explicat fa un moment. Veiem que es captura l'estat del teclat i del ratolí (getMouseButton(0) especifica el botó esquerra del ratolí).

```
         if ( currMouse && ! mMouseDown )
         {
             Light *light = mSceneMgr->getLight( "Light1" );
             light->setVisible( ! light->isVisible() );
         }
```

CcurrMouse=cert si s'ha premut el ratolí (tecla de l'esquerra). Farem visible aquesta llum cada vegada que cliquem amb el ratolí.

```
if ( (mToggle < 0.0f) && mInputDevice->isKeyDown( KC_2 ) )
          {
              mToggle = 1.0f;

              mCamNode->detachObject( mCamera );
              mCamNode = mSceneMgr->getSceneNode( "CamNode2" );
              mSceneMgr->getSceneNode( "PitchNode2" )->attachObject( mCamera
);
          }   return ! mInputDevice->isKeyDown( KC_ESCAPE );
      }
```

Mtoogle controla que no hi hagi rebot al clicar amb el ratolí, per tant fins que mToggle no sigui inferior a zero no s'accedirà al condicional que s'observa al codi. Aquí es veu que si s'apreta la tecla 2, la càmera es deslliga del sceneNode que estava i es lliga amb el sceneNode “PitchNode2”, permetent així veure una altra part de l'escena tal i com s'havia explicat a l'apartat de l'objecte càmera.

```
protected:
     bool mMouseDown;       // Whether or not the left mouse button was
down last frame
     Real mToggle;          // The time left until next toggle
     Real mRotate;          // The rotate constant
     Real mMove;            // The movement constant
     SceneManager *mSceneMgr;   // The current SceneManager
     SceneNode *mCamNode;   // The SceneNode the camera is currently
attached to
 };
class TutorialApplication : public ExampleApplication
 {
//falta el codi del constructor i el destructor.
 protected:
     void createCamera(void)
     {
         // create camera, but leave at default position
         mCamera = mSceneMgr->createCamera("PlayerCam");
         mCamera->setNearClipDistance(5);
     }

     void createScene(void)
     {
         mSceneMgr->setAmbientLight( ColourValue( 0.25, 0.25, 0.25 ) );

         // add the ninja
         Entity *ent = mSceneMgr->createEntity( "Ninja", "ninja.mesh" );
         SceneNode *node = mSceneMgr->getRootSceneNode()-
>createChildSceneNode( "NinjaNode" );
         node->attachObject( ent );
```

Aquí es pot veure com es crea una entitat a la qual se li posarà el nom de Ninja i serà representat per la mesh: ninja.mesh.

```
        // create the light
        Light *light = mSceneMgr->createLight( "Light1" );
        light->setType( Light::LT_POINT );
        light->setPosition( Vector3(250, 150, 250) );
        light->setDiffuseColour( ColourValue::White );
        light->setSpecularColour( ColourValue::White );
```

Es crea un llum de tipus LT_POINT.

```
        // Create the scene node
        node = mSceneMgr->getRootSceneNode()->createChildSceneNode(
"CamNode1",
Vector3( -400, 200, 400 ) );

        // Make it look towards the ninja
        node->yaw( Degree(-45) );

        // Create the pitch node
        node = node->createChildSceneNode( "PitchNode1" );
        node->attachObject( mCamera );

        // create the second camera node/pitch node
        node = mSceneMgr->getRootSceneNode()->createChildSceneNode(
"CamNode2", Vector3( 0, 200, 400 ) );
        node = node->createChildSceneNode( "PitchNode2" );
```

Lliguem la càmera al PitchNode1 i tot seguit definim el PitchNode2 per poder-lo canviar després al prémer la tecla 2.

```
    void createFrameListener(void)
    {
      mFrameListener = new TutorialFrameListener(mWindow, mCamera, mSceneMgr);
       mRoot->addFrameListener(mFrameListener);

       // Show the frame stats overlay
       mFrameListener->showDebugOverlay(true);
          }
  };
```

Es crea l’objecte de la subclasse del frameListener que s’ha creat i s’afegeix als frameListeners amb el mètode root addFrameListener (*Per veure el codi complert: http://www.ogre3d.org/wiki/index.php/Basic_Tutorial_4*).

RenderTargetListener

Serveix per rebre events de RenderTargets.

2.2.2 Materials a l’Ogre

L’objecte material permet establir uns paràmetres per cada objecte que s’ha col·locat a l’escena per així controlar certes particularitats de cada objecte. Per exemple permet establir el grau de transparència de cada objecte, el resplendor, la brillantor, com les textures són filtrades, la reflexió que tindrà amb els llums que rebi i molts altres efectes més.

Hi ha dos maneres de crear els materials, o bé programant directament amb c++, per exemple cridant el mètode: SceneManager:: createMaterial i a partir d'aquí cridant molts altres mètodes de la classe material, com per exemple (entre molts altres):

- void setShininess(Real val)
- void setAmbient(Real red,Real green, Real blue)
- void setTransparencyShadows(bool enabled)
- bool isTransparent (void) const

O bé definint el material en un script per poder-lo carregar en temps d'execució.

a) Creació d'un material amb programació directa, afegir un material de la manera descrita a dalt o també es pot fer carregant directament una mesh, la qual carregarà propietats de l'objecte Material. Sempre que s'hagi definit un nou material al SceneManager, a causa de que moltes propietats d'aquest material no hauran estat definides, Ogre posa automàticament aquestes propietats per defecte. Les propietats que posa per defecte són aquestes:

- Ambient reflectance = ColourValue::White (full)
- Diffuse reflectance = ColourValue::White (full)
- Specular reflectance = ColourValue::Black (none)
- Emmissive = ColourValue::Black (none)
- Shininess = 0 (not shiny)
- No texture layers (& hence no textures)
- SourceBlendFactor = SBF_ONE, DestBlendFactor = SBF_ZERO (opaque)
- Depth buffer checking on
- Depth buffer writing on
- Depth buffer comparison function = CMPF_LESS_EQUAL
- Culling mode = CULL_CLOCKWISE
- Ambient lighting in scene = ColourValue(0.5, 0.5, 0.5) (mid-grey)
- Dynamic lighting enabled
- Gourad shading mode
- Solid polygon mode
- Bilinear texture filtering

A partir d'aquí el programador podrà fer tots els canvis que vulgui, ja que disposa dels mètodes de la classe material per fer-ho.

b)Definir un material en un script, normalment aquesta és la forma més còmoda de crear materials, ja que es pot crear un material complex on es vulgui, però quan s'ha aconseguit crear ja es tindrà el material guardat en un fitxer, i després simplement s'haurà de cridar per utilitzar-lo, i lògicament es podràs fer tants cops com es desitgi.

Format de l'script

Per definir un material en un script s'han de seguir unes normes (com amb tot llenguatge). Per escriure el codi del material simplement es farà servir un llenguatge molt semblant al c++ amb alguns delimitadors al principi i al final de cada material ({ }), i també dispondrem de les "//" per fer comentaris. Aquí hi ha un exemple:

```
// Aquí podem comentar tot el que volguem
material walls/funkywall1 //nom del material, ha de ser únic
{
//possibilitat de col·locar 4 atributs
    technique
    {
        // passem els paràmetres desitjats
        pass
        {
            ambient 0.5 0.5 0.5
            diffuse 1.0 1.0 1.0

            // Texture unit 0
            texture_unit
            {
                texture wibbly.jpg
                scroll_anim 0.1 0.0
                wave_xform scale sine 0.0 0.7 0.0 1.0
            }
            // Texture unit 1
            texture_unit
            {
                texture wobbly.png
                rotate_anim 0.25
                colour_op add
            }
        }
    }
}
```

Just després de definir el material, a la línia de sota (després del "{"), hi ha la possibilitat de col·locar 4 diferents paràmetres:

- **Lod_distance:** Aquest atribut controla la distància entre les diferents tècniques que poden utilitzar-se.

- **receive_shadows:** Que un objecte faci ombra o no. Depèn d'una combinació de factors importants, però aquest atribut controla si aquest material que s'ha definit podrà crear ombres o no. Aquest atribut només serà útil pels objectes sòlids, ja que pels materials amb transparència es considera que no poden tenir ombra i per tant activar aquest atribut seria una incoherència. Aquest atribut, per defecte, estarà activat.

- **transparency_casts_shadows:** Normalment, els objectes amb transparència no haurien de crear ombra, ja que en principi no

bloquejarien el pas de la llum, però si es necessita crear una certa ombra per algun objecte amb transparència s'haurà d'activar aquest atribut que per defecte estarà en "off" (apagat).

- **set_texture_alias**: Aquest atribut simplement permet associar un nom a un nom d'imatge, veiem un exemple:

```
material exemple/exemple
{
set_texture_alias mapa mapa.png
}
```

Com es pot veure, ara, per treballar amb la imatge mapa.png, només caldrà anomenar-la mapa i ja en tindrem suficient.

Technique

Un material pot estar compost per vàries tècniques, tot i que, com es veu en el de l'exemple, només n'utilitza un. S'utilitzen diferents tècniques per aconseguir al detall el resultat desitjat.

Pass

És una simple crida a la API amb uns determinats paràmetres que serveixen com a propietats de renderitzacio pel material que s'està creant.

Cada tècnica pot tenir de 1 a 16 atributs diferents, i per diferenciar-los se li poden assignar un nom, però és opcional. Si no se li assigna un nom, ell mateix assigna un "0" al primer, un "1" al segon i així successivament. Com passa amb els noms dels materials, els noms dels atributs han de ser únics dins de cada tècnica. Si no és així, obtindríem un resultat que no és el desitjat, ja que els dos atributs es combinarien, a causa d'això també s'escriurà un "warning" al ogre.log.

Aquests són alguns dels atributs que es poden utilitzar:

- ambient= Determina les propietats de la reflexió de la llum ambiental sobre l'objecte.
- diffuse= Determina les propietats de la reflexió de la llum difusa sobre l'objecte.
- specular= Determina les propietats de la reflexió de la llum especular sobre l'objecte.
- emissive= Determina la capacitat d'auto il·luminació de que té l'objecte.
- scene_blend= aquest atribut ofereix vàries opcions per mesclar el color de sortida combinant el color de l'escena.
- depth_check= Si aquest atribut està activat, cada vegada que es vulgui renderitzar un píxel, es comprovarà si és a davant dels altres píxels. Si no és així, aquest píxel no es mostrarà, altrament si. Si està desconnectat, els píxels seran escrits sempre.

- depth_write= Cada vegada que escrivim un nou píxel, el buffer de profunditat (depth buffer) serà actualitzat agafant el valor d'aquest. Si aquest atribut està desactivat no s'actualitzarà el búffer de profunditat. Per defecte, aquest atribut està activat, però a vegades pot ser necessari desactivar-lo, ja que per exemple amb un conjunt d'objectes amb transparència de forma que entre ells es superposin, si no es desactivés aquesta opció, no s'obtindria el resultat desitjat.

- depth_func= Aquest atribut funcionarà si el depth_check està activat. Disposa de vuit altres atributs; si el depth_check està activat, compararà el valor de buffer de profunditat amb el que ha de ser escrit i, segons l'altre atribut que col·loquem farà una cosa o altre. Els atributs que té depth_func són pràcticament tots de comparació. Per exemple hi ha el less_equal que escriurà el nou píxel si és més a prop o igual que el píxel que hi ha guardat al buffer de profunditat. Els altres que hi ha son: always_fail, always_pass, less, less_equal, equal, not_equal, greater_equal, greater.

- lighting= Per triar si aquest objecte utilitzarà o no la llum dinàmica.
- shading= Per triar quin tipus de ombres es crearan amb la llum dinàmica. Hi ha tres opcions: flat, gouraud o phong.
- polygon_mode= Amb aquest atribut defineixes com es vol que sigui la trama dels polígons. Poden ser sòlids (la opció per defecte), o es pot definir que només siguin dibuixats amb línies o punts.
- point_size= Per escollir la mida dels punts.

Texture_unit

Un altre atribut que es pot afegir als scripts .material és el texture_unit, que també té molts altres atributs que permeten molt joc. La texture_unit bàsicament permet certes opcions per poder aprofitar imatges en el material que s'està definint.

Aquests són alguns dels atributs que es poden utilitzar dins del texture_unit:

- Texture_alias= Permet posar un nom qualsevol que serà l'equivalent al nom de la imatge.
- Texture= Serveix per posar el nom de la imatge que es vol utilitzar.
- Filtering= Permet filtrar o no la imatge que s'ha passat per paràmetre de la manera que es desitja. Té quatre opcions per triar: no filtrar la imatge, filtratge bilineal, filtratge trilineal i filtratge anisotròpic. Per defecte fa un filtratge bilineal.

Com utilitzar vèrtex i fragments shaders

Si a l'objecte que s'està definint se li vol afegir un vèrtex shader o un fragment shader, aquests s'han de definir. Es poden definir de varies maneres: es pot definir al .material (al pass) o al .program. Si es defineix en el .program es podrà garantitzar que el shader serà definit abans de tots els .materials, i que

per tant qualsevol .material el podrà utilitzar. En qualsevol dels dos casos s'haurà de tenir el .material, el .program i els shaders utilitzats en un dels path que s'han escollit.

Exemple de com definir un vèrtex shader:

```
vertex_program esbos/esbosVp cg
{
	source esbos.cg
	entry_point main_vp
	profiles vs_2_0 arbvp1

	default_params
	{
		param_named_auto modelViewProj worldviewproj_matrix
	}
}
```

En aquest exemple es veu que primer es defineix el nom i el path que tindrà el shader. Entry_point serveix per indicar quin serà el nom de la funció que se li ha definit al shader, ja que en un mateix fitxer hi poden haver vàries funcions diferents, i per tant, definint quin és el nom de la funció que es vol utilitzar ja n'hi haurà suficient. Finalment s'ha de definir quins són els "profiles" en què haurà de compilar el shader. Se'n pot posar més d'un, per exemple es pot posar un profile dels més senzills i un altre que sigui dels més moderns (sempre que siguin acceptats per la targeta gràfica utilitzada), així a l'hora d'executar el programa segons la targeta s'utilitzarà el profile més modern que s'ha entrat si la targeta gràfica el suporta, o el més senzill en cas contrari.

A l'exemple es veu que la part final es fa servir un atribut anomenat: **default_params**. Aquest atribut serveix per passar alguns paràmetres que necessita el shader (ja que així ha estat definit). Per exemple, es veu que el paràmetre que es defineix al .program és del tipus param_named_auto, que és una variable que Ogre anirà actualitzant de manera automàtica segons el valor d'aquest. En aquest cas es passa com a paràmetre el **worldviewproj_matrix**, que és una matriu molt utilitzada, ja que és la matriu de la vista i de la projecció concatenada a l'actual vista del món d'Ogre que s'està renderitzant en aquell moment. Des del .material també es poden passar paràmetres al shader, encara que no s'hagués definit en el .material i sí en el .program. Per aconseguir-ho simplement s'ha d'utilitzar l'atribut vèrtex_program_ref. Després, dins dels corxets, s'hi pot passar tants paràmetres com siguin necessaris.

Aquí es veu en l'exemple:

```
vertex_program_ref myVertexProgram
{
	param_indexed_auto 0 worldviewproj_matrix
	param_indexed      4 float4  10.0 0 0 0
}
```

Per passar paràmetres en un fragment program és exactament el mateix, però variant una mica la capçalera: S'ha de substituir vèrtex_program_ref, per fragment_program_ref, és a dir per exemple la capçalera s'ha de definir així:

```
fragment_program_ref myFragmentProgram
{
}
```

2.2.3 Sistemes de partícules d'Ogre

Un sistema de partícules és un conjunt de partícules (format per un determinat material) que es comporten d'una manera definida, per exemple l'Ogre disposa de varis sistemes de partícules ja definits, com pot ser la pluja, la boira o per exemple la font de colors que explicarem a continuació.

S'ha estudiat els sistemes de partícules d'Ogre3D perquè entre els sistemes de partícules de que disposa l'Ogre3D, un d'ells és el sistema de partícules de pluja. Com es vol estudiar aquest sistema de partícules, per poder entendre el funcionament d'aquest primerament s'ha estudiat el funcionament dels sistemes de partícules d'Ogre3D.

A continuació es podrà observar com es creen aquests sistemes de partícules i per poder-ho observar més fàcilment s'ha posat l'exemple d'una font de colors.

Com si es tractessin de materials els sistemes de partícules d'Ogre estan definits amb scripts. Lògicament seran diferents als del .material, tindran una extensió .particle i tindran els seus propis atributs.

Com es pot veure l'script utilitzat serà del mateix estil, és a dir amb un llenguatge molt similar al c++. Els scripts .particle tenen tres parts ven diferenciades, la primera en què es defineixen els atributs generals, la segona en què defineixen els atributs segons el tipus d'emitter triat i la tercera que es definiran els atributs segons l'affector també seleccionat. Tot seguit s'explicarà el significat i el funcionament d'aquests nous conceptes.

2.2.3.1 Atributs generals

Com s'ha dit per seguir millor com es crea un sistema de partícules s'anirà explicant pas per pas un exemple ja definit per l'Ogre.

Comencem amb el codi que il·lustra la primera part de l'exemple esmentat.

```
Examples/PurpleFountain
{
    material Examples/Flare2
    particle_width 20
    particle_height 20
    cull_each false
    quota 10000
    billboard_type oriented_self
   ...
```

En aquesta primera part només es defineixen uns atributs que seran generals, per exemple la quantitat de partícules que tindrà, quina rotació tindrà cada billboard, el punt de renderització de cada partícula etc etc.

Aquí es veuen alguns dels atributs que es poden utilitzar després de definir el nom del sistema de partícules:

- Quota= total de partícules.
- Material= material utilitzat.
- Particle_width= Defineix l'amplada que ocuparà aquest sistema.
- Particle_height= Defineix l'alçada que ocuparà aquest sistema.
- Billboard_type= Defineix el tipus de billboard que s'utilitzarà

En aquest cas s'ha definit que el material utilitzat és el de Examples/Flare2 *(veure figura 7)*, la qual per fer-nos una idea al veure el resultat final (*veure figura 2.11)*.

Figura 2.11, textura utilitzada al material Flare2.

2.2.3.2 Emitters

Veiem el codi que corresponent a aquesta segona part del sistema de partícules estudiat:

```
...
    emitter Point
    {
        angle 15
        emission_rate 75
        time_to_live 3
        direction 0 1 0
        velocity_min 250
        velocity_max 300
        colour_range_start 1 0 0
        colour_range_end 0 0 1
    }
...
```

Com es pot veure en aquest exemple, en la segona part dels sistemes de partícules caldrà definir quin tipus d'emitter s'utilitzarà ja que n'hi ha varis per

escollir i cadascun amb els seus atributs particulars, aquests emitters definiran on es creen les partícules d'aquest sistema i per on es mouran.

L'atribut emitter, disposa de 5 diferents atributs, però que bàsicament es poden agrupar en:

- El primer grup seran els dels objectes que heretaran de la mateixa classe, la classe **areaEmitter**, i que per tant defineix el sistema de partícules que s'emetrà en una àrea determinada, els quatre objectes que formen aquest grup són:
    - Box Emitter
    - Cylinder Emitter
    - Ellipsoid Emitter
    - Hollow Ellipsoid Emitter
    - Ring Emitter

Que les diferències que tindran aquests seran en la zona que podran definir que actuï el sistema de partícules definit.

- L'altre objecte que no pertany a aquest grup és el **PointEmitter**, que es diferencia d'aquest grup, ja que defineix que el sistema de partícules només afecti en un determinat punt. Aquest és el diagrama de classes que forma el ParticleEmitter*(figura 2.12):*

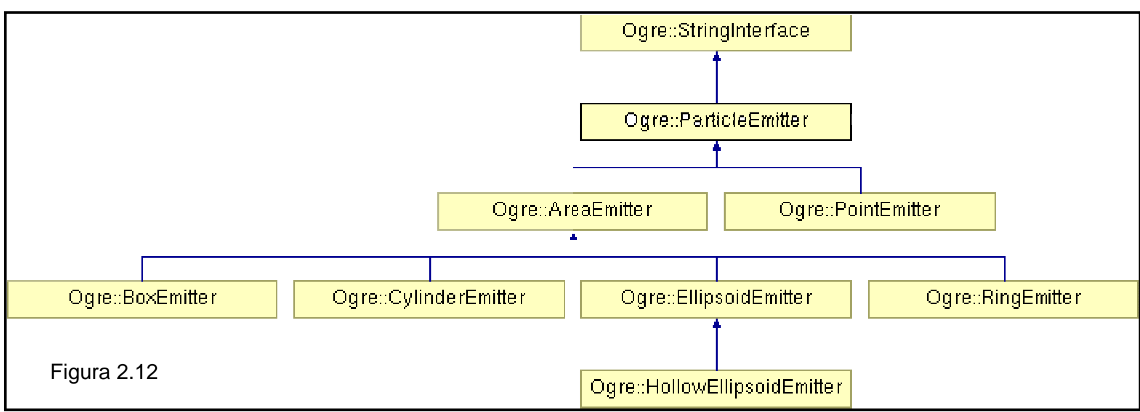


Figura 2.12

2.2.3.3 Affector

La tercera part, com s'ha dit és l'afector. Els seus atributs tots serveixen per modificar les partícules. Podem veure el diagrama de classes que deriva del ParticleAfector a la figura 2.13.

Com es pot interpretar a partir del diagrama de classes i de les classes derivades del particleAffector (figura 2.13), gràcies a aquests objectes es podran canviar el color de les partícules, canviar el moviment d'aquestes, canviar l'escala etc.

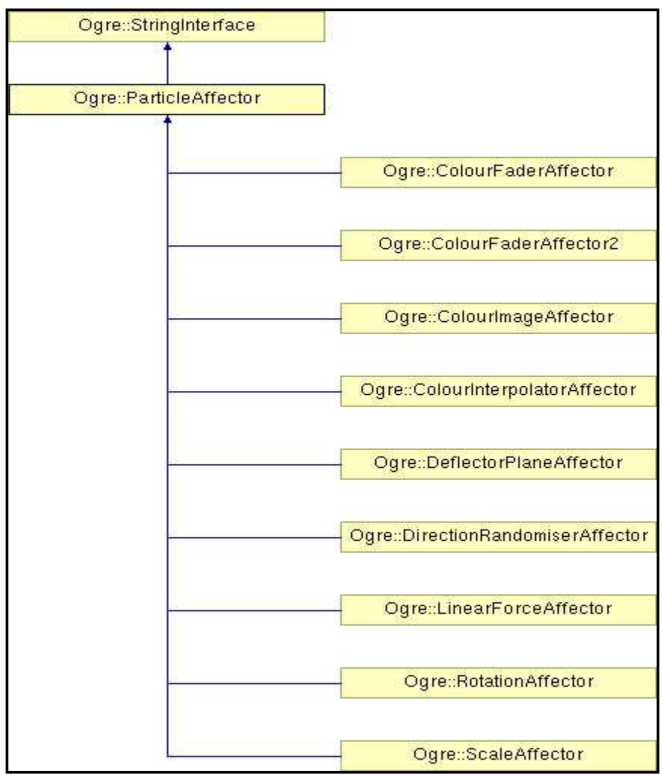


Figura 2.13, diagrama de classes del particleAfector

A la última part de l'exemple que es mostra a continuació, es pot apreciar que fa servir dos tipus de affectors, el linealForce el qual veiem que aplica una força de -100 a l'eix de les y, per tant anirà de dalt cap a baix (recordem el sistema de coordenades de la figura 2.6 del capítol: 2.2.1).

```
    affector LinearForce
    {
        force_vector      0 -100 0
        force_application add
    }
    // Fader
    affector ColourFader
    {
        red -0.25
        green -0.25
        blue -0.25
    }
}
```

El Com es veu també utilitza el colourFade per establir el color de les partícules.

Imatge del resultat d'aplicar l'exemple del sistema de partícules (veure figura 2.14):

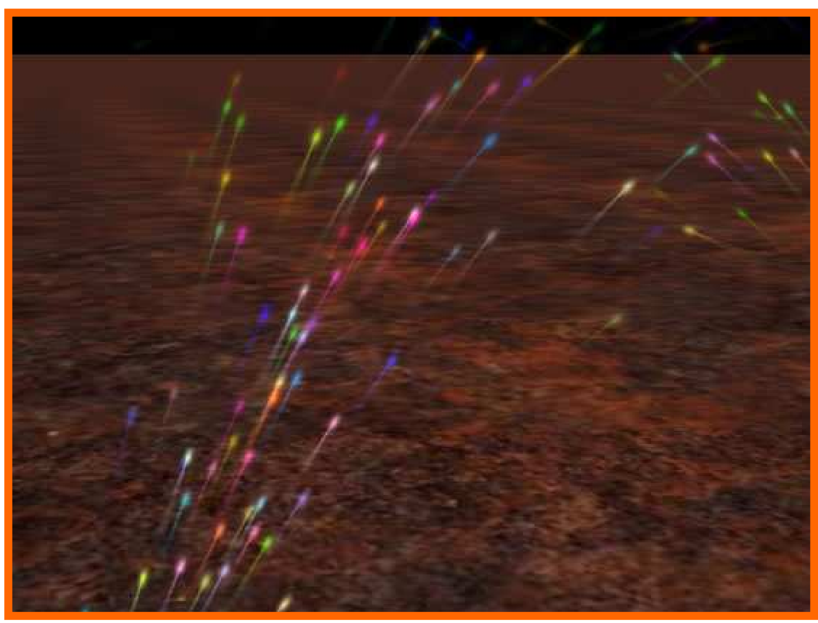


Figura 2.14

2.2.4 Simulació de pluja a l'Ogre

OGRE ja disposa des de fa temps d'un script que simula la caiguda de gotes d'aigua, està creat totalment per software (no utilitza ni vèrtex ni fragments programs), ja que utilitza les classes que s'han mostrat abans, és a dir les particleEmitter, areaEmitter etc. Aquí veiem l'script del sistema de pluja d'Ogre.

```
Examples/Rain
{
   material         Examples/Droplet
   particle_width   20
   particle_height  100
   cull_each        true
   quota            10000
   billboard_type   oriented_common
   common_direction 0 -1 0
...
```

Com es pot veure utilitza el material Droplet, mostrat més avall. Defineix una amplada de les partícules de 20 i una alçada de 100, defineix l'atribut cull_each com a cert i un total de partícules (quota) de fins a 10000. Defineix que les partícules seran del tipus oriented_common, és a dir que estaran orientades cap un mateix punt, d'aquí que tingui de definir aquest punt. Això ho fa al fer common_direction, el qual defineix que sigui 0 -1 0, per tant la direcció serà de dalt cap a baix ja que segueix l'eix de les y.

Veiem la segona part:

```
...
    emitter Box
    {
        angle           0
        emission_rate   100
        time_to_live    5
        direction       0 -1 0
        velocity        50
        colour_range_start  0.3 1 0.3
        colour_range_end    0.7 1 0.7
        width           1000
        height          1000
        depth           0
    }
...
```

Veiem que s'aplica un emitter de tipus Box. Pel que fa als atributs definits podem dir que les gotes tindran un angle 0, per tant seran perpendiculars al terra, emission_rate defineix el total de partícules que es creen per segon, en aquest cas són 100. Defineix que el temps de vida d'aquestes gotes és de 5 segons, i la velocitat d'aquestes de unitats/s. Defineix el color amb el qual començaran al principi de ser creades i el color amb el que acabaran. Finalment defineix que la zona de pluja serà de 1000 per 1000 en els eixos x i z amb les coordenades d'Ogre.

Tercera i última part de codi del .particle:

```
    affector LinearForce
    {
        force_vector      0 -200 0
        force_application add
    }
```

Veiem que s'aplica un affector de tipus LinearForce i la força que se li aplicarà serà en direcció de –y i el valor d'aquesta serà de 200. El force_application add defineix que serà una força, que intentarà representar el d'una acceleració.

Com s'ha vist aquest sistema de partícules defineix que el material que utilitzarà serà el de Examples/Droplet, aquest n'és el codi corresponent:

```
material Examples/Droplet
{
      technique
      {
      pass
      {
      scene_blend colour_blend
      depth_write off

      texture_unit
      {
      texture basic_droplet.png
      }
      }
      }
}
```

En aquest material únicament s'especifica que l'scene_blend serà de tipus colour_blend, això vol dir que el color de l'escena serà el resultat del reflexa dels colors d'entrada d'aquests, però que no serà fosc. Depth_write serà of ja que per contra no es renderitzarien les gotes del sistema de partícules.

Aquí es mostra la imatge Droplet.png, utilitzada en el material (veure figura 2.15):

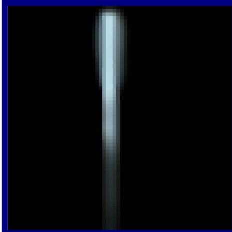


Figura 2.15

Aquest és el resultat d'utilitzar el codi que s'ha vist, és a dir el del sistema de pluja d'Ogre (veure figura 2.16):

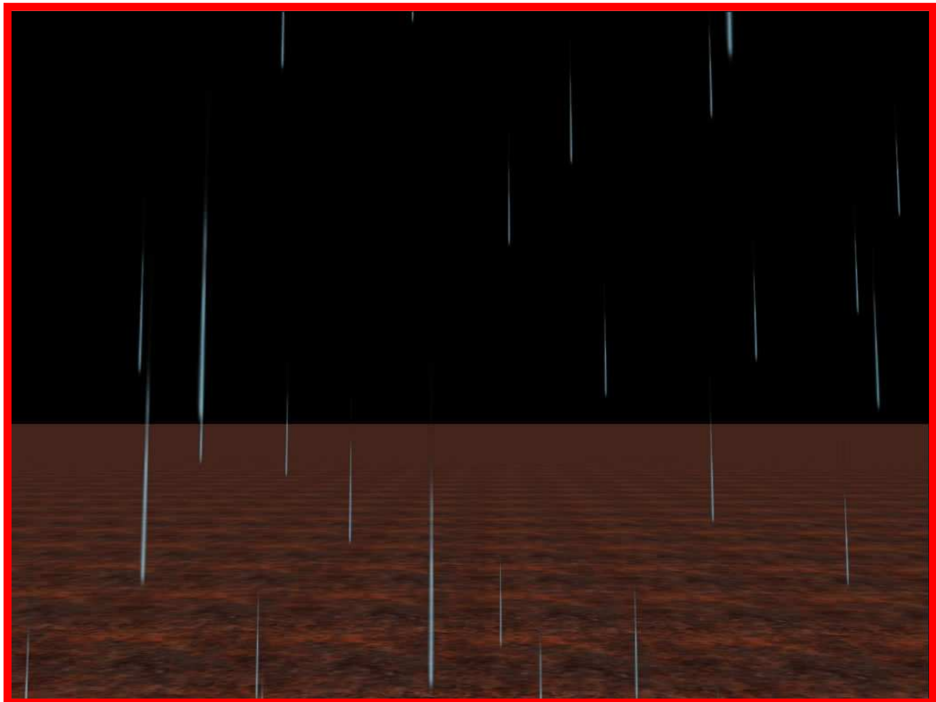


Figura 2.16

2.3 Física de la pluja

Per a realitzar aquest projecte prèviament s'ha hagut de fer un estudi sobre la física de la pluja, és a dir, com actuen les gotes d'aigua al ploure. Sobretot ens hem fixat en el moviment d'aquestes, ja que s'havia de decidir quina fórmula de moviment li aplicàvem a les gotes del nostre sistema de pluja en temps real. Per tant s'ha estudiat quin és el comportament (velocitat i/o l'acceleració) de la caiguda d'objectes per una atmosfera, i finalment en concret amb la caiguda de gotes d'aigua.

Un objecte que està caient per l'atmosfera està lligada a dos forces. La primera és la força gravitacional, expressada com el pes de l'objecte, i la segona són les forces de resistència que realitza l'aire, les quals seran el fregament de l'objecte *(veure vectors de força en la figura 2.16)*.

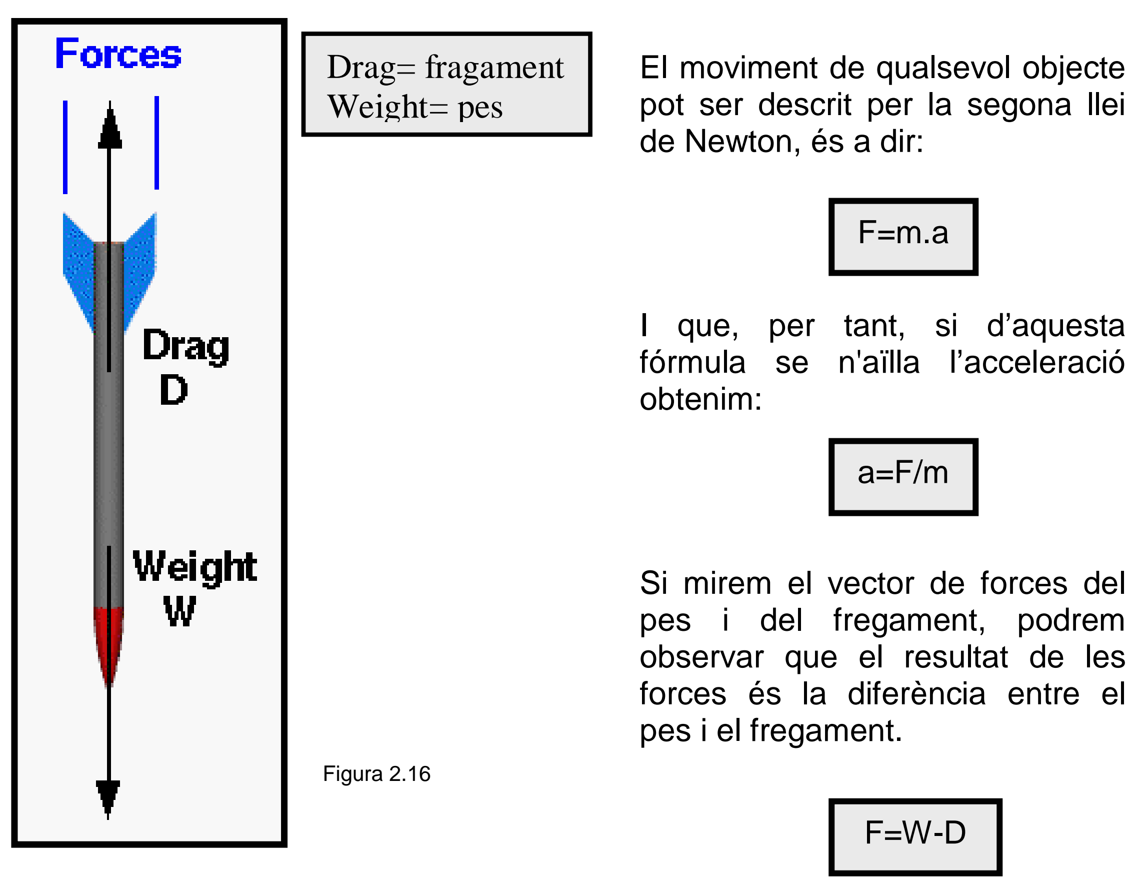


Figura 2.16

El moviment de qualsevol objecte pot ser descrit per la segona llei de Newton, és a dir:

$$F=m.a$$

I que, per tant, si d'aquesta fórmula se n'aïlla l'acceleració obtenim:

$$a=F/m$$

Si mirem el vector de forces del pes i del fregament, podrem observar que el resultat de les forces és la diferència entre el pes i el fregament.

$$F=W-D$$

Per tant al substituir els valors obtinguts podem determinar una fórmula que definirà l'acceleració:

$$a=(W-D)/m$$

El fregament **D** és donada per una fórmula coneguda, en la qual es veu que el fregament depèn d'un coeficient de fregament **cd**, la densitat atmosfèrica **r**, i al

quadrat de la velocitat de l'aire **V**, i alguna referència sobre l'àrea de l'objecte **A.** Aquesta en seria la fórmula:

$$D=cd*r*v^2*A/2$$

La densitat augmenta al quadrat de la velocitat. Per tant, l'objecte cau, i ràpidament les condicions del fregament s'igualen a les del pes. Quan el fregament és igual al pes, ja no existeix la xarxa de vectors de forces, ja que les dos forces existents, fregament i pes, s'han igualat i l'acceleració vertical és zero. Sense acceleració, l'objecte cau a una velocitat constant, tal com explica la primera llei de Newton :

1a llei de Newton: Si sobre un cos no actua cap força o la resultant de forces que actuen és nul·la, el cos està en repòs o es mou amb moviment rectilini i uniforme.

La velocitat constant vertical és anomenada **velocitat terminal.**

Podem determinar el valor d'aquesta velocitat terminal:

$$D=W \qquad W=cd*r*v^2*a/2$$

Obtenim la fórmula aïllant la velocitat:

$$V=sqrt(2W/(cd.r.A))$$

Amb les gotes d'aigua de la pluja, passa exactament el mateix, és a dir, la velocitat de caiguda de les gotes anirà augmentant fins que la resistència de l'aire (que farà de fregament) sigui igual a l'atracció de la gravetat. A partir d'aquell moment, la gota cau amb la velocitat que queia quan s'han igualat les forces. Per tant, a partir d'aquest moment caurà amb velocitat constant, és a dir tindrà una velocitat terminal. Normalment les gotes d'aigua de la pluja tenen un mida de 2 mil·límetres, i amb les fórmules que s'han trobat s'ha arribat al resultat que amb aquest mida les gotes baixen a una velocitat terminal de 6.5m/segon.

2.4 Funcions de densitat de probabilitat (PDF) i de distribució acumulatives (CDF)

Com es veurà en l'apartat 3*.1.3.2 calcular les posicions de pluja*, per distribuir la pluja s'ha utilitzat les funcions de densitat de probabilitat (PDF) i de distribució acumulatives (CDF), per així aconseguir distribuir les gotes de manera aleatòria però imposant que certes gotes tinguin una probabilitat més elevada de caure en la zona escollida per l'usuari. Aquestes funcions, a grans trets, permeten això i seguidament s'explicaran els seus fonaments matemàtics.

Funcions de densitat de probabilitat

Per distribucions discretes, la PDF és la probabilitat d'observar un resultat en concret. Per distribucions contínues, la probabilitat d'observar un valor particular és zero. Per obtenir aquestes probabilitats s'ha d'integrar sobre un interval d'interès, per exemple la probabilitat de que un número aleatori sigui igual a un interval entre 1 i 2, és la integral de la funció PDF apropiada des de 1 fins a 2.

Una funció PDF té les dos propietats teòriques següents:

- La funció és positiva o zero per qualsevol resultat.
- La integral d'una funció PDF sobre el seu rang complert de valors ha de ser u.

La funció PDF serà f(x) i està definida per:

$$f(x_i) = P(X = x_i)$$

Funcions de distribució acumulatives

La funció de distribució acumulativa (CDF) serà F(X) i està definida per:

$$F(x) = P(X \leq x)$$

Per tant podem veure que el sumatori de totes les f(x) serà igual a F(x), que és el mateix que dir que el sumatori de totes les PDF és igual al CDF, per tant podem dir que les dues funcions són equivalents:

$$F(x) = \sum_{x_j \leq x} f(x_j)$$

Quan parlem de X estarem parlant d'una variable contínua quan agafa valors reals, o d'algun subconjunt d'aquest.

La funció de densitat de probabilitat de X estarà definida per:

$$P(a \leq X \leq b) = \int_a^b f(x)\, dx$$

Si f és una funció de densitat de probabilitats per una variable aleatòria X, la funció de distribució acumulativa o CDF associada anomenada F és:

$$F(x) = P(X \leq x) = \int_{-\infty}^{x} f(t)dt$$

Per tant podem dir que les dos funcions són equivalents. La funció estarà definida així:

$$f(x) = \frac{dF(x)}{dx}$$

Considerem que f serà contínua. Per tant, ja que defineix una probabilitat, ha de ser positiva per tota x. A la figura 2.17 es pot veure el gràfic que la representa.

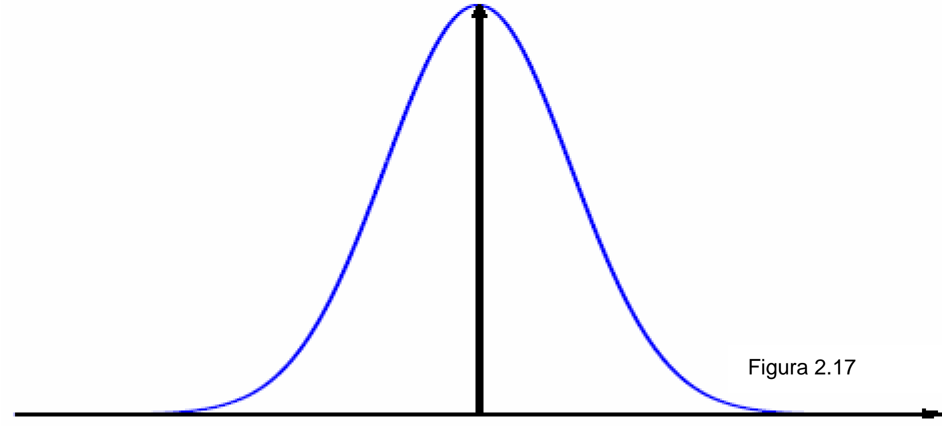


Figura 2.17

Considerem que F(X), és contínua. Donat que F(x) mesura la probabilitat de que X estigui per sota de x, es pot assegurar que F(x) =0 si x és - ∞ i F(x)=1 si x és ∞. Aquesta és la gràfica que la representa:

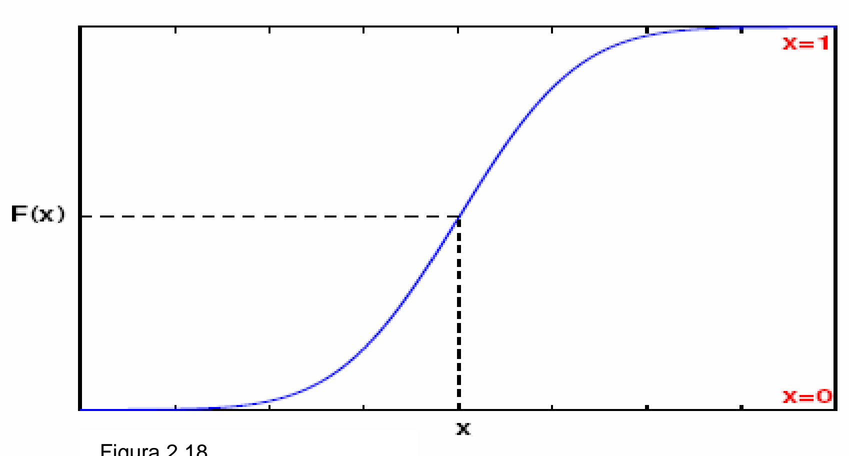


Figura 2.18

2.5 Renderització de gotes d'aigua de pluja

2.5.1 Idea General

La renderització de gotes d'aigua de pluja amb els efectes de la llum i els diferents punts de vista és un problema obert. Les gotes d'aigua, durant la seva caiguda, pateixen distorsions constantment: És el fenomen anomenat com oscil·lacions. Degut a aquestes oscil·lacions, la reflexió de la llum sobre, i la refracció a través de la gota, produeixen durant la caiguda de la pluja complexos patrons de brillantor dins del moviment de la gota d'aigua capturat per la camera o el mateix ull humà. El patró de brillantor de les gotes de pluja normalment inclou petites taques i contorns brillants. En aquest projecte, nosaltres farem servir un nou model d'aparença de les gotes d'aigua de pluja que captura les complexes interaccions entre la direcció de la llum, la direcció de l'observador i la forma oscil·lant de la caiguda [Kshitiz Garg and Shree K. 2006]. Els autors tenen mesurades l'aparença de les gotes de pluja sota un rang de condicions de llum i vistes, les quals seran els paràmetres que determinaran les oscil·lacions més comuns en les gotes d'aigua. Utilitzant aquests paràmetres, tenen renderitzats centenars de gotes d'aigua amb les quals van crear una base de dades que captura les diferents aparences que pot agafar la gota d'aigua segons quina sigui la direcció de la llum i de l'observador. En aquest PFC hem desenvolupat un algorisme de renderitació en temps real que utilitza la base de dades de gotes d'aigua per afegir pluja a una simple imatge o una captura de vídeo amb moviment dels objectes.

2.5.2 Motivació

La pluja sovint és utilitzada en moviment i en animacions per expressar l'estat d'ànim de l'escena. Per exemple, en les pel·lícules "Seven" i "The Matrix Revolutions" la pluja va ser utilitzada per ressaltar a algunes escenes una sensació de malestar. La filmació d'escenes de pluja és, no obstant, una tasca difícil i cara que requereix tenir en funcionament aspersors i diferents punts de llum en una àmplia zona. El rodatge d'una simple escena amb pluja es pot allargar varis dies. Degut als alts costos, és poc pràctic incloure escenes de pluja en pel·lícules amb pressupostos reduïts. Per aquestes raons, un algorisme per renderitzar pluja foto realista és molt desitjable, i encara més si és en temps real per poder incloure-ho en un videojoc. A més a més proporcionaria a un creador de cine o videojocs, un control visual de la pluja.

2.5.3 Treballs Previs per la visualització de pluja

S'han desenvolupat molts mètodes per renderitzar pluja, alguns dels quals estan disponibles en softwares comercials com Maya, 3d Studio Max i Inferno. Aquests mètodes utilitzen sistemes de partícules [Reeves 1983; Sims 1990] per simular amb un grau de realisme els aspectes de resolució i la distribució de les gotes de pluja. No obstant, els programaris de renderització de pluja estan

limitats ja que utilitzen uns models fotogràfics per renderitzar l'aparença de la pluja molt senzills. Molt sovint, les gotes són formes simples, com rectangles o el·lipses, i la brillantor de cadascuna de les gotes és constant. Els mètodes de renderització de pluja que no utilitzin sistemes de partícules han estat desenvolupats[Starik and Werman 2003], [Langer et al. 2004], [Wang and Wade 2004], normalment, amb l'objectiu de reduir el cost computacional als aspectes de representació i distribució de la pluja. Tot i així, com en els mètodes basats en sistemes de partícules, aquests utilitzaven models de gotes d'aigua amb formes molt senzilles. Com són molt simples, aquests només es poden utilitzar quan la renderització de la pluja és a molta distància de la càmera, en aquests casos, com totes les formes que representen les gotes són molt primes, la brillantor de cada gota ja no és important, ja que no s'aprecia, i per això és constant.

Figura 2.19, imatge d'una escena de la pel·lícula The Matrix Revolutions on s'hi pot veure una pluja.

En renderitzacions de pluja on aquesta estigui a prop de la càmera, cada gota projecta una imatge allargada, en la que, a diferència de plans llunyans, s'hi pot observar brillantor (observar la fila superior de la figura 2.21). Aquest patró de brillantor és molt complex a causa de les deformacions que pateix la gota d'aigua mentre està caient. Aquestes deformacions són degudes a les oscil·lacions induïdes per les diferents forces aerodinàmiques que rep la gota. Aquestes deformacions amb la llum fan que les gotes de pluja obtinguin formes amb contorns corbats brillants. Per tant, el mètode descrit anteriorment on les gotes tenien una brillantor constant, faria que, per pluges en primer pla, les gotes no tindrien una aparença real. Per solucionar aquest problema, els investigadors sovint dibuixen manualment les gotes d'aigua per representar la pluja quan aquesta es visualitzés en primers plans. Un exemple recent de la utilització de les textures de les gotes d'aigua dibuixades a ma, s'ha pogut veure a la pel·lícula "The Matrix Revolutions" [Lomas 2005]. Aquest enfocament és complex, ja que l'aparició de gotes uniformes en les diferents seqüències és bastant àmplia. A més a més, l'aparença també varia significantment depenent de la direcció de llum i del punt de vista, cosa que fa extremadament difícil fer-

ho amb la tècnica de dibuixar a mà les textures incloent variacions segons les posicions dels punts de llum i de l'observador. Per tots aquests aspectes, nosaltres hem observat dels experts en efectes especials i de la indústria d'animació i videojocs, que realitzar un renderitzat automàtic de fotos realistes de pluja representa un important problema.

Figura 2.20, escena de la pel·lícula seven on també es pot observar que és una escena amb pluja.

2.5.4 Aparença de les gotes de pluja

Primerament s'ha fet un detallat estudi de l'aparença visual de les gotes de pluja a l'article [Kshitiz and Shree K. Nayar 2006] on s'ha desenvolupat un model d'aparença de gotes de pluja que captura les complexes interaccions entre la direcció de la llum, la direcció de l'observador i les oscil·lacions de la gota. Aquestes interaccions produeixen una àmplia gamma de sorprenents efectes visuals. Alguns exemples de les gotes de pluja capturades per la càmera es poden veure a la figura 2.21, concretament a la primera de les dos files de imatges. Per la base de dades, es van agafar un total de 810 imatges reals de gotes de pluja amb diferents condicions de llum i des de diferents punts. Per comparar aquest conjunt de captures de gotes de pluja amb les diferents condicions, els autors van determinat una sèrie de valors per els paràmetres, els quals van comprovar que són els més comuns a la pluja real. Aquest valors dels paràmetres permeten aconseguir una aparença al renderitzar molt realista, alguns exemples es mostren a la fila inferior de la figura 2.21.

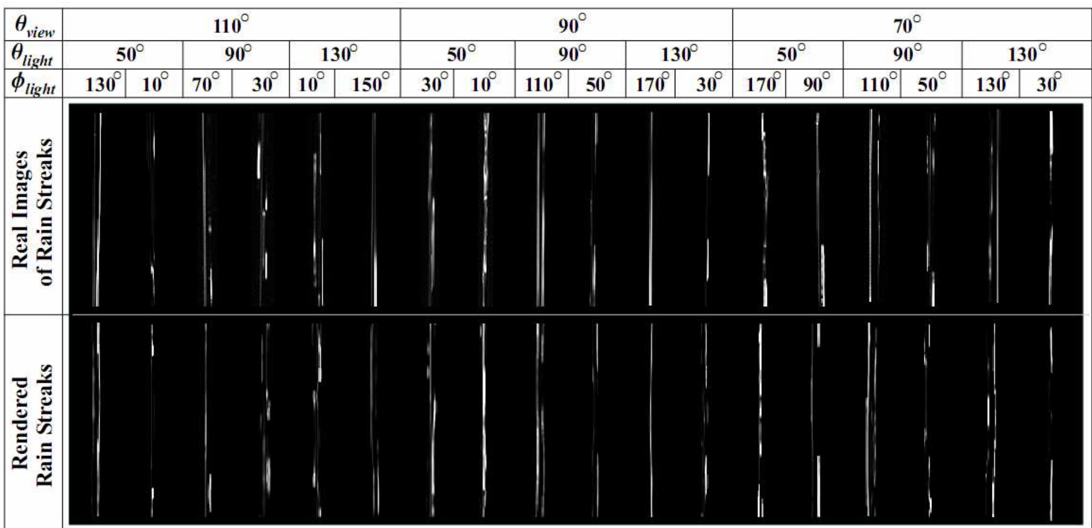


Figura 2.21, aparença de gotes d'aigua de pluja i gotes d'aigua de pluja renderitzades. La fila superior mostra imatges actuals de gotes d'aigua sota moltes diferents direccions de llum (θlight ,φlight) i direccions de l'observador θview. El patró complex d'intensitats utilitzat en cada gota de pluja és degut a la interacció de la llum amb les gotes d'aigua i les deformacions durant la caiguda de la gota. TA la BBDD tením empíricament els valors dels paràmetres d'oscil·lació que són dominants en les gotes d'aigua i utilitzarem aquests paràmetres per desenvolupar un model de pluja. La fila inferior, mostra les gotes d'aigua renderitzades utilitzant la nostra tècnica.

2.5.5 Base de dades les gotes de pluja

Donat que l'aparença de les gotes de pluja és complexa, normalment es fan servir mètodes tipus ray-tracing. Per tant, la renderització de centenars de gotes d'aigua a cada frame pot ser, computacionalment parlant, prohibitiu. La idea és renderitzar les imatges abans de començar el procés, i guardar en una base de dades les gotes de pluja. Aquesta base de dades serà utilitzada per un algoritme en temps real basat en imatges per renderitzar escenes de pluja. L'aparença de les gotes d'aigua depèn de molts factors, la direcció de la llum i de l'observador, les distàncies entre les gotes i la càmera, els paràmetres oscil·ladors, la mida de la gota i el temps d'exposició de la càmera. Una base de dades que guardés tots aquests paràmetres seria massa gran per poder-la utilitzar de forma correcte i eficient. Afortunadament, aspectes com la distància entre la càmera i la gota, les diferents mides que pugui tenir la gota d'aigua, o el temps d'exposició de la càmera, únicament produeixen petites modificacions de l'aparença de la gota, les quals podran ser realitzades en temps d'execució de forma eficient. Per tant, només necessitem una base de dades que guardi els següents paràmetres: la direcció de la llum, la direcció de l'observador i els paràmetres d'oscil·lacions. La base de dades inclou aproximadament 6300 renderitzacions de gotes d'aigua, les quals són d'accés públic, per obtenir-ne una còpia cal anar a:
http://www1.cs.columbia.edu/CAVE/databases/rain_streak_db/rain_streak.php.

Aquesta base de dades té un objectiu similar a la idea de les textures dibuixades a mà, una tècnica utilitzada en antics treballs i explicada a l'anterior apartat. No obstant, aquestes, es captura una significativa gamma més alta de d'aparences diferents de gotes.

2.5.6 Algorisme de renderització de pluja

Nosaltres hem desenvolupat un algorisme de renderització de pluja en temps real amb una base de dades d'imatges que aplica petites transformacions a les gotes de la nostra base de dades, per renderitzar les gotes segons els paràmetres de llum i de càmera ja descrits. L'usuari especifica les diferents propietats dels punts de llum. Un cop fet això, l'algoritme pot renderitzar pluja amb els paràmetres de pluja especificats per l'usuari, com la distribució de la pluja, la mida de les gotes i la direcció de les gotes d'aigua. Els resultats obtinguts amb el nostre algoritme aconsegueix un nivell de realisme molt elevat.

2.5.7 Captura de les gotes d'aigua reals

En aquesta secció analitzarem com les oscil·lacions afecten la visualització de la gota.

Considerem que una gota cau i és enfocada per la càmera i que és il·luminada per un punt de llum, podem veure aquesta situació observant la figura 2.22. L'aparença real de la gota d'aigua depèn de varis factors, els angles formats per el punt de llum respecte la gota d'aigua (θl ,φl), i l'angle format per la càmera respecte la gota. També depèn d'altres factors com la distància entre la càmera i la gota.

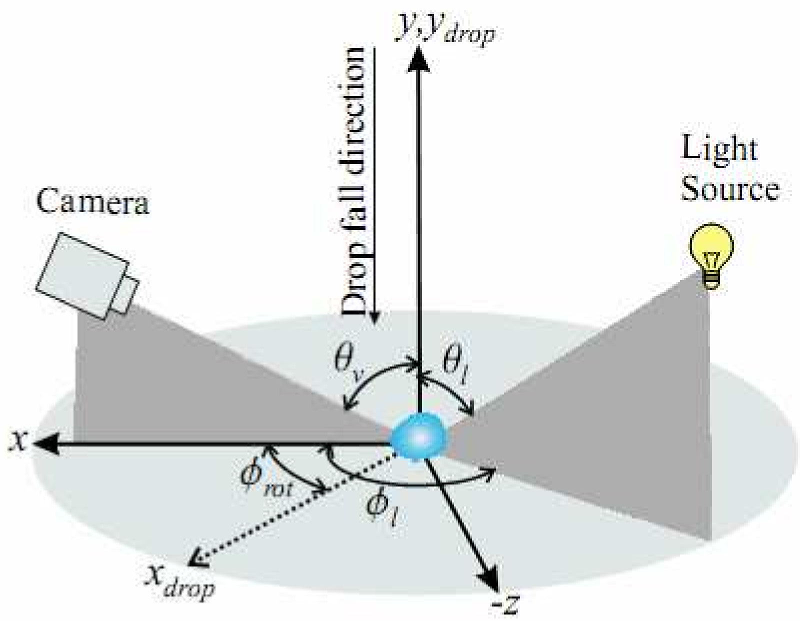


Figura 2.22, Sistema de coordenades utilitzat per mesurar la renderització de les gotes d'aigua. L'origen de coordenades està localitzat al centre de la gota d'aigua, amb l'eix y oposat a la direcció de la caiguda de la gota d'aigua. θ i φ són els angles d'elevació i d'azimut respectivament. Les variables, v i l són per diferenciar la càmera (viewing), i la llum respectivament.

A continuació explicarem com es van obtenir les imatges de la base de dades. Les gotes estaven il·luminades per un punt de llum posat a una distància d'un metre de l'origen de coordenades, veure figura 2.22. Els autors van capturar un ampli rang d'imatges de gotes d'aigua caient a una distància de 3 metros amb una Canon EOS-20D. La mida mitja de les gotes d'aigua va ser de 2 mil·límetres. Les imatges van ser capturades des de molts diferents localitzacions (θl ,φl), algunes de les quals es mostren a la figura 2.21. El punt de llum es va situar a diferents alçades formant els següents angles: 70º, 90º, 130º. L'angle azimut varia d'entre 10º i 170º en intervals de 20º. Per calcular la dependència entre la direcció de la càmera, es van capturar imatges amb els següents angles θv: 70º, 90º i 110º. 10 imatges van ser agafades per cada angle de llum-càmera per així obtenir un conjunt d'imatges en les que es pogués veure les diferents formes de les gotes d'aigua, segons els punts de llum i l'observador. En total, els autors de l'article van capturar 810 gotes d'aigua.

2.5.8 Base de dades de les gotes d'aigua renderitzades

Per tot això els autors van utilitzar imatges per a construir la base de dades que nosaltres utilitzarem a l'algoritme, ja que ens permetrà tenir un control total a l'hora de seleccionar les imatges segons els paràmetres de l'escena.

Per capturar els efectes de la llum, nosaltres tenim l'angle θl que disposa de l'interval de graus de 0ª fins a 180º i l'angle φl de 10º a 170º els dos amb passos de 20º. Per capturar els efectes de la càmera l'angle θv pot anar de 10º a 170º també amb passos de 20º entre cada angle possible.

L'aparença de la gota d'aigua també depèn de la mida de la imatge, totes les imatges tenen la mateixa amplada (16 píxels), però segons quin sigui l'angle de l'observador (θv) que caracteritzi aquella imatge tenen una alçada o una altra. Les imatges tenen la següent mida segons l'angle θv:

- θv=0º, les imatges tenen una alçada de 525 píxels.
- θv=20º, les imatges tenen una alçada de 494 píxels.
- θv=40º, les imatges tenen una alçada de 405 píxels.
- θv=60º, les imatges tenen una alçada de 272 píxels.
- θv=80º, les imatges tenen una alçada de 108 píxels.

A la figura 2.23 podem veure un exemple de les diferents mides que pot tenir les diferents textures.

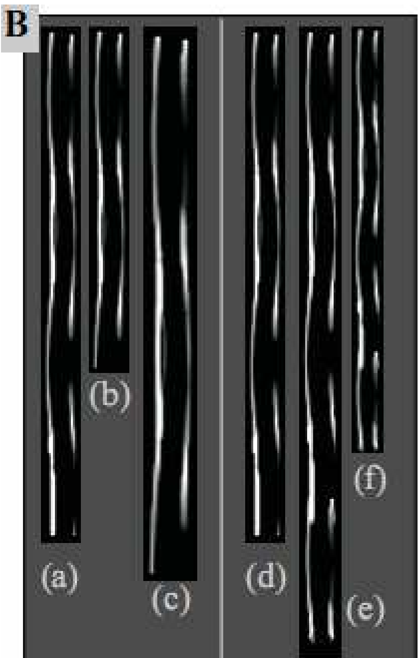


Figura 2.23, com veiem a la imatge les textures de la base de dades poden tenir cinc alçades diferents.

Podem veure els resultats de renderitzar la nostra pluja en diferents escenaris variant els punts de llum i l'observador. Veure figures 2.24 i 2.25.

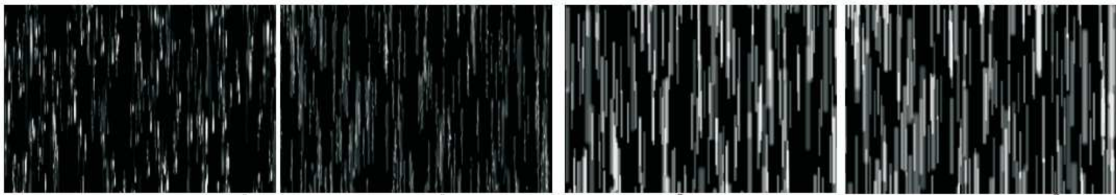


Figura 2.24, imatges que mostren el canvi de visualització després de variar la direcció de la llum.

Figura 2.25, imatges en les que podem veure un cotxe en moviment el qual amb els seus llums il·lumina i per tant ressalta vàries gotes d'aigua. Imatges agafades de [Kshitiz Garg and Shree K. 2006].

Capítol 3

-Disseny i Implementació-

En el següent capítol s'analitzarà tot el disseny del projecte, és a dir què s'ha fet, com s'ha fet i perquè s'ha fet. No es mirarà el codi exclusivament, per observar tot el codi caldrà entrar ja a dins dels fitxers de c++, veure manual tècnic.

El projecte consta bàsicament de dos parts ven diferenciades però lògicament, relacionades: l'editor de pluja i l'aplicació en 3D d'OGRE. Primerament s'explicarà tot el disseny relacionat amb l'editor, i tot seguit l'aplicació 3D, a la qual a més de veure com s'ha aconseguit l'aplicació final, s'hi podrà veure tot el procés que s'ha seguit per aconseguir resoldre petits problemes previs als objectius finals.

3.1 Editor de pluja

Per crear l'editor de pluja és necessari alguna llibreria que faci possible la creació d'entorns amb finestres d'una manera senzilla i eficient. S'ha utilitzat la llibreria wxWidgets, en concret la versió 2.4.2. En alguns casos l'editor haurà de tractar amb imatges, per fer més potent aquest tractament, s'ha buscat una altra llibreria especialitzada. En aquest cas s'ha utilitzat la llibreria d'imatges Devil. L'editor de pluja es basarà en poder carregar una imatge que anteriorment haurà estat capturada des del "cel" de l'aplicació en 3D d'OGRE. Tot seguit es modificarà aquesta imatge i es pintarà la zona la qual es vol que plogui, pintant aquesta amb un color gris determinat que indicarà la intensitat de pluja, com més fosc sigui el gris (o negre), serà indicació de més intensitat de pluja, i blanc voldrà dir que no plou. Per dibuixar aquesta zona de pluja s'han creat tres tipus de figures, un cercle, un rectangle, i un polígon irregular. Un cop es tingui la imatge amb la zona escollida per ploure, es podrà difuminar la imatge per així suavitzar el contrast dels règims de pluja. Finalment aquest editor generarà un fitxer amb totes les posicions de les partícules per l'OGRE, les quals faran referència a la posició de cada gota d'aigua dins al mon amb tres dimensions que es crearà.

3.1.1 Creació de les figures

Per aconseguir crear aquestes figures, s'ha creat una petita jerarquia, en què s'han necessitat quatre classes diferents: la classe figura, la qual serà la classe base d'aquesta estructura, la classe cercle, la classe rectangle i la classe polígon, aquestes tres últimes seran les que heretaran de la classe figura. S'ha creat la classe estructuraFigures per agrupar totes aquestes diferents figures que ja s'hagin donat l'ordre de dibuixar-les, i així disposar de tota la informació.

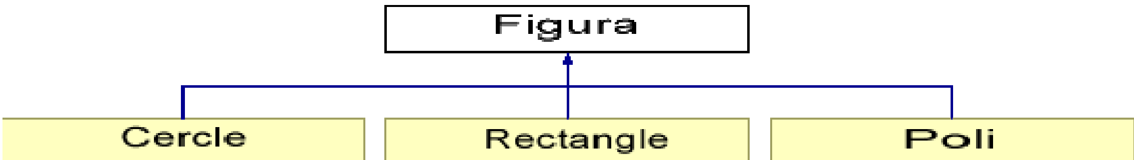


Figura 3.1, Diagrama de classes de figura

3.1.1.1 Classe figura

Variables:

La classe base d'aquesta jerarquia només disposa d'una variable privada.

wxColour color: És la variable encarregada de guardar el valor del color de la figura, wxColour és una variable de la llibreria wxWidgets. Es podria dir que aquesta variable és com un vector de tres valors, concretament aquest valors serà per aconseguir el format de color RGB (Red. Green i Blue). Aquests tres valors hauran d'estar entre el 0 i el 255. En aquesta versió de la llibreria aquesta variable wxColour encara no disposa de la possibilitat d'integrar el quart valor alpha per aconseguir una certa transparència als colors.

Mètodes:

A la part pública de la classe, hi ha tots els mètodes de la classe figura.

- Constructor: Simplement posarà al color inicial de la figura que serà blanc (és a dir color=(255,255,255).
- Destructor.
- Constructor còpia: Assigna el color de la figura passada per referència a la nova figura.
- wxColour mostrarColor(): Utilitzat per obtenir el color de la figura.
- void posarColor(int intensi): Utilitzat per assignar el color a la figura.
- virtual void dibuixar(wxDC& dc)=0: Funció declarada com a virtual pura, redefinida i implementada en les classes derivades.
- virtual void guardar (wxMemoryDC& m)=0: Funció declarada com a virtual pura, redefinida i implementada en les classes derivades.

3.1.1.1.1 Classe Cercle

Variables:

La classe cercle disposa de tres variables privades, que bàsicament definiran la posició del centre del cercle, i l'altre variable que definirà la distància del radi per així poder dibuixar el cercle.

Mètodes:

- Constructor: Heretarà el valor del color de la classe base, així com el booleà mostrat. A més a més li definirà unes posicions que s'han posat per defecte per inicialitzar el cercle.
- void posarCentre(long a,long b): Assigna les coordenades del centre de la circumferència.
- void posarRadi(long a): Assigna el valor passar per paràmetre al radi del cercle.
- void dibuixar(wxDC& dc): Per implementar el mètode **dibuixar** s'han utilitzat varis mètodes de la llibreria wxWidgets. Es rebrà per paràmetre un objecte wxDC passat per referència, ja que l'objectiu serà assignar

els valors que es tenen de l'objecte cercle (color i posicions del centre i del radi) a l'objecte que s'ha rebut per referència. Aquí es veuen els tres mètodes que s'han utilitzat de la classe wxDC:

```
dc.SetBrush(wxBrush(mostrarColor(),wxSOLID));
dc.SetPen(wxPen(mostrarColor(),1,wxSOLID));
dc.DrawCircle(centreX,centreY,radi);
```

La classe wxDC de la llibreria wxWidgets és l'encarregada de permetre visualitzar per pantalla diferents objectes d'una manera genèrica, fent possible que es pugui visualitzar tant text, com per exemple en aquest cas, figures. Per dibuixar el cercle, primerament s'ha definit el color en què es vol dibuixar la figura (setBrush i setPen) i finalment s'ha fet servir el mètode dibuixar de la classe wxDC.

- void guardar(wxMemoryDC& m): El mètode **guardar** treballa d'una manera semblant al mètode que s'acaba de veure (dibuixar), amb la diferència que se li passa per referència un objecte diferent, en aquesta ocasió l'objecte és un wxMemoryDC. Aquest objecte deriva del wxDC. WxMemoryDC és l'encarregat de guardar en memòria el que s'escriu sobre aquest objecte. Per exemple en aquest cas si l'objecte que s'ha passat per referència es diu m i fem m.DrawCircle(...), guardarà que s'ha creat una circumferència amb totes les propietats que se li han definit, proporciona els passos per poder crear un nou bitmap amb la informació que ha guardat. Així doncs el mètode guardar conté aquests tres mètodes de la classe wxMemoryDC:

```
    m.SetBrush(wxBrush(mostrarColor(),wxSOLID));
    m.SetPen(wxPen(mostrarColor(),1,wxSOLID));
    m.DrawCircle(centreX,centreY,radi);
```

3.1.1.1.2 Classe Rectangle

Variables:

Aquesta classe disposa de quatre variables que definiran la posició del rectangle, les 2 primeres variables correspondran a les coordenades de l'extrem superior esquerre del rectangle, i les altres 2 correspondran a les coordenades de l'extrem inferior dret del rectangle.

Mètodes:

- Constructor per defecte: Inicialitza a un valor per defecte les variables.
- Constructor còpia: Assigna els valors de les variables del rectangles passat per referència al nou rectangle.
- Destructor.
- void posarPuntInici(int x,int y): Assigna els paràmetres a les variables corresponents a les coordenades de l'extrem superior esquerra del rectangle.

- void posarSY(int y): Assigna el paràmetre a la variable corresponent a la coordenada y de l'extrem inferior dret del rectangle.
- void posarSX(int x): Assigna el paràmetre a la variable corresponent a la coordenada x de l'extrem inferior dret del rectangle.
- void dibuixar(wxDC& dc): Igual que el mètode de la classe cercle *(veure apartat 1.1.1.1)*, la única diferència és que en aquesta ocasió no s'utilitza el mètode DrawCircle i es fa servir el mètodeDrawRectangle.
- void guardar(wxMemoryDC& m): Igual que el mètode de la classe cercle *(veure apartat 1.1.1.1)*, la única diferència és que en aquesta ocasió no s'utilitza el mètode DrawCircle i es fa servir el mètodeDrawRectangle.

3.1.1.1.3 Classe Poli

Variables:

La classe poli (el nom ve de polígon), disposa de dos variables privades:

- Int costats: Enter a la qual se li assignaran el total de costats que té el polígon.

- wxPoint * pol: Pol és una cadena de wxPoints, la qual és una variable de wxWidgets que bàsicament guarda dos coordenades x i y. Per tant la variable pol guardarà totes les coordenades, les quals formaran tots els punts de la figura polígon.

Mètodes:

- Constructor: se li passa per paràmetre el número de costats que formaran el polígon, i així a partir d'aquest s'inicialitzeran les variables costats i pol.
- Constructor còpia: Copia els valors de l'objecte passat per referència al nou polígon.
- Destructor.
- void posarPunts(int c,wxPoint *po): Assigna a les variables els valors que se li passen com a paràmetres.
- void dibuixar(wxDC& dc): Igual que el mètode de les classes cercle i rectangle però en aquest cas es fa servir el mètode Polygon.
- void guardar(wxMemoryDC& m): Igual que el mètode de les classes cercle i rectangle però en aquest cas es fa servir el mètode Polygon.

3.1.1.2 Classe EstructuraFigures

L'objectiu d'aquesta classe és agrupar totes les diferents figures que s'hagin dibuixat, per en un futur poder-les gravar com a una imatge si així ho desitja l'usuari. Per tant s'ha creat una llista de figures que estaran lligades entre elles amb punters.

Variables:

L'estructura consta d'una estructura anomenada Node que s'ha declarat com a privada. Aquesta estructura conté dos variables:

- **Figura * element:** L'estructura disposarà d'un punter a una figura que s'ha anomenat element.

- **Node * següent:** Aquesta variable servirà per enllaçar l'actual node amb el pròxim node que es pugui afegir a l'estructura.

Apart d'aquesta estructura la classe conté dos variables més declarades com a privades, les variables són dos punters a nodes més, anomenades **inici** i **actual.**

Mètodes:

Aquesta és la llista de funcions (constructors i destructors inclosos) de la classe:

- **estructuraFigures**(): Inicia les variables inicial i actual que apuntin a un NULL.
- ~**estructuraFigures**().
- **void afegir**(figura * t): El nou element serà afegit al principi de la llista.
- **void avançar**(): El punter actual passarà a apuntar a la pròxima element de la llista.
- **bool final**(): El mètode final retornarà cert si actual apunta a NULL.
- **figura * obtenirFigura**(): Retornarà l'element (és a dir una figura) que sigui apuntat per el node actual.
- **void situarAlPrincipi**(): Fa apuntar el node actual al principi de l'estructura, és a dir allà on apunta inici.
- **void mostrarNoMostrats**(wxDC& dc): Té com objectiu mostrar les figures que no s'han mostrat encara per pantalla. Aquest mètode ha estat necessar-hi crear-lo ja que cada vegada que l'usuari dóna la ordre de dibuixar una figura, simplement el que es fa és afegir-la a l'estructura, per tant quan s'actualitza la pantalla de l'editor és quan es crida aquesta funció. El que fa és recórrer totes les figures de l'estructura fins al final i va donant la ordre de dibuixar-les (amb el mètode dibuixar implementat en les subclasses de la classe figura).
- **void guardar**(wxMemoryDC& m): Farà exactament el mateix que mostrarNoMostrats, però amb la única diferència que al obtenir la figura, no es cridarà el mètode dibuixar sinó que cridarà el de guardar. Afegir(figura *a) tindrà la funció d'anar creant nous nodes per col·locar-hi la figura que li passaran com a paràmetre. Aquest nou element, com que no importa l'ordre en què es mostra per pantalla o es guarda cada figura, s'afegirà al principi de la llista. Per tant, apuntarà (el seu node->seguent) allà on apuntava el node inici i després s'igualarà el node inici a aquest nou node.
- **void buidar**(): S'utilitzarà quan, per exemple, s'obri un nou document, per tant les figures que s'hagin creat anteriorment ja no importaran,

buidar el que fa és igualar les variables de l'estructura (actual i inici) a NULL.

- **bool buida**(): Retornarà cert si inici és igual a NULL
- **void guardarFigures**(wxString a,wxBitmap *g): Se li passa com a paràmetres un wxString i un wxBitmap, serà l'encarregat de guardar les figures que ha creat l'usuari des de l'editor en un format d'imatge. La variable wxString serà el nom que l'usuari haurà escollit com a nom d'imatge, per exemple podria ser figures.jpg. I la variable wxBitmap (classe encarregada de tractar les imatges) serà la que en aquell moment hi havia en la pantalla de l'editor i únicament serà utilitzada per agafar l'alçada i l'amplada d'aquesta imatge. Per tant, gràcies a aquesta imatge, en crearem una altra (de forma temporal). Aquí veiem com creem aquest nou bitmap:

```
wxBitmap  *esbos = new wxBitmap(g_TestBitmap->GetWidth(),g_TestBitmap->GetHeight(),-1);
```

G_TestBitmap és el wxBitmap que s'ha rebut per paràmetre. Aquesta classe permet, per exemple, saber l'amplada i l'alçada d'aquesta imatge amb el mètode GetWidth i GetHeight.

El mètode s'encarrega de controlar si s'ha entrat un nom correcte o no, per fer-ho primerament controla si l'string rebut està buit o no, si és així es sortirà d'aquest automàticament:

```
if (f == _T(""))  return;
```

Tot seguit controla quin tipus format d'imatge té el nom que s'ha seleccionat la imatge. Això s'ha aconseguit gràcies a una combinació de mètodes de que disposa wxString: Primer obtindrem el wxString que hi ha després de l'últim punt del nom seleccionat (AfterLast) i amb el que s'obtingui se li aplicarà el mètode també de wxString Lower, per assegurar que l'extensió que s'acaba d'aconseguir sigui amb minúscules i no creï cap problema a l'hora de fer les pròximes comparacions. Amb el wxString que s'ha obtingut, fent una simple comparació, es podrà saber si es vol guardar les figures dibuixades amb format png, o bé amb format jpg, o bmp, etc.

```
if (extensio==wxT("jpg")){
```

Abans de poder guardar totes les figures que s'han dibuixat en el fitxer escollit, s'ha hagut de crear un objecte wxMemoryDC per preparar el bitmap al qual gravarem les figures dibuixades. És a dir, seleccionarem la imatge (SelectObject) que s'està mostrant per pantalla, es guardaran les seves propietats a l'objecte wxMemoryDC, i després per guardar una imatge en blanc s'esborraran els píxels dels quals s'acaben de quedar

gravats al seleccionar l'objecte (Clerar). Finalment se li assignarà el fondo blanc. Veiem com s'ha fet:

```
mem.SelectObject(*esbos);
mem.Clear();
mem.SetBackground(wxBrush(*wxWHITE, wxSOLID ));
```

Per tant amb el mètode que ja s'ha explicat fa un moment (guardar), junt amb el mètode saveFile de la classe wxBitmap s'aconseguirà guardar tot el que s'hi hagi escrit en l'objecte que s'ha creat (i modificat al mètode guardar), aquí es veu un exemple de les comparacions que fa per determinar el format desitjat de la imatge, aquí es veu el cas del format jpg:

```
...
guardar(mem);
esbos->SaveFile( f, wxBITMAP_TYPE_JPEG, (wxPalette*)NULL );
}
```

3.1.2 Construcció de l'entorn de finestres

Per aconseguir implementar l'editor amb un sistema de finestres, com ja s'ha comentat anteriorment, s'ha utilitzat la llibreria wxWidgets versió 2.4.2. A continuació es podrà veure un resum de com i què s'ha utilitzat d'aquesta llibreria.

3.1.2.1 Classes creades

En total s'han creat un total de 5 classes noves, totes elles que deriven d'altres classes de la llibreria wxWidgets, aquestes són:

- MyApp, la qual deriva de wxApp
- MyFrame, la qual deriva de wxFrame
- MyCanvas, la qual deriva de wxScrolledWindow
- MyMiniFrame, la qual deriva de wxMiniFrame
- MyMiniFrameDos, que també deriva de wxMiniFrame

3.1.2.1.1 Classe MyApp

Únicament s'ha definit el constructor per defecte i s'ha redefinit la funció ja definida a la classe base OnInit().

OnInit: És el mètode més important de tots: no es pot fer una subclasse de la classe wxApp i no proporcionar aquest mètode a la classe. Normalment és l'encarregada de crear la pantalla principal de l'aplicació que es vol realitzar. En aquest cas el que fa primer onInit és assegurar que els diferents formats d'imatges podran funcionar en aquesta aplicació. Per garantitzar que no hi

hauran problemes, es crida una funció estàtica de wxImage anomenada addHandler, per garantitzar el funcionament dels següents formats: PNG, JPEG, TIFF, GIF, BMP, ICO, PCX, PNM, XPM, CUR, i ANI. Aquí es veu un dels casos explicats:

```
wxImage::AddHandler( new wxJPEGHandler );
```

Tot seguit s'inicialitzarà la classe devil, ja que en algunes ocasions al llarg de la implementació d'aquest editor serà de gran utilitat per manipular, carregar i/o guardar les imatges.

Al llarg d'aquesta aplicació en varis mètodes s'ha necessitat la llibreria Devil, aquesta llibreria es centre exclusivament amb carregar, guardar, manipular aplicar filtres, a les imatges, en definitiva el seu objectiu és el tractament de les imatges. De Devil utilitzarem dos de les seves classes les quals tenen dos funcions ven diferenciades:

- La primera és **il.h**: L'objectiu d'aquesta serà carregar, guardar, seleccionar i extreure informació de les imatges.
- La segona és **ilu**: L'objectiu d'aquesta és modificar les imatges carregades.

Per tant, per poder utilitzar les seves funcions, inicialment s'han d'haver carregat, per això també en el mètode OnInit s'inicialitzen ja que es necessitaran en algunes funcions de l'editor.

Un cop fet això, s'ha creat el frame principal de l'aplicació (*veure el pròxim apartat*), serà un objecte de la classe creada MyFrame. Per crear aquest objecte cal passar-li quatre paràmetres que defineixen la posició d'aquest, la grandària i el nom que tindrà. Aquí es veu com s'ha creat el frame principal de l'aplicació:

```
wxFrame *frame = new MyFrame((wxFrame *) NULL, _T("Localitzacio de la
pluja"), wxPoint(0, 0), wxSize(300, 300));
```

3.1.2.1.2 Classe MyFrame

Com ja s'ha dit la classe MyFrame deriva de wxFrame, i només disposarà d'una variable que serà un punter de la classe MyCanvas, que s'anomenarà canvas.

Mètodes:

- **MyFrame**(wxFrame *parent, const wxString& title, const wxPoint& pos, const wxSize& size): El constructor de la classe myFrame serà l'encarregat de crear tots els menús, afegir barres, definir certs missatges i inicialitzar l'objecte canvas (que seria l'equivalent a la pantalla on es mostraran les imatges carregades), i a on es podrà

dibuixar les figures. Per tenir comunicació amb aquesta classe, com a paràmetre de l'objecte MyCanvas si passa aquest objecte, MyFrame.

Per crear els diferents menús que tindrà el frame s'han necessitat dos objectes diferents, el wxMenu i el wxMenuBar. Amb el wxMenu s'aconsegueix crear els menús desplegables de l'aplicació, en aquest cas s'han creat els menús: file_menu (el qual el seu nom visible per els usuaris serà fitxer), help_menu(el nom visible serà ajuda), inserir_menu (inserir), efectes i posicions (*veure figura 3.6)*.

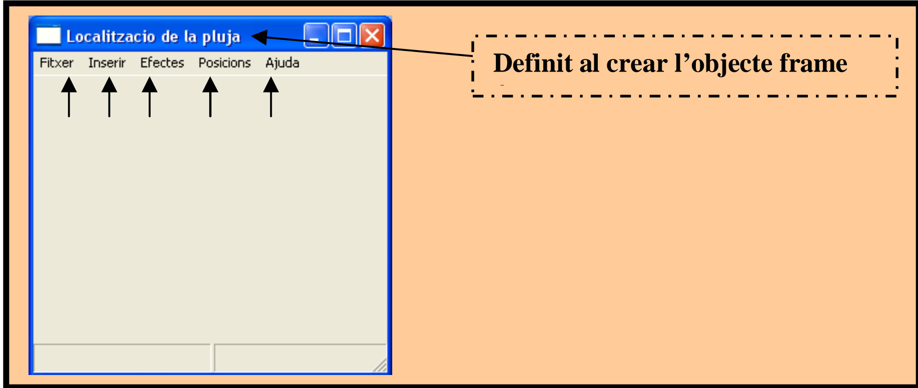


Figura 3.6

- virtual ~MyFrame().
- void OnActivate(bool).
- void OnLoadFile(wxCommandEvent& event).
- void OnSaveFile(wxCommandEvent& event).
- void OnGuardarFigures (wxCommandEvent & event).
- void OnPosarFigures (wxCommandEvent & event).
- void OnQuit(wxCommandEvent& event).
- void OnAbout(wxCommandEvent& event).
- void OnInserirCercle(wxCommandEvent& event).
- void OnInserirRectangle(wxCommandEvent& event).
- void OnInserirPoligon(wxCommandEvent& event).
- void OnDifuminar (wxCommandEvent& event).
- void posarPosicions (wxCommandEvent& event).
- void OnInserirInfoPosicions (wxCommandEvent& event).
- void OnCreateAtlas (wxCommandEvent& event).
- void OnCreateAtlasAmbient (wxCommandEvent& event).

Com es veu tots els mètodes de la classe tenen com a paràmetre un objecte de la classe event, i és que al ser una llibreria pensada per finestres, pràcticament tot es basa amb events. Per això cal decidir quins events pertanyen a cada funció i com reaccionaran al activar-se. Per declarar la llista d'events també s'ha definit com a públic a la classe definició, d'aquesta manera:

```
DECLARE_EVENT_TABLE()
```

I aquesta és la implementació:

```
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(QUIT,                          MyFrame::OnQuit)
    EVT_MENU(ABOUT,                         MyFrame::OnAbout)
    EVT_MENU(LOAD_FILE,                     MyFrame::OnLoadFile)
    EVT_MENU(SAVE_FILE,                     MyFrame::OnSaveFile)
    EVT_MENU(GUARDAR_FIGURES,          MyFrame::OnGuardarFigures)
    EVT_MENU(INSERIR_CERCLE,           MyFrame::OnInserirCercle)
    EVT_MENU(INSERIR_RECTANGLE,        MyFrame::OnInserirRectangle)
    EVT_MENU(INSERIR_POLIGON,          MyFrame::OnInserirPoligon)
    EVT_MENU(DIFUMINAR,                MyFrame::OnDifuminar)
    EVT_MENU(POSAR_POSICIONS,          MyFrame::posarPosicions)
    EVT_MENU(POSAR_HELP,         MyFrame::OnInserirInfoPosicions)
    EVT_MENU(CREATE_ATLAS,       MyFrame::OnCreateAtlas)
    EVT_MENU(CREATE_ATLAS_AMBIENT, MyFrame::OnCreateAtlasAmbient)

END_EVENT_TABLE()
```

Com es pot veure, defineix quin tipus d'event serà, que en aquest cas seran tots evt_menu, explica que quan es produeixi l'event per exemple quit, la funció que s'haurà de cridar de la classe MyFrame és OnQuit. Tots els events quit, about, etc, estan definits prèviament amb un conjunt de defines.

Mètodes generals de wxWidgets

A continuació s'explicarà els mètodes més destacats que s'utilitzen en les funcions de la classe MyFrame que són propis de la llibreria wxWidgets:

wxFileSelector: Funció que permetrà seleccionar un fitxer segons el format que se li descriurà al definir la funció , aquí es veu un exemple de la crida(utilitzat en el mètode OnSaveFile, OnGuardarFigures i OnDifuminar):

```
wxString f = wxFileSelector(wxT("Guarda la imatge"), (const wxChar *)
NULL,(const wxChar *) NULL,(const wxChar *) NULL,wxT("BMP files
(*.bmp)|*.bmp|") wxT("PNG files (*.png)|*.png|") wxT("JPEG files
(*.jpg)|*.jpg|") wxT("TIFF files (*.tif)|*.tif|") wxT("PCX files
(*.pcx)|*.pcx|"),wxSAVE);
```

wxMessageBox: Utilitzat per a informar al usuari d'errors que ha realitzat, per fer sortir missatges d'informació, etc.

3.1.2.1.3 MyCanvas

Aquesta classe, tindrà la funció de proporcionar l'espai per col·locar les imatges en la seva pantalla, de reaccionar als event a la pantalla, etc.

MyCanvas és una subclasse de la classe wxScrolledWindow, que disposa de dos variables privades: MyFrame *frame i MyMiniFrame *mini_frame.

Aquests són els constructors i funcions de que disposarà la classe:

- **MyCanvas(MyFrame *parent, const wxPoint& pos, const wxSize& size):** El constructor d'aquesta classe copia a la variable frame. El paràmetre parent, per així poder saber en tot moment com està la classe MyFrame (recordem que l'objecte MyCanvas és cridat en el constructor de MyFrame).
- ~MyCanvas(void) .
- **void OnPaint(wxPaintEvent& event):** Com es veu **OnPaint** s'activa quan rep un event de wxPaintEvent. Aquest event s'envia quan a la finestra s'hi ha dibuixat o pintat alguna cosa. Per tant s'envia un event de pintat, és a dir, la pantalla s'haurà d'actualitzar. El que pot haver passat és que l'usuari hagi ordenat dibuixar alguna figura, per tant al final d'aquest mètode, després d'haver preparat el bitmap, es cridarà el mètode de la classe estructuraFigures **mostrarNoMostrats** per dibuixar les figures que s'han ordenat. La resta continua com ja s'ha explicat.
- **void clicarPantalla(wxMouseEvent &event)**: És el mètode que únicament tindrà sentit quan s'hagi entrat en l'apartat d'insertar cercle, o rectangle, o polígon, ja que, la classe MyCanvas disposa d'un objecte mini_frame, el qual disposa d'algunes variables per determinar si s'està intentant dibuixar un cercle, o un rectangle o un polígon.

La funció que ha de desenvolupar el mètode clicarPantalla és el d'esperar que l'usuari intenti dibuixar alguna figura a la pantalla. Quan ho faci, aquest mètode entrarà en funcionament. Calcularà la posició de la pantalla que s'ha clicat i emmagatzemarà la dada per utilitzar-lo a l'algorisme. Per que es pugui veure més clara la idea d'algorisme que s'ha utilitzat, posarem l'exemple de que es vol dibuixar un **polígon**:

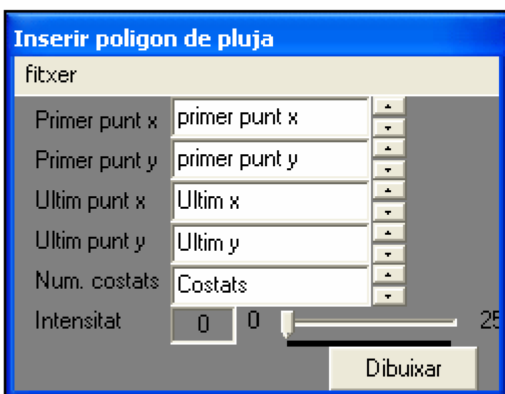


Figura 3.7, miniFrame per insertar un polígon

Quan activem l'opció d'inserir polígon, podrem veure la finestra de la figura 3.7, estarem en disposició de començar a seleccionar les posicions. Ara, si cliquem en la pantalla general, veuríem que el mini_frame s'ha actualitzat posant les coordenades del primer punt on s'ha clicat. El mètode sap la posició on s'ha clicat gràcies al mètode de l'event que s'ha rebut:

```
wxPoint pt( event.GetPosition() );
```

L'algorisme actualitza les dades de l'objecte mini_frame, concretament les variables coodx i coody que són variables spinCtrl (coodx i coody que correspondran a les coordenades del primer punt del polígon). Veiem com comprova que sigui el primer punt i la manera en què actualitza el valor de l'spinCtrl:

```
if (mini_frame->costats==0){
        mini_frame->coodx->SetValue(a);mini_frame->coody->SetValue(b);
}
```

També actualitza el punter de posicions de que disposa l'objecte mini_frame, el qual emmagatzemarà tots els costats del polígon, i augmentarà el número de costats a 1. Aquí veiem el que s'ha explicat:

```
mini_frame->poligon[mini_frame->costats]=wxPoint(a,b);
mini_frame->costats++;mini_frame->costat->SetValue(mini_frame->costats);
```

El pròxim cop que es faci clic a la pantalla (en senyal de seleccionar el segon punt del polígon) i es torni a activar aquest algorisme s'actualitzaran les variables que indiquen últim punt x i últim punt y (a part del codi ja explicat d'actualitzar els costats del polígon i el número d'aquests). Aquest procés s'anirà repetint fins que l'usuari decideixi prémer el botó dibuixar o tancar el mini_frame de polígons.

- **void movimentRatoli(wxMouseEvent &event):** ens a servirà d'ajuda quan s'estigui movent el ratolí per la pantalla de l'editor, ja que en tot moment veurem a la part inferior de l'aplicació la posició exacte del ratolí(veure figura 3.8). Això ens permetrà poder dibuixar amb exactitud la zona que volem que plogui.

Figura 3.8, podem observar les coordenades

Com en el cas anterior el mètode es basa en l'event que es generarà al moure el ratolí per sobre de la pantalla de l'aplicació. En aquest cas l'event que es generarà és el de wxMouseEvent, MovimentRatolí, un cop cridat utilitzarà el mètode de l'event GetPosition. Si només utilitzéssim aquesta funció per aconseguir la posició del ratolí, en moltes ocasions no ens seria útil el valor obtingut, ja que, per exemple, si una imatge és més gran que la que s'ha

dissenyat en l'objecte MyCanvas es crearà un scroll per recórrer tota la imatge, per tant patiríem del problema de que no es podrien calcular totes les posicions de la imatge, ja que només calcularia com a màxim les de la pantalla sense contar l'scroll que s'ha pujat o baixat. Per aconseguir calcular la posició exacta tenint en compte l'scroll disposem d'una funció pròpia de la classe base wxScrolledWindow, la funció CalcUnscrolledPosition. Veiem aquestes dos funcions aplicades al nostre codi:

```
int a,b;
wxPoint pos = event.GetPosition();
CalcUnscrolledPosition(pos.x,pos.y,&a,&b);
```

Veiem que la funció CalcUnscrolledPosition modificarà els valors de a i b, ja que han estat passat per referència, després només ens caldrà passar la frase amb aquest resultat en la barra d'estat de l'objecte frame, aquí veiem com ho hem fet:

```
wxString str;
str.Printf( wxT("Posicio actual del ratolí: %d,%d"), a,b);
frame->SetStatusText( str );
```

- **void setMiniFrame(MyMiniFrame * a):** únicament tindrà l'objectiu d'assignar un objecte myMiniFrame al de l'objecte de que disposa myCanvas.

3.1.2.1.4 MyMiniFrame

Aquesta classe ens permetrà generar frames de mida més petita que el general explicat anteriorment. Concretament s'utilitzarà a l'hora de insertar figures a la pantalla, a més aquesta classe tindrà la missió de guardar els valors de les posicions (entre altres coses) a les que s'han de dibuixar les figures (com s'ha explicat en l'apartat anterior), i de tenir una comunicació directa amb l'objecte canvas que té com a paràmetre. Apart de l'objecte MyCanvas la classe tindrà les variables públiques de tipus booleà miniCercle, miniRectangle i miniPoligon, les quals seran certes si s'està procedint a dibuixar una d'aquestes tres figures, altrament romandran en fals. Aquesta classe també disposarà de dos variables de tipus enter per guardar el número de clics que s'han fet a la pantalla, i també el número de costats que s'han generat en el cas de que s'estigui fent un polígon. Disposa de cinc variables de tipus wxSpinCtrl (veure figura 3.9),que són les coordenades de les figures. Un altre objecte d'aquest estil del qual disposarà aquesta classe és wxSlider (veure figura 3.10), que s'ha utilitzat per guardar el valor de la intensitat de gris (o pluja) que se li aplica a la figura.

Figura 3.9 Figura 3.10

Finalment la classe disposarà d'un punter a wxPoint, el qual serà utilitzat per guardar tots els costats del polígon.

El constructor s'encarregarà de construir l'entorn del frame, és a dir, col·locarà els menús de que disposaran els seus objectes, instanciarà alguns dels objectes presentats fa un moment, etc.

Mètodes:

La classe disposa d'aquests mètodes:

- **void OnTancar(wxCommandEvent& WXUNUSED(event))**: Com es pot preveure tanca el frame que s'ha obert al clicar a una de les opcions d'insertar figura.
- **void dibuixar(wxCommandEvent & WXUNUSED(event)):** Al prémer la opció del frame insertar cercle (per exemple), s'obrirà el mètode onInserirCercle. Dins d'aquest es crearà un objecte MyMiniFrame al qual apareixerà per pantalla, com s'ha explicat en anteriors apartats l'usuari podrà escollir on es vol el cercle seleccionant les parts de la pantalla o també inserint manualment les coordenades que es vol que es dibuixi el cercle. Al acabar de seleccionar la posició que es vol dibuixar el cercle i de seleccionar la intensitat en què es vol que es pinti aquest, només caldrà prémer el cercle per veure dibuixat aquest per pantalla. Automàticament quan es s'apreta el botó de nom dibuixar es crida el mètode d'aquesta pròpia classe **dibuixar.**

S'ha creat un mètode que sigui general per les tres possibles figures, que únicament el que fa és primer de tot mirar quina figura s'està intentant dibuixar, i tot seguit agafar les dades que s'han guardat en els paràmetres de l'objecte i crear la figura en qüestió per després afegir aquesta figura a la llista de figures dibuixades.

Perquè la pantalla (MyCanvas) s'adoni de que té d'actualitzar-se es crida el mètode de que disposa per fer-ho, un cop fet això apareixerà la nova figura en pantalla.

Començarem per explicar el funcionament del mètode dibuixar, per explicar-ho més fàcilment ho farem suposant que estem intentant dibuixar un cercle. Al prémer el botó dibuixar com s'ha dit es cridarà el mètode de MyMiniFrame dibuixar, aquest comprovarà quina figura s'està intentat dibuixar. Això ho podrà mirar gràcies al paràmetre booleà que té cada objecte MyMiniFrame, si és cert entrarà a mirar les coordenades i intensitat del cercle, veiem el un tros de codi utilitzat en el mètode:

```
...
if (miniCercle){
       cercle *cerc=new cercle;
       cerc->posarCentre((long)coodx->GetValue(),(long)coody-
>GetValue());
       cerc->posarRadi((long)rad->GetValue());
       cerc->posarColor(intensitat->GetValue());
       totalFigures.afegir(cerc);
…
```

Com es pot veure, aquest codi primerament comprova si s'està dibuixant el cercle, si és així crea una figura del tipus cercle. Tot seguit agafa els valors dels paràmetres de l'objecte MyMiniFrame per saber les coordenades en què estarà el centre i el radi. Finalment afegeix a l'objecte la intensitat que s'ha seleccionat en el mini frame. Quan s'han posat totes les dades a la figura, s'afegeix a l'estructura de figures totalFigures.

Un cop tenim l'estructura omplerta, només cal que la pantalla sàpiga que s'ha d'actualitzar. Com s'ha explicat en el tema de la classe MyCanvas tenim el mètode **OnPaint** que està a l'espera de rebre un event per actualitzar la pantalla, per això com en l'objecte MyMiniFrame tenim l'objecte MyCanvas podem cridar el seu mètode refresh que farà que cridi el mètode citat. Aquí veiem com fem servir l'objecte de nom canvas per refrescar i la resta de codi del mètode:

```
…
    totalFigures.afegir(cerc);
    canvas->Refresh();
    miniCercle=false;
    Close(false);
      }
```

3.1.2.1.5 MyMiniFrameDos

Aquesta, com l'anterior, disposa d'un objecte MyCanvas, i a més a més un objecte MyFrame, un wxSpinCtrl, dos wxSlider i un punter de caràcters.

Mètodes:

Aquests són els mètodes de que disposa la classe:

- void OnTancar(wxCommandEvent& WXUNUSED(event));
- void ajudaDifuminar (wxCommandEvent& WXUNUSED(event));
- void calcular(wxCommandEvent & WXUNUSED(event));
- void transformar(wxCommandEvent & WXUNUSED(event));
- void setNom(wxString);

MyMiniFrameDos serà cridat des de la classe de frame **onDifuminar**, que començarà amb un mètode de wxWidgets ja explicat, el wxFileSelector, en el

que l'usuari podrà escollir quina imatge desitja. Un cop escollida automàticament a part de poder veure la imatge escollida es crearà un objecte MyMiniFrameDos i li assignarem el nom que s'ha obtingut de cridar el wxFileSelector. Això ho farem amb el mètode **setNom**, i també li assignarem l'objecte MyCanvas de que disposa la classe MyFrame. Aquí podem veure la creació de l'objecte i les assignacions explicades:

```
…
MyMiniFrameDos   * mini_frameDos = new MyMiniFrameDos( this, -1,
_T("Transformar zona de pluja"),
wxPoint(100, 100), wxSize(250, 155));
mini_frameDos->setNom(f);
mini_frameDos->canvas=canvas;
…
```

Observem com en aquest mateix mètode de frame es construeix el mini frame, en aquest cas veiem com es crea un botó i com s'instancia els objectes pixels i blur els quals estan definits a la classe:

```
...
mini_frameDos->pixels = new wxSlider( mini_frameDos, -1, 0, 0, 30,
wxPoint(10,5), wxSize(155,-1),wxSL_AUTOTICKS | wxSL_LABELS );
mini_frameDos->blur = new wxSlider( mini_frameDos, -1, 0, 0, 30,
wxPoint(10,55), wxSize(155,-1),wxSL_AUTOTICKS | wxSL_LABELS );
boto=new wxButton(mini_frameDos, event_transformar,"transormar");
SizerPrimer= new wxBoxSizer(wxVERTICAL);
SizerPrimer->Add(mini_frameDos->pixels,0,wxLEFT|wxALIGN_LEFT,78);
SizerPrimer->Add(mini_frameDos->blur,0,wxLEFT|wxALIGN_LEFT,78);
...
```

Podem veure el resultat a la figura 3.11:

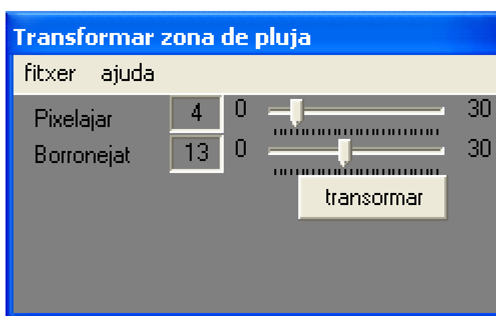


Figura 3.11

Al prémer el botó dibuixar, es cridarà el mètode **transformar**, que serà explicat en el pròxim mètode ja que s'ha considerat que és un mètode diferent als que hem explicat fins ara.

Al clicar el menú Posicions activarem el mètode **posarPosicions**, que crearà un objecte MyMiniFrameDos i com, en el cas anterior, crearà els objectes que necessitarà per tenir l'entorn de finestra desitjat. En aquest cas només afegirà a l'objecte wxSlider que servirà per escollir el número de gotes que desitjarem

col·locar a l'aplicació OGRE. Quan s'apreti el botó anirem al mètode de MyMiniFrameDos **calcular**, el qual podríem considerar que és el més important de l'editor, ja que és l'encarregat de calcular les posicions en què cauran les gotes segons les figures desitjades, com en el mètode anterior el veurem més explicat a fons en el pròxim apartat.

3.1.3 Processament del mapa de pluja

Podríem dir que l'aplicació d'autor té dos funcions que es diferencien clarament de la resta. Fins ara els mètodes que s'han vist utilitzaven una combinació de la llibreria wxWidgets i l'estructura de figures creada. Els objectius d'aquests mètodes són:

3.1.3.1 Difuminar les zones de pluja

Quan ens referim a **difuminar les zones de pluja**, l'objectiu és que el contrast entre les diferents intensitats generades per les figures no sigui tant gran. Si ho enfoquem dintre l'aplicació en 3d de pluja que es vol crear, el que volem aconseguir és un major realisme, evitant veure que en una posició de l'escena plogui i un centímetre al costat no ho faci. Per aconseguir-ho, s'ha creat aquest mètode, que es cridarà primerament accedint al menú del frame efectes->difuminar. Tot seguit, quan ja s'hagi triat la imatge. On ja hi ha la zona de pluja escollida, s'haurà d'escollir quin valor de difuminació es vol. Aquest valor es tria combinant dos valors, el de pixelització i el de blur. Quan s'hagin escollit aquests valors només caldrà prémer el botó transformar i esperar el resultat.

- **Pixelització,** baixarà la resolució de la imatge.
- **Blur,** la traducció de blur és la de barrejar. El seu objectiu serà agafar els diferents colors que estiguin en contacte i barrejar-los una mica, fent l'efecte de suavitzar i de dispersar el que d'altre manera seria unes figures amb regions massa homogènies cosa que faria que fos poc creïble en una pluja.

A les figures veiem dos exemples de zones editades amb les mateixes figures *(3.12 i 3.13).*

Figura 3.12, observem que s'han dibuixat tres figures

Podem veure d'esquerra a dreta, un rectangle amb una intensitat de pluja molt forta, tot seguit veiem un petit espai blanc que significaria que no hi plou, i tot seguit un cercle amb una intensitat suau i un rectangle altre vegada amb una

intensitat molt elevada. Veiem que just a sobre i a sota d'aquestes figures és completament blanc, comprovem a veure que passa a la figura número 3.13:

Figura 3.13, veiem les figures difuminades

Aquí podem veure les mateixes figures que en la imatge número 3.12, però a diferència de la primera aquesta ha estat transformada i editada. Concretament se li ha aplicat pixelització blur. Com es pot veure, el primer rectangle i el cercle pràcticament s'han ajuntat, i si més no, fa una escala de grisos molt suau. Sobretot en els rectangles que són de més intensitat que el cercle, podem comprovar que s'ha suavitzat molt el pas de negra a blanc.

Aquest efecte s'ha implementat en la classe MyMiniFrameDos, concretament al mètode transformar (s'activarà aquest mètode un cop ja s'hagi escollit la imatge i s'hagin seleccionat els valors per modificar aquesta, tot això en el mètode OnDifuminar).

En aquest mètode s'utilitzarà la llibreria Devil, que, com ja s'ha dit, és una llibreria especialitzada en tractar imatges. Començarem per declarar la variable que ens permetrà estar amb comunicació amb la imatge. Tot seguit li assignarem un identificador i finalment la seleccionarem :

```
ILuint ImageName;
ilGenImages(1, &ImageName);
ilBindImage(ImageName);
```

Un cop fet això, podem carregar la imatge amb la que volem treballar. Farem servir el nom que s'ha entrat en el mètode OnDifuminar (la variable es diu fit) i s'ha passat com a paràmetre de l'objecte MyMiniCanvasDos. També podem observar que si la imatge no està en format RGB la converteix a aquest format.

```
if (!ilLoadImage(fit)){ //carreguem la imatge
      wxMessageBox(_T("error"),
      _T("error al carregar la imatge"),
      wxOK|wxCENTRE, this);
}
ilConvertImage(IL_RGB,IL_UNSIGNED_BYTE);
```

Un cop tenim la imatge preparada, fem servir els mètodes de la classe ilu de la llibreria Devil, **iluPixelize** i **iluBlurAvg**, agafant els valors que s'han guardat a l'objecte MyMiniFrameDos.

```
iluPixelize(pixels->GetValue());//pixelagem la imatge segons el valor
escollit
iluBlurAvg(blur->GetValue());//utilitzem el filtre
```

Ara ja tenim modificada la imatge, i la guardem amb un nom per defecte i es mostrarà per pantalla. Per tant, cal tornar a generar un event perquè s'actualitzi la pantalla. Com a l'objecte mini_frameDos disposa d'un objecte canvas, podrem cridar el seu propi mètode refresh:

```
canvas->Refresh();
```

3.1.3.2 Calcular les posicions de pluja

Descripció general:

Per **calcular les posicions de les gotes** que han de caure a l'escenari de l'aplicació d'OGRE, primerament hem de saber les coordenades de que disposem del mon de l'aplicació amb 3d. Per tant, com es veurà en el proper tema, al moment de capturar l'escenari del videojoc, també es generarà un fitxer .bin amb totes les coordenades del mon real que es podien veure al moment de realitzar la captura, ja que si no ho féssim no tindríem manera de saber l'equivalència entre les coordenades de pantalla (de 2 dimensions) i les del mon d'Ogre3D.

L'objectiu, com ja s'ha dit, és repartir les gotes d'aigua de forma que sigui aleatòria, però alhora complint la distribució què l'usuari ha decidit. Per aconseguir això s'ha creat una classe per poder aplicar l'algorisme: **Cumulative Density Function** (CDF), adaptant-la a les necessitats del cas. La classe s'ha anomenat **distribuïdorGotes** i s'ha utilitzat, apart de la llibreria devil, les stl. Dins d'aquesta classe se n'ha creat una altra, anomenada **parcel·les**.

La classe **parcel·les** disposa d'aquests paràmetres:

- Float x, y: Guardaran una coordenada d'OGRE del píxel de la imatge corresponent, i per tant serà una de les possibles posicions en les que podran caure les gotes.
- Double valorGris: Guardaran el valor del resultat entre el gris (intensitat), al qual l'usuari li ha assignat a aquella posició, dividit pel total d'intensitats.
- Double valorGrisAcumulat: L'objecte parcel·les no té sentit fora del vector, per tant, la missió d'aquest paràmetre consisteix en guardar la suma de tots els paràmetres "valorGris" dels objectes que el precedeixen en el vector.

La classe parcel·la, disposa de dos mètodes “operators” <, que seran necessaris per el mètode sort de les STL que més tard s’utilitzarà. Veiem com s’han creat:

```
friend bool operator<(const parceles& esq, const parceles &dret){
        return esq.valorGrisAcumulat > dret.valorGrisAcumulat;
 }
 friend bool operator<(const parceles& esq, const double dret){
        return esq.valorGrisAcumulat < dret;
}
```

Aquests dos mètodes són necessaris ja que l’algorisme d’ordenació les utilitzarà per fer la comparació entre el valor del valorGrisAcumulat de dos parcel·les diferents.

DistribuïdorGotes només compta amb el constructor, el destructor i un mètode anomenat distribuirles, així com dos paràmetres:

- **vector<parceles> VectorParceles.**
- **double total.**

Per tant vectorParceles estarà creat per un vector d’objectes de **parceles**, que és una classe creada dins del distribuidorGotes. I total que s’explicarà pròximament.

Objectiu de l’algorisme CDF: Distribuir d’una forma coherent però aleatòria la posició de les gotes d’aigua, seguint la distribució indicada per l’usuari.

En aquest cas hem adaptat l’algorisme fent la major part de la feina al constructor de la classe distribuidorGotes, veiem els passos que segueix aquest:

1- Primerament omplirem tot el vector de parceles, per fer-ho es recorrerà pixel per pixel de la imatge d’intensitats per veure si l’usuari ha decidit que plogui o no. Per tant si aquell pixel és diferent de blanc, es guardarà aquest valor d’intensitat a la variable valorGris i al valorGrisAcumulat, a més a més cada vegada aquest valor d’intensitat es va acumulant a la variable total. A les variables x i y li posarem la posició 3D equivalent que la sabrem gràcies al fitxer generat anteriorment per OGRE. Un cop realitzat tot el recorregut ja tindrem omplert tot el vector. A la figura 3.14 veiem un exemple que representa una imatge de 4*4 píxels, i a continuació el vector que s’hauria generat:

	X=0	X=1	X=2	X=3
Y=0				
Y=1		5	9	
Y=2		3	1	
Y=3				

Figura 3.14

X=1 Y=1 ValorGris=5 VGAcumulat=5	2 1 9 9	1 2 3 3	2 2 1 1

Com veiem el vector només serà formada per quatre cel·les de parceles, ja que les de color blanc ja no les ha afegit.

2- Un cop es té el vector de parceles omplert, es fa un recorregut per tot el vector dividint el valor de les variables ValorGris i ValorGrisAcumulat per la variable total. Hauria de quedar així:

Suposem total=18

1 1 0.277 0.277	2 1 0.5 0.5	1 2 0.166 0.166	2 2 0.0555 0.0555

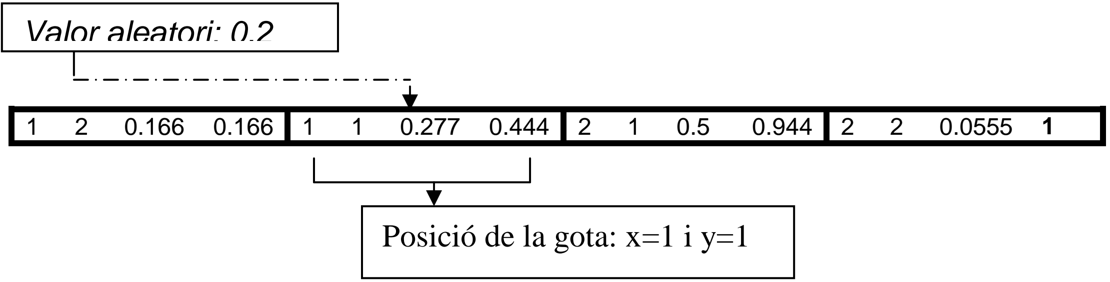
3- Ara s'hauria d'ordenar el vector considerant el valor de la variable ValorGris. L'ordre en què s'ordeni no és important, és a dir tant és de petit a gran, com de gran a petit. En el cas de l'exemple s'ha ordenat de petit a gran.

2 2 0.0555 0.0555	1 2 0.166 0.166	1 1 0.277 0.277	2 1 0.5 0.5

4- Un cop tenim tot el vector ordenat s'ha d'agafar la variable que s'ha anomenat ValorGrisAcumulat i se li van sumant tots els valors que tenen la seva variable de les parceles anteriors a ella. És a dir que l'exemple quedaria així:

1 2 0.166 0.166	1 1 0.277 0.444	2 1 0.5 0.944	2 2 0.0555 **1**

Per saber si l'algorisme ha funcionat de la forma esperada, com es veu en l'exemple realitzat, el valor de l'acumulat de la última cel·la hauria de valer 1. Tot aquest procés ho fa el constructor de la classe distribuidorGotes. Al mètode **distribuirles,** al qual se li passa com a paràmetre el número de gotes que s'han de distribuir, comença generant un número aleatori, el valor del qual ha d'estar entre 0 i 1. Un cop generat, s'ha de fer una cerca i mirar cada valorGrisAcumulat, si aquest és més gran que el valor que s'ha generat de forma aleatòria es mirarà la pròxima cel·la, i així successivament fins a trobar un valor el qual sigui inferior al generat. Quan això passi s'agafarà el valor de les coordenades d'aquella cel·la i aquestes seran les coordenades de la gota. Suposem que volem distribuir una gota de forma aleatòria, mirem un possible resultat a l'exemple (*veure figura 3.15 per observar el resultat final*).



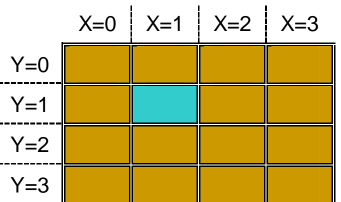


Figura 3.15, aquest requadre representa el terra en l'aplicació 3D, la zona blava representaria la zona en que cauria la gota d'aigua.

Implementació:

Constructor: Com en el mètode de l'apartat anterior, es necessitarà la llibreria Devil per carregar i accedir a cada píxel de la imatge. Un cop ja tinguem seleccionada la imatge que serà tractada agafarem d'aquesta els valors d'amplada i d'alçada:

```
ilGetIntegerv(IL_IMAGE_WIDTH,&he);
ilGetIntegerv(IL_IMAGE_HEIGHT,&wi);
```

Gràcies a un mètode de il.h es podrà obtenir un punter de bytes amb tots els valors de cada píxel, com s'ha definit que la imatge estigui amb el format RGB, obtindrem tres valors per cada píxel, un per el vermell, un per el verd i un altre per el blau. Aquí es pot veure:

```
mida=he*wi;
midaImatge = he * wi * 3;
ILubyte *p_Data=new ILubyte [midaImatge];
p_Data=ilGetData();
```

Tot seguit es procediria a fer el primer dels quatres passos explicats anteriorment. S'omplirà el vector de parceles. Per fer això, com s'ha dit, es recorrerà píxel per píxel tota la imatge, per tant es necessitarà un doble for per recórrala. Abans de fer el doble bucle s'haurà obert el fitxer que s'ha obtingut de l'aplicació d'OGRE que conté totes les posicions. I per tant, ja a dins del doble bucle, i en el cas que el valor del píxel (només caldrà que s'observi un dels tres valors que representarà el píxel, ja que al ser una escala de grisos com els grisos sempre tenen els tres valors iguals seria feina envà comprovar-los tots) de la imatge no sigui igual a 255 (el qual voldria dir que és blanc), s'afegirà aquesta nova parcel·la amb les coordenades d'OGRE llegides del fitxer. Veiem aquesta part:

```
FILE* FitxerDeLectura = fopen("comp.bin", "rb");
double temporal=0;
for (i;i<he;i++){
j=0;
for (j;j<wi;j++){

       fread(&posicioPluja[0],sizeof(float),2,FitxerDeLectura);//llegim i
abancem el cursor
              if (p_Data[color]!=255){//si no és blanc
                     temporal=(0-p_Data[color])+255;//normalitzem a 1, com
més fosc més aprop de l'1 serà
                     //afagim parcela

       VectorParceles.push_back(parceles(posicioPluja[0],posicioPluja[1],te
mporal,temporal));
                     total+=temporal;
              }
       color=color+3;//preparem la variable per mirar el pròxim píxel, per
això li sumem 3
}
}
```

S'ha implementat el segon pas explicat fent un simple for per el vector de parceles, i agafant i dividint les dos variables de cada parcel·la per el total, aquí es pot veure:

```
for (unsigned int t=0;t<VectorParceles.size();t++){
VectorParceles[t].valorGris/=total;
VectorParceles[t].valorGrisAcumulat/=total;
}
```

Gràcies a la classe algorithm s'ha utilitzat el mètode sort per ordenar el vector. Tot seguit s'ha fet l'últim recorregut per tot el vector per donar el valor final de cada ValorGrisAcumulat, veiem com s'ha fet:

```
sort(VectorParceles.begin(),VectorParceles.end());
double acumulat=0;
for (unsigned int p=0;p<VectorParceles.size();p++){
       acumulat+=VectorParceles[p].valorGrisAcumulat;
       VectorParceles[p].valorGrisAcumulat=acumulat;
}
```

Ara al cridar al constructor ja disposarem de tot el vector omplert, ara explicarem com s'ha fet per explicar el mètode **distribuirles.**

El mètode farà un for fins que s'hagin calculat totes les posicions de pluja que s'han demanat per paràmetre. Cada vegada es generarà un número aleatori, a priori sembla una operació senzilla, però com el que s'està buscant és molta exactitud si es fes únicament amb el random de c++ de sempre, només aconseguiríem 2^15=32768 valors diferents, i per tant i haurien moltes parcel·les del vector que mai s'arribarien a ocupar i per tant no es distribuirien les gotes de la forma correcta, a més a més si volguéssim crear per exemple un milió de gotes notaríem que moltes gotes cauen a la mateixa coordenada, ja

que hi haurien, en molts casos, moltes més parcel·les que números possibles. Per tant s'ha tingut d'anar a buscar una manera de generà número aleatoris diferent. Finalment s'ha utilitzat una classe ja existent anomenada **Mersenne Twister** la qual permet generà un total de 2^32 números aleatoris amb una gran velocitat, per tant amb aquesta classe guanyarem moltíssim la precisió. Finalment el número aleatori es generarà així:

```
MTRand test;
int result;
result = test.randInt(total);
valor=result/total;
```

Ara únicament ens quedarà buscar les coordenades d'aquesta gota, comparant el valor aleatori generat amb els de la variable ValorGrisAcumulat, en aquest cas ens hem ajudat altre vegada de les stl, comprovem com s'ha fet en aquesta última part del codi:

```
vector<parceles>::const_iterator que=
lower_bound(VectorParceles.begin(),VectorParceles.end(),valor);
posicioPluja[0]=(*(que)).x;posicioPluja[1]=(*(que)).z;
fwrite (posicioPluja,sizeof(float),2,FitxerPosicionsPluja);
```

Com veiem en la última línia del codi s'escriu en un fitxer totes les posicions que ocuparan cada gota, aquest fitxer serà utilitzat per l'aplicació amb tres dimensions.

3.1.4 Generació d'atles

Com s'ha pogut veure a l'apartat 2.5 necessitem una base de dades d'imatges de gotes de pluja. Aquest conjunt d'imatges l'hem aconseguit descarregant-nos el zip que posa a disposició dels visitants de la pàgina de l'article:

http://www1.cs.columbia.edu/CAVE/databases/rain_streak_db/rain_streak.php

Per poder-les utilitzar d'una manera eficient des de l'algorisme de renderització, es crearà una sola imatge on s'hi guardarà un conjunt d'imatges amb una estructura lògica que ens permetrà recuperar les imatges que pertoquen segons els paràmetres que es vagin obtenint durant l'execució de l'aplicació. Aquesta imatge s'anomenarà atles.

El primer que s'ha hagut de pensar ha set la organització que tindria aquest atles, ja que depenent de com distribuíssim les imatges, guardant el mateix número de imatges l'atles tindria una mida o una altra.

Cal generar dos atles ja que a la base de dades d'imatges de gotes de pluja hi ha dos tipus d'imatges, les de gotes d'aigua de punts de llums i les de llum ambiental.

3.1.4.1 Generació de l'atles de gotes amb llum puntual

Observant l'apartat 2.5.7 i les imatges descarregades corresponents a la base de dades de gotes de pluja, podem extreure les següent conclusions:

- Cada imatge té associat un angle θv, aquest pot tenir 5 valors diferents: 0º, 20º, 40º, 60º i 80º. Cada imatge tindrà una alçada concreta, la qual anirà assignada depenent d'aquest angle, la relació angle θv-alçada de la imatge es pot observar a l'apartat 2.5.8 Com podem comprovar per cada angle li correspon una alçada diferent. Aquest fet serà important a l'hora d'organitzar la distribució de les imatges a l'atles.

- Cada imatge pot tenir 10 angles diferents de tipus θlight. Pot tenir un dels 10 angles següents: -90º, -70º, -50º, -30º, -10º, 10º, 30º, 50º, 70º i 90º

- Cada imatge pot tenir 9 angles diferents de tipus φl. Pot tenir un dels 9 angles següents: 10º, 30º, 50º, 70º, 90º, 110º, 130º, 150º i 170º.

- Hi ha tantes imatges com combinacions d'angles possibles, és a dir, cada combinació possible dels tres angles serà representada per una imatge.

Sabem que totes les imatges tenen la mateixa amplada, que és de 16 píxels. En canvi les alçades poden anar de 525 píxels, la més alta, a 108 píxels la més baixa. Amb aquesta dada, per optimitzar el mida de l'atles, es decideix que hi haurien d'haver el menor número de files d'imatges possibles, ja que les alçades d'aquestes comparant-lo amb les amplades són molt més grans.

Finalment s'ha estructurat l'atles de la següent manera:

Les files d'imatges de l'atles vindran definides per l'angle θv, per tant, l'atles tindrà 5 fileres d'imatges. A cada fila hi haurà la següent estructura: per cada angle θlight s'hi inserirà a l'atles les nou imatges corresponents als nou angles φl.

Per tant podem concloure que les dimensions de l'atles seran les següents:

- Amplada: Per cada angle de tipus θlight hi hauran 9 angles φl. Com d'angles de tipus θlight n'hi ha 10 de diferents tindrem en total 10*9 imatges, és a dir 90. Si cada imatge fa 16 píxels d'amplada tindrem que l'atles tindrà una amplada de 16*90=1440 píxels.
- Alçada: Cada fila tindrà una alçada diferent de la següent fila. Per tant, per saber l'alçada de l'atles cal sumar les 5 alçades diferents que tindrem. 525+494+405+272+108=1804 píxels.

A l'apartat 3.1.2.1.2 classe MyFrame, s'ha pogut observar que es disposava d'una funció anomenada OnCreateAtlas, en aquesta es fa un recorregut per tot el directori on hi ha guardades totes les imatges de la base de dades i va construint tot el mosaic. Per fer el recorregut s'ha observat el nom que tenien

les imatges per així poder determinar quina imatge era i a on s'havien de posar. El nom de les imatges segueix un patró que permet descriure les característiques de la gota. Veiem un exemple:

cv80_v10_h10_osc0

- cv80, ens indica que aquesta imatge tindrà d'angle θv 80º.
- V10, ens indicarà que l'angle θlight és igual a 10.
- H10, ens indica que l'angle φl és igual a 10.
- Osc0, ens indica que de les 10 oscil·lacions que hi ha per cada combinatòria dels tres angles, aquesta és la primera de les 10. En el nostra altes, només s'utilitzarà una oscil·lació de les 10 que hi ha a la base de dades, la qual s'escollirà de forma aleatòria, ja que si utilitzéssim les 10 oscil·lacions l'algorisme de renderització perdria eficiència i l'atles augmentaria massa de mida.

Així doncs a l'algorisme es fa un recorregut amb tres bucles els quals controlaran els tres angles diferents. Primerament es llegirà la imatge, després s'escriurà la imatge allà on apunti el punter cap al fitxer, s'actualitzarà el punter fent-lo apuntar allà on li pertoca i finalment s'actualitzarà el nom de la pròxima imatge que cal llegir. Veiem el codi:

```
//augmento langle de la camera, que és de 0-80
for (int camera=0;camera<91;camera=camera+20)
{
//augmento langle de la llum V, que és de -90 a 90
    for (int llumV=-90;llumV<91;llumV=llumV+20)
    {
      //augmento langle de la llumH, que és fins a 10-170
        for(int llumH=10;llumH<171;llumH=llumH+20)
        {
sprintf(gota,".\\BaseDeDadesGotes\\cv%d_v%d_h%d_osc.jpg",camera,llumV,llumH
);
FIBITMAP *image=FreeImage_Load(FIF_JPEG, gota, 0);
//copio la imatge,amb les mides k te. anles d llum de v_90
FIBITMAP *dib=FreeImage_Copy(image, 0, 0, FreeImage_GetWidth(image),
FreeImage_GetHeight(image));
//enganxem la imatge a la posicio indicada, si no ha anat be, copiem una
imatge negra(sense imatge de gota) ja que no haura trobat akella imatge i
lenganxem
if (!FreeImage_Paste(imagedst, dib, amplada, alsada,256)){
//com per la resta d'angles de llumH,només hi ha una imatge ja que totes les
aparences són iguales, si no troba la imatge que vagi a buscar la següent que és la
que hi pertocaria
sprintf(gota,".\\BaseDeDadesGotes\\cv%d_v%d_h%d_osc.jpg",camera,llumV,170);
image=FreeImage_Load(FIF_JPEG, gota, 0);
dib=FreeImage_Copy(image, 0, 0, FreeImage_GetWidth(image),
FreeImage_GetHeight(image));
FreeImage_Paste(imagedst, dib, amplada, alsada,256);
            }
amplada=amplada+16;//augmento punter amplada
        }
    }
   amplada=0;  //ja que ara posarem fotos a sota de les que hi hem posat, i per tant
altre cop posem les imatges al ppi esquerra
```

També tinc de modificar el punter que marcarà la posició inicial de l'alçada, com depenent de la foto es té una alçada o una altra, ha calgut realitzar els següents condicionals:

```
//augmento punter alsada, segons on sigui s'augmenta duna manera o     altra el
punter
      if (alsada==0)
        alsada=alsada+525;
     else if (alsada==525)
        alsada=alsada+494;
     else if (alsada==1019)
        alsada=alsada+405;
     else if (alsada==1424)
        alsada=alsada+272;
     else if (alsada==1696)
        alsada=alsada+108;
}
FreeImage_Save(FIF_JPEG, imagedst, "atles.jpg", 0);//gravem latles
```

Podem observar l'atles generat a la figura 3.16.

3.1.4.2 Generació de l'atles de gotes amb llum ambiental

Generar l'atles de gotes amb llum ambiental és més senzill que amb les de llums puntuals, ja que aquestes depenen de menys paràmetres. Com acabem de veure en l'anterior apartat, aquestes depenien del valor de tres angles. Amb les imatges de gotes amb llum ambiental, només depenem de l'angle θv.

Per tant només tindrem una columna d'imatges i cinc files, una per cada angle θv.

La funció que s'ha pogut veure a l'apartat 3.1.2.1.2 classe MyFrame anomenada OnCreateAtlasAmbient, genera l'atles de gotes amb llum ambiental. En aquesta ocasió com ja s'ha dit només depenem d'un angle, per tant només necessitarem un sol bucle. Veiem el codi:

```
       int amplada=0,alsada=0;
       //augmento langle de la camera, que és de 0-80
       for (int camera=0;camera<91;camera=camera+20)
       {
             sprintf(gota, ".\\BaseDeDadesGotesAmbient\\cv%d.jpg",
       camera);
             FIBITMAP *image=FreeImage_Load(FIF_JPEG, gota, 0);
             FIBITMAP*dib=FreeImage_Copy(image,0,0,FreeImage_GetWidth
             (image), FreeImage_GetHeight(image));
             //enganxem la imatge a la posicio indicada
             FreeImage_Paste(imagedst, dib, amplada, alsada,256);
```

Com ha passat en l'anterior funció, en aquesta també s'ha de modificar el punter que indica la posició inicial per el que fa a l'alçada de la pròxima imatge

que s’ha d’enganxar. Com en l’anterior cas, les imatges tenen mides diferents, i és per això que s’ha hagut d’implementar mitjançant condicionals:

```
//augmento punter alsada, segons on sigui s'augmenta duna manera o altra el punter
if (alsada==0)
      alsada=alsada+525;
else if (alsada==525)
      alsada=alsada+494;
else if (alsada==1019)
      alsada=alsada+405;
else if (alsada==1424)
      alsada=alsada+272;
else if (alsada==1696)
      alsada=alsada+108;
}
      //gravem latles
FreeImage_Save(FIF_JPEG, imagedst, "atlesAmbient.jpg", 0);
```

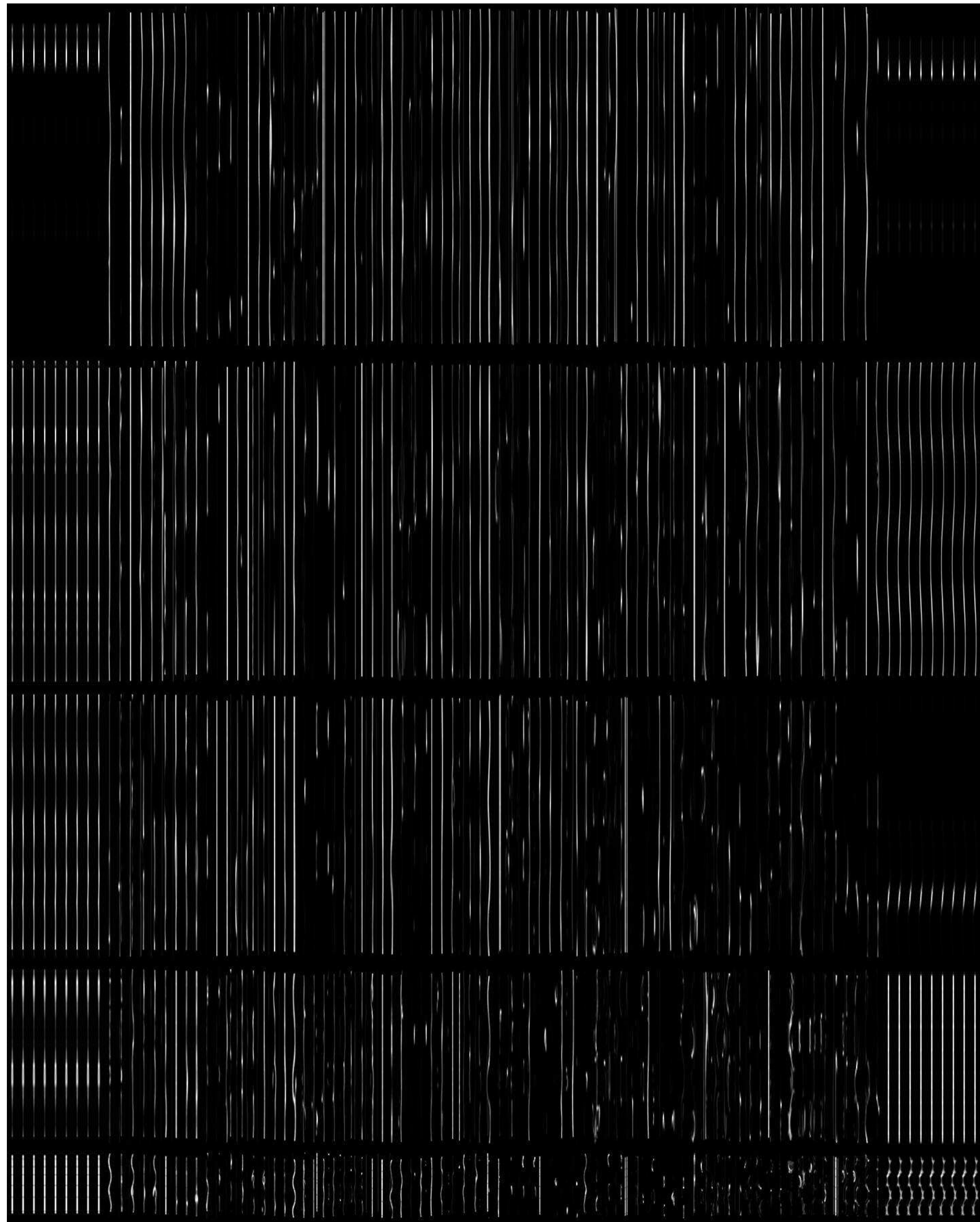


Figura 3.16, atles de gotes de pluja de llums puntuals.

3.2 Pluja en temps real

En aquest apartat s'explicarà detalladament què fa l'aplicació que s'ha desenvolupat de pluja en temps real. A més a més s'explicarà quins mètodes, classes o algorismes permeten aquests resultats obtinguts.

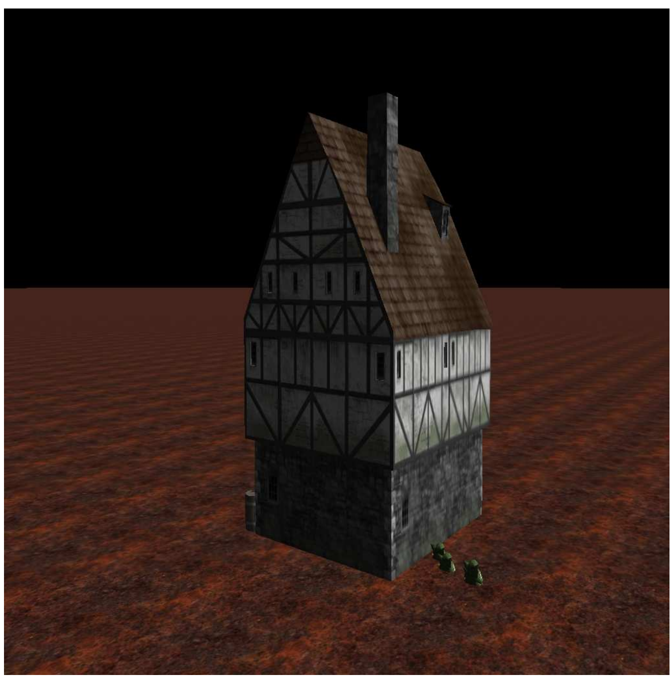


Figura 3.16, imatge capturada de l'aplicació en funcionament. L'escenari que es pot veure serà l'utilitzat per crear totes les proves necessaris per comprovar el bon funcionament de la pluja en temps real.

Com ja s'ha explicat en apartats anteriors, l'objectiu d'aquesta aplicació serà la de realitzar la simulació de pluja en temps real al motor Ogre3D. Per aconseguir-ho, abans s'han de seguir uns passos per que les gotes caiguin al lloc desitjat i de la forma correcta:

- **Obtenció de la imatge de l'escenari**. Necessitarem obtenir la imatge de l'escenari des d'una vista ortogràfica cenitial. Per tant l'aplicació desenvolupada disposarà d'una opció la qual, al prémer una certa tecla de l'ordinador, generarà una imatge amb un nom per defecte. Aquesta imatge generada ens serà útil per després poder editar-la amb l'aplicació d'edició (explicat a l'apartat anterior).

- **Obtenció del mapa de profunditat**. Al mateix temps que capturem la imatge de l'escenari, aprofitarem per generar una segona imatge, formada per una escala de grisos. El to de gris vindrà definit per l'altura a la qual es troba l'objecte dibuixat. És a dir, si s'està dibuixant el terra de l'escenari, aquest es veurà blanc, ja que aquesta serà l'altura màxima de que es disposarà. Per contra, si s'està dibuixant el sostre d'un gratacels, aquest es dibuixarà amb un to de gris molt fosc, ja que l'alçada que hi haurà

entre el sostre i la posició en què s'ha capturat la imatge serà molt petita (comparada amb la que hi ha entre el terra i la posició de captura de la imatge).

- **Obtenció de les posicions del mon real**. Apart de capturar la imatge de l'escenari i la imatge de profunditat, al mateix temps es crearà un arxiu al qual es dipositaran totes les posicions en mon real de l'escenari, ja que aquestes es necessitaran a la part en què l'editor fa la distribució de gotes *(veure documentació apartat 3.1.3.2 Calcular les posicions de pluja)*. Ja que si no es creessin, no tindríem cap possibilitat de saber quines posicions en món real són les que es veuen a la imatge.

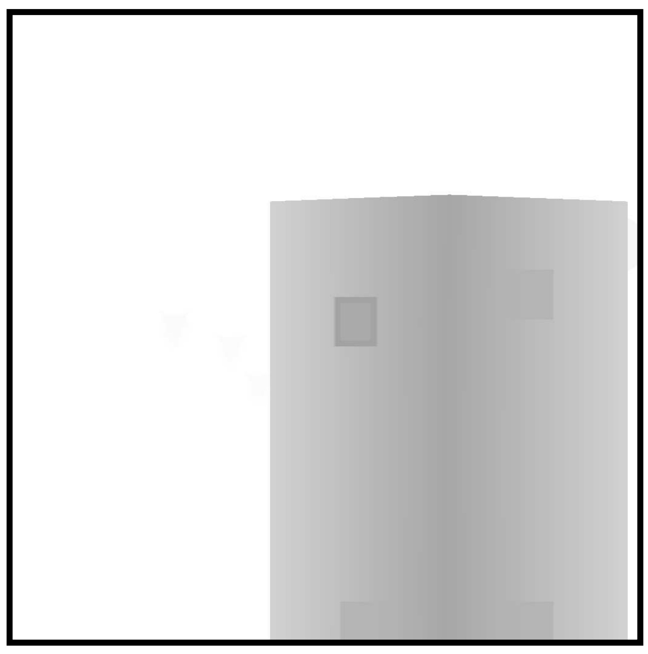


Figura 3.17, exemple del resultat d'obtenir la profunditat de l'escenari. S'ha agafat la profunditat de l'escenari des del cel.

- **Creació dels punts (ManualObjects).** Serà la part encarregada de crear els punts els quals simularan la pluja. S'agafarà el fitxer generat per l'editor per saber en quines posicions del mon tridimensional s'han de col·locar aquests punts (o gotes) inicialment.

- **Creació del material**. En aquesta part, es defineix com es comportaran aquests punts. Per tant es desenvoluparà un .material i un .program a més a més d'un vèrtex shader per aconseguir aquest objectiu.

Tot seguit s'explicaran les classes creades i el seu funcionament:

3.2.1 Classes creades

Aquestes són les classes que s'han creat:

- Classe MyApplication
- Classe MyFrameListener
- Classe CalculProfunditat
- Classe ParamatresShader

Tot seguit s'explicarà com han estat construïdes i la seva utilitat.

3.2.1.1 Classe MyApplication

Aquesta classe s'encarregarà de crear el mon tridimensional en el que treballarem, és a dir, s'encarrega de crear totes les entitats de l'escenari, de crear els llums i posicionar-los en la direcció i posició adequats, de crear les càmeres, d'afegir el frameListener que s'ha creat, etc. Serà l'encarregada de cridar la funció d'arrencada (és a dir, l'encarregada d'arrencar l'aplicació) la qual ha heretat de la classe base.

És una classe derivada de l'exampleApplication, classe la qual ja ve definida amb el codi d'Ogre3D. Disposa de varis objectes per així fer més fàcil la creació de les altres classes derivades d'aquesta. Els objectes de que disposa són:

- Root *mRoot.
- Camera* mCamera.
- SceneManager* mSceneMgr.
- ExampleFrameListener* mFrameListener.
- RenderWindow* mWindow.

La classe MyApplication disposa dels següents paràmetres els quals estan declarats com a protected:

- Viewport* mViewport: un objecte viewport es podria definir com el rectangle en el qual s'envia la renderització de sortida.
- CalculProfunditat* mCalculProfunditat: punter a la classe creada CalculProfunditat, la qual serà explicada pròximament *(veure apartat 3.2.1.3 Classe CalculProfunditat)*
- RenderTexture* m_renderTexture: punter a un objecte RenderTexture, aquest objecte s'encarrega de renderitzar les textures creades del mon tridimensional el qual s'està executant.

I aquests són els mètodes de que disposa:

- virtual void createScene(): Implementarà el mètode que ha estat definida en la classe base. En aquest es crea tota l'escena, és a dir les entitats, els llums, les càmeres, a més a més posa la posició adequada a la qual es vol que enfoqui la càmera i especifica els paràmetres de cada objecte, com per exemple el color dels llums i el tipus de llum. També instància l'objecte m_renderTexture i el mCalcularProfunditat, per després poder-los passar com a paràmetre en l'objecte myFrameListener que es crearà en el pròxim mètode, i així poder utilitzar-los en aquella classe. Veiem com s'han creat els objectes:

```
mCalculProfunditat = new CalculProfunditat(mViewport,alturaMa,alturaMi);
m_renderTexture=mRoot->getRenderSystem()->

createRenderTexture("RttTex",mWindow->
getWidth(),mWindow->
getHeight(),TEX_TYPE_2D,PF_R8G8B8);
```

Veiem que com a paràmetres de mCalculProfunditat s'hi passen tres paràmetres, els quals seran explicats més a fons en els propers capítols:

- mViewport, el qual és el viewport de que es disposarà.
- alturaMa, que és un enter que indica l'altura a la qual està la càmera.
- alturaMi, que, com el seu nom indica ,és l'enter que indicarà què definirà la posició mínima de l'escenari (en el cas de l'aplicació desenvolupada aquesta posició serà el terra, és a dir y=0).

A l'hora de crear m_renderTexture, veiem que primer s'utilitza un mètode de l'objecte mRoot, de classe root, el qual obté un objecte de tipus RenderSystem gràcies a la funció getRenderSystem, per així poder després, a partir d'aquest objecte obtingut, cridar la funció createRenderTexture. En total se li passen cinc paràmetres:

  - RttTex serà el nom que se li assigna.
  - Tot seguit se li defineixen l'amplada i l'alçada agafant els valors actuals de l'objecte mWindow.
  - Finalment se li assigna el tipus de textura que en aquest cas serà de 2D.
  - El format dels píxels, en aquest cas se li han assignat que seran del tipus RGB amb 8 bits per cada color.

- virtual void createFrameListener(): Igual que el mètode anterior, s'implementa el mètode createFrameListener definit a la classe base. L'objectiu d'aquest mètode és crear el nostre propi frameListener (MyFrameListener), i registrar-lo per què així sigui cridat a cada frame realitzat. Veiem com s'ha fet:

```
mFrameListener = new MyFrameListener(mWindow, mCamera, mSceneMgr,
                                     mCalculProfunditat,m_renderTexture);
mFrameListener->showDebugOverlay(true);
mRoot->addFrameListener(mFrameListener);
```

MyframeListener, el qual com s'ha dit és un objecte de tipus MyFrameListener, se li passen un total de cinc paràmetres per instanciar-lo:

- Se li passa l'objecte window.
- La càmera que s'està utilitzant.
- L'sceneManager utilitzat.
- L'objecte CalculProfunditat.
- L'objecte RenderTexture.

- virtual void destroyScene(): Novament s'implanta un mètode de la classe base, en aquest cas la seva funció és destruir l'objecte mCalculProfunditat al finalitzar l'execució de l'aplicació.
- void createCamera(void): Com diu el seu nom, aquest és el mètode encarregat de crear la càmera. Veiem com s'ha fet:

```
mCamera = mSceneMgr->createCamera("PlayerCam");
```

Com es pot veure s'utilitza el mètode d'un objecte sceneManager per cridar el mètode cretateCamera, el qual així ja estarà lligada a aquest objecte sceneManager *(veure apartat 2.2.1 Objectes principals d'OGRE per més informació),* el qual s'hi passarà el nom que tindrà la nova càmera.

- void createViewports(): Aquest mètode afegeix un viewport al renderTarget.

```
mViewport = mWindow->addViewport(mCamera);
```

Aprofita el mètode d'un objecte RenderWindow per a cridar el mètode addViewport, per paràmetre si passa la càmera, amb aquest mètode s'obté també un viewoprt.

3.2.1.2 Classe MyFrameListener

Aquesta classe hereta de la classe exampleFrameListener, la qual ja ve definida amb les classes d'ogre per facilitar la creació de frameListeners. L'exampleFrameListener és una classe que disposa de dues classes bases, la classe FrameListener explicada en temes anteriors i la classe keyListener, que és la classe encarregada d'escoltar si es produeix algun event al prémer-se alguna tecla del teclat. Per saber més a fons que es podrà fer amb l'aplicació que s'ha creat primerament es veuran els aspectes més destacats de la classe exampleFrameListener.

Moviment de la càmera: La classe exampleFrameListener disposa d'un vector3 el qual se l'ha anomenat mTranslateVector.

```
if (mInputDevice->isKeyDown(KC_A))
        {
        // Move camera left
            mTranslateVector.x = -mMoveScale;
    }
```

Al prémer una de les tecles seleccionades per fer avançar, anar a la dreta, endarrere o a l'esquerra, gràcies al frameEvent que s'ha generat, entra en el mètode de que disposa: **processUnbufferedKeyInput** i comprova si s'han premut certes tecles. Si, per exemple, la tecla que s'ha premut és la A (voldrà dir que l'usuari desitja desplaçar-se a l'esquerra), entrarà en el seu condicional (el que veiem en l'exemple anterior), i restarà al vector mTranslateVector de l'eix de coordenades x, el valor corresponent a mMoveScale. Vegem-ho:

La variable mMoveScale és un float, el qual s'encarregarà de guardar el desplaçament que s'ha produït, això ho farà aplicant la fórmula de la posició:

$$X=V*t$$

La velocitat és una variable que s'inicialitza en el constructor i és un float que s'anomena mMoveSpeed. Per exemple, en el nostre cas, s'ha inicialitzat a 400 (per defecte està a 100 però per a una millor usabilitat s'ha ampliat a 400) per així aconseguir un moviment bastant ràpid. El temps, com ja s'ha explicat en el tema de fonaments previs, serà una variable que calcularà el temps que porta amb la tecla premuda. Per tant, si sabem que el temps es guarda amb la variable anomenada timeSinceLastFrame i la velocitat és la variable mMoveSpeed, ja es pot aplicar la fórmula:

```
mMoveScale = mMoveSpeed * evt.timeSinceLastFrame
```

Ara que ja sabem com es desplaça la càmera, de la nostra aplicació ens centrarem amb la nova classe que s'ha creat.

MyFrameListener disposa dels següents paràmetres definits com a protected :

Variables:

- RenderTexture * m_renderTexture: definim un objecte RenderTexture.
- CalculProfunditat* mCalculProfunditat: definim un objecte CalcularProfunditat.
- paramatresShader * params: definim un objecte, paramatresShader, el qual serà explicat en els pròxims capítols.
- SceneManager *mSceneMgr: definim un objecte sceneManager per poder incorporar els manualsObjects al mon tridimensional que s'ha creat al createScene de la classe MyApplication.

Mètodes:

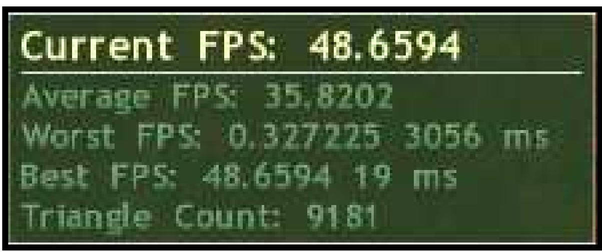
- **Constructor:** Aquí podem veure el constructor:

```
MyFrameListener::MyFrameListener(RenderWindow* renderWindow, Camera*
camera, SceneManager* sceneManager,
CalculProfunditat* dof,RenderTexture *rText)
     : ExampleFrameListener(renderWindow, camera,false, false)
       , mCalculProfunditat(dof)
{
       m_renderTexture=rText;
       mSceneMgr=sceneManager;
}
```

Aquest constructor cridarà al constructor còpia de la classe CalculProfunditat gràcies a que se li ha passat per paràmetre (serà el que s'ha definit en el mètode cretateScene de la classe MyApplication) i assignarà el RenderTexture passat per paràmetre a l'objecte RenderTexture anomenat m_renderTexture.

- **frameStarted:** El mètode, al començar, crida el mètode de la seva classe base. El mètode realitzarà les següents tasques:

  - Controlarà la modificació de la posició de la càmera.
  - Disminuirà el temps de la variable mTimeUntilNextToggle, utilitzada per evitar la repetició de les tecles, és a dir, que al prémer una tecla, com el dit tarda algunes dècimes en tenir la tecla premuda l'aplicació no executés vàries funcions alhora.
  - S'encarregarà de controlar si l'usuari vol mostrar la informació de la velocitat dels frames o no *(veure figura 3.18)*.



Fingura 3.18, finestra opcional la qual mostra a l'usuari la velocitat de l'aplicació mostrada amb quadres per segon.

  - Controlarà si l'objecte RenderWindow s'està aplicant o no.

Un cop acabada aquesta funció, el mètode assignarà a l'objecte de CalculProfunditat, l'actual altura de la càmera.

Tot seguit aquest mètode controlarà si s'han premut certes tecles per realitzar algunes funcions:

  - Al prémer-se la tecla "return", es controlarà quina vista s'està utilitzant, si s'està utilitzant la vista panoràmica es canviarà a l'ortogonal i al revés. Veiem com controla s'hi s'ha premut aquesta tecla:

```
if ((mTimeUntilNextToggle <= 0.0f) && mInputDevice->
isKeyDown( KC_RETURN ) )
```

Passarà el condicional en el cas de que la variable mTimeUntilNextToggle sigui inferior o igual a zero i aplicant el mètode de l'objecte mInputDevice, el qual és del tipus ImputReader, isKeyDown(KC_RETURN), en aquest cas KC_RETURN és la tecla return.

- Al prémer-se la tecla "2" acosta 1000 unitats la posició del clipping sobre el frustum (*explicat a l'apartat 2.1.1 descripció del procés de renderització*). Veiem com ho fa:

```
sn=sn-1000;
sf=sf-1000;
mCamera->setNearClipDistance(sn);
mCamera->setFarClipDistance(sf);
```

Sn i sf són dos reals, els quals guardaran el valor de la distància entre el clipping sobre el frustum.

- Al prémer-se la tecla "3", s'allunya en 1000 unitats la posició del clipping sobre el frustum.
- Al prémer-se la tecla "9", es crearan els punts que seran utilitzats per representar les gotes d'aigua. Aquests punts es crearan amb l'objecte manualObject. Tot seguit s'agafaran les posicions que s'han calculat a l'editor *(apartat 3.1.3.2calcular les posicions de pluja)* i s'assignaran com a posició d'aquests punts. Aquests s'assignaran al material que s'ha creat (veure apartat 3.2.1.2). Finalment es crearà l'objecte paramatresShader.

Per aconseguir-ho, primerament es tractarà el fitxer de que disposem:

```
fseek(FitxerDeLectura,0,SEEK_END);
int tamanyTot = ftell(FitxerDeLectura);
rewind(FitxerDeLectura);
int tamanyFinsAPunter=0;
```

Es posarà primer el punter al final del fitxer (gràcies a fseek i el paràmetre SEEK_END), i obtenim la mida amb el mètode ftell. Un cop obtinguda la mida, tornarem a posar el punter al principi del fitxer de posicions i ja podrem crear els punts que simularan les gotes, i se'ls assignaran les posicions del fitxer.

```
ManualObject *mObjects=mSceneMgr->createManualObject("manualOb");
mObjects->estimateVertexCount((int)(2*(tamanyTot/10)));
mObjects->begin("pluja/pluja",RenderOperation::
OT_POINT_LIST);
while (tamanyFinsAPunter<tamanyTot){
      fread(&posicioPluja[0],sizeof(float),2,
                                  FitxerDeLectura);
      mObjects->position(posicioPluja[0],MINIM + rand()
      % (MAXIM-MINIM),posicioPluja[1]);
      tamanyFinsAPunter=ftell(FitxerDeLectura);
       }
```

Primer es crea un ManualObject (objecte que permet crear figures geomètriques al mon tridimensional). Tot seguit s'inicia el vèrtex amb el mètode begin, al qual se li assigna el material desitjat (en aquest cas serà el del vèrtex shader). Finalment, mitjançant un while, s'aniran creant punts fins haver llegit tot el fitxer. Això ho sabrem ja que anirem comparant l'espai que ocupa el fitxer fins el lloc per on passa el punter, si és igual a la mida total del fitxer, voldrà dir que ja s'han creat tots els punts. Com es veu, el manualObject, disposa d'una funció per assignar-se posicions. Veiem que a la coordenada x i z (primer i tercer paràmetre) li assignarem una posició agafada del fitxer, i al segon (eix de coordenades y i, per tant, l'altura d'aquest punt respecte al terra) se li assignarà un enter aleatori, ja que la següent funció genera un número aleatori entre els valors MINIM i MÀXIM (els quals estan definits):

```
MINIM + rand() % (MAXIM-MINIM)
```

Les gotes d'aigua es posaran entre el cel i la terra, per aconseguir el realisme desitjat, ja que si no ho féssim així o únicament les poséssim a un interval inferior no es veuria un flux de pluja, sinó que es veurien un conjunt de gotes que baixen amb bloc. A la figura 3.19 es pot observar un exemple de què passaria si les gotes d'aigua no es col·loquessin entre una posició aleatòria entre el cel i la terra.

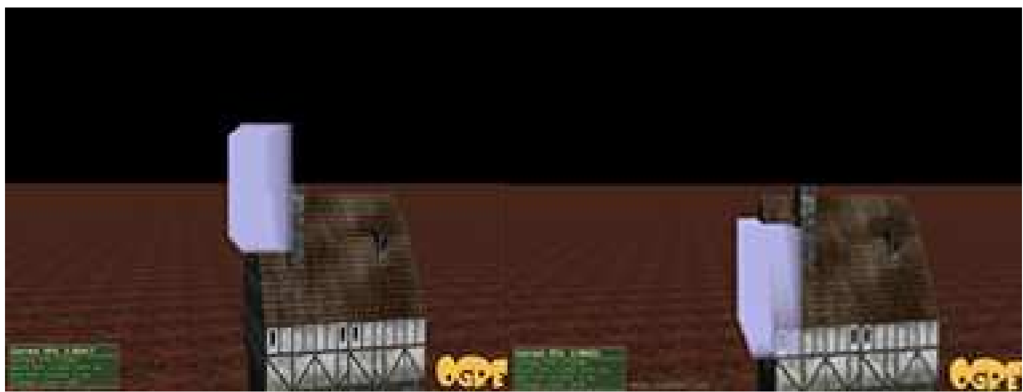


Figura 3.19, exemple de la caiguda de gotes d'aigua en bloc, per tant observem que no és el resultat desitjat.

Un cop ja estiguin totes els punts generats, s'indicarà al manualObject que ja s'ha acabat de dibuixar figures (amb el mètode end), i per tant, com tot objecte que estigui a l'escenari, s'haurà de lligar a l'objecte sceneManager:

```
mObjects->end();
mSceneMgr->getRootSceneNode()->createChildSceneNode(
Vector3(0.0, 0.0, 0.0))->attachObject(mObjects);
```

Al prémer l'espai, es realitzaran tres tasques diferents (Cal dir que les tasques que realitza, s'han de fer totes amb el mode de càmera orthographic):

Descripció:

1. Es capturarà la imatge que s'està veient des de la càmera *(veure figura 3.20)*.
2. Es generarà el mapa de profunditats (*Veure figura 3.21).*
3. Es generarà un fitxer amb totes les coordenades de la imatge capturada, però no seran coordenades 2d (que serien les de la imatge), sinó les del mon real, és a dir les de l'aplicació Ogre3D.

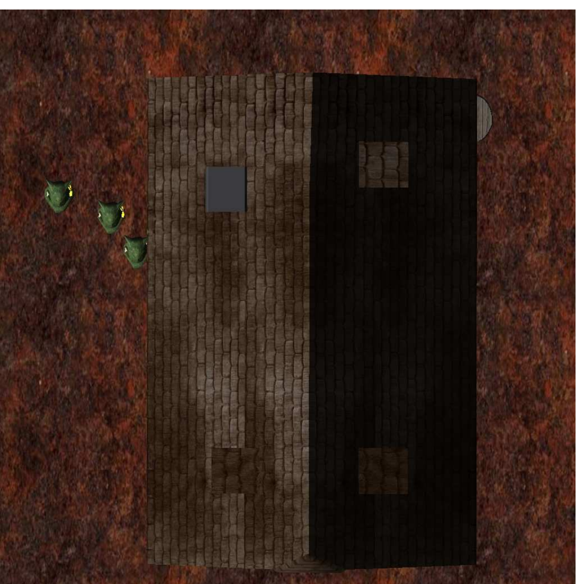


Figura 3.20, Imatge capturada des de la posició de la càmera, com es veu aquesta posició seria des de dalt del cel, i el que s'observa és la teulada d'una casa, ja que així es podrà editar de la millor forma a l'editor.

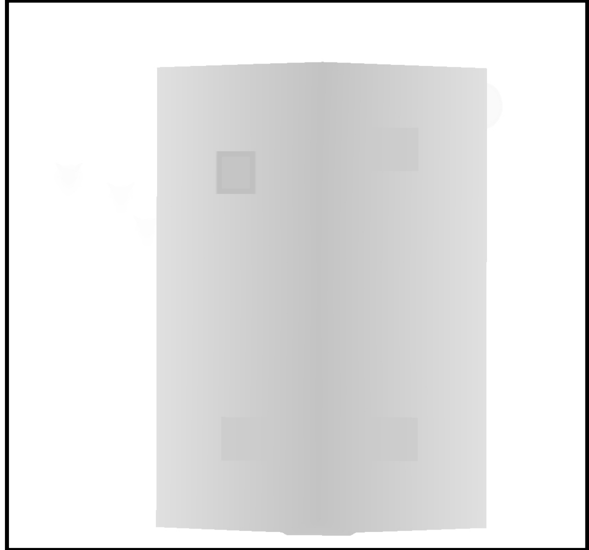


Figura 3.21, mapa de profunditats, com es pot veure és la mateixa imatge de la figura 3.20 però amb l'escala de grisos que indiquen la profunditat de la imatge capturada. Per exemple es pot veure que les xemeneies són la part més fosca, a causa que són les que estan més aprop de la càmera.

Implementació

1. Per capturar la imatge, primerament a l'objecte RenderTexture: m_renderTexture se li ha afegit la càmera que s'està utilitzant, després força que l'objecte root cridi el mètode startRendering. A partir d'aquí l'objecte RenderTexture ja disposa de tota la informació de l'escena. Tot seguit s'utilitzarà el mètode per guardar aquesta informació en un fitxer, aquest mètode és el writeContentsToFile:

```
m_renderTexture->addViewport(mCamera);
m_renderTexture->update();
m_renderTexture->writeContentsToFile(tmp);
```

2. Com s'ha dit a l'apartat 3.2.1.1 Classe MyApplication, en el mètode createScene es crea l'objecte CalculProfunditat i se li passen tots els paràmetres que necessitarà (*veure el tema 3.2.1.3 Classe CalculProfunditat)*. Tot seguit se'l passa com a paràmetre per així disposar d'ell en aquesta classe. En concret al prémer la tecla espai, el que es farà serà crear una textura a partir dels materials (on actuaran els shaders creats). Tot seguit obtenim l'objecte RenderTexture a partir de la textura creada i finalment tornem a capturar la imatge de l'objecte RenderTexture, aquest n'és el codi:

```
TexturePtr depthTexturee =
(TexturePtr)TextureManager::getSingleton().
         getByName("DoF_Depth");
RenderTexture* depthTarge =depthTexturee->getBuffer()->
                     getRenderTarget();
depthTarge->writeContentsToFile("d.jpg");
```

3. Per aconseguir les dades, farem un recorregut per tots els píxels de la pantalla. Per saber la resolució d'aquesta es definirà una variable de tipus image, es carregarà la imatge que s'acaba de generar al capturar la imatge de l'escenari i s'obtindrà l'alçada i l'amplada d'aquesta. Veiem aquesta part:

```
Image im;
im.load("jo.jpg",ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
int height=(int)im.getHeight();
int width=(int)im.getWidth();
```

Ara ja es podrà fer el recorregut per saber totes les coordenades 3d de l'aplicació Ogre3D. Per realitzar això es realitzarà un doble bucle per fer el recorregut. Per obtenir la coordenada del mon real, primerament s'obtindrà un objecte de tipus Ray. Per fer això, l'objecte càmera disposa d'una funció anomenada getCameraToViewportRay, la qual donant les posicions x i y de la pantalla retornarà un objecte de tipus Ray. Ray és un objecte que genera una línia amb un origen i

una direcció. Aquest objecte disposa d'un mètode (getPoint) el qual se li passa un real per paràmetre i aquest, sumant aquest real a la línia que seguia, retorna a quina posició (posició del mon real) es troba. En el nostre cas, aquest paràmetre ens interessava que fos igual a zero. Aquest n'és el codi:

```
a=mCamera->getCameraToViewportRay((float)j/width,
                                  (float)i/height);
aa=a.getPoint(0);
pos[0]=aa.x;pos[1]=aa.z;
```

Les variables "j" i "i" utilitzades en el getCameraToViewportRay són les que s'encarreguen de fer el recorregut de la imatge, i es divideixen per les variables que conten l'amplada i l'alçada (width i height) ja que la funció necessita valors entre 0 i 1. Finalment les variables "pos" a les quals s'ha guardat les posicions x i z es guardarà en un fitxer.

3.2.1.3 Classe CalculProfunditat

Aquesta classe ha estat creada per poder acabar de crear el mapa de profunditats de la nostra aplicació. La classe hereta de dos classes base: la renderTargetListener i la RenderQueue::RenerableListener *(veure tema 2.2.1 Objectes principals d'Ogre).* Disposa de les següents variables:

Variables:

- Viewport* mViewport: Per tenir el viewport que s'està utilitzant.
- MaterialPtr mDepthMaterial: Objecte que guardarà el material de la profunditat calculada.
- Technique* mDepthTechnique: Objecte que serà utilitzat per guardar la tècnica que s'utilitzarà.
- float màxim: Guardarà la posició a la qual estarà la càmera, i per tant, ja que ha de fer una foto des del cel per capturar l'escena desitjada, serà la posició màxima.
- float mínim: Serà la posició de l'objecte a que es trobi a una posició més baixa respecte l'eix de coordenades de la y.

Mètodes:

- **CalculProfunditat(Ogre::Viewport* viewport,float max, float min):** Es cridarà el mètode de la pròpia classe cretateDepthRenderTexture i s'assignaran a les variables màxim i mínim els paràmetres passats.
- **~CalculProfunditat():** Destructor de la classe.
- **void setMaxim(float &ma):** Assigna la variable passada per referència a la variable màxim.
- **float getAltures():** Obté l'altura màxima de la càmera.

- **void createDepthRenderTexture():** Crearà una textura de la profunditat de l'escena. Veiem com ho fa:

```
TexturePtr depthTexture = TextureManager::getSingleton().createManual(
   "DoF_Depth", ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME,
   TEX_TYPE_2D, mViewport->getActualWidth(), mViewport->getActualHeight(),
    0, PF_L8, TU_RENDERTARGET);
```

Creem una textura de manera manual escollint el material que utilitzarem (DoF_Depth), i les propietats d'aquesta, com per exemple l'amplada i l'altura. Tot seguit d'aquesta textura s'obtindrà el renderTexture, i a partir d'aquesta el viewport que s'utilitza, a més a més registrem la classe com una classe target listener. Tot això s'ha fet així:

```
RenderTexture* depthTarget = depthTexture->getBuffer()->getRenderTarget();
Viewport* depthViewport = depthTarget->addViewport(mViewport->getCamera());
depthTarget->addListener(this);
```

Tot seguit obtindrem la tècnica òptima per després utilitzar-la a l'hora de renderitzar la textura de profunditat render texture.

```
mDepthMaterial = MaterialManager::getSingleton().getByName("DoF_Depth");
mDepthMaterial->load();
mDepthTechnique = mDepthMaterial->getBestTechnique();
```

Com es veu, primerament assignarem a la variable de la classe el material que s'obtindrà després de generar el material des dels shaders que s'han creat en el material. Tot seguit es carregarà manualment (load) i obtindrem la millor tècnica de que disposarem.

Ara es generarà un RenderQueueInvocatSequence, el qual és un objecte que representa el total de les invocacions del viewport, que servirà per assegurar que aquestes instàncies siguin destruïdes. S'ha fet de la següent manera:

```
  RenderQueueInvocationSequence* invocationSequence
=Root::getSingleton().createRenderQueueInvocationSequence("DoF_Depth");
RenderQueueInvocation* invocation = invocationSequence-
>add(RENDER_QUEUE_MAIN,
  "main");
  invocation->setSuppressShadows(true);
depthViewport->setRenderQueueInvocationSequenceName("DoF_Depth");
```

- **void destroyDepthRenderTexture():** Crida el mètode de la classe base per destruir el renderTexture.
- **virtual void preViewportUpdate(const Ogre::RenderTargetViewportEvent& evt):** S'implementa el mètode de la classe base RenderTargetListener. Serà el mètode encarregat de passar els paràmetres al vèrtex i fragment shader, abans d'actualitzar-se el viewport. En aquest cas s'hi passaran els paràmetres de l'altura

mínima i màxima de l'escenari, perquè així els shaders puguin crear l'escala de grisos segons aquest patró, és a dir, com més a prop de l'altura màxima es trobi un objecte més fosc serà el gris. Veiem a les figures 3.21 i 3.22 dos exemples del mateix escenari vist des de dos altures diferents:

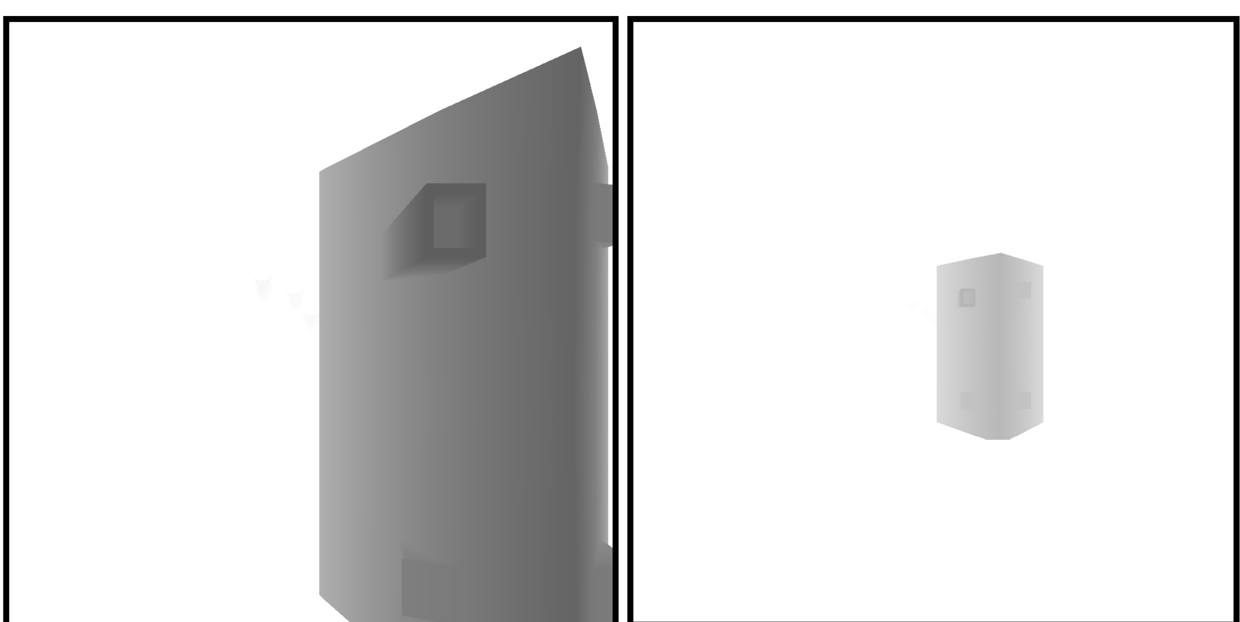


Figura 3.21 i 3.22, Podem veure que la figura de l'esquerra la càmera està molt aprop, i a la de la dreta, la càmera està molt allunyada (perquè es vegi la diferència s'ha capturat la imatge amb vista prespectiva). Ell fons dels dos és blanc ja que és el terra.

Veiem el codi:

```
GpuProgramParametersSharedPtr fragParams =
            mDepthTechnique->getPass(0)->getFragmentProgramParameters();
fragParams->setNamedConstant("diferencia", diferencia);
```

Podem observar que cridant mètodes de l'objecte Technique, mDetphTechnique, s'aconseguirà obtenir l'objecte GpuProgramParamatersParamatersSharedPtr, que és l'objecte que recollirà els paràmetres que necessiten els diferents vèrtexs i fragmentes shaders. Tot seguit se li passarà al fragment shader que s'ha creat (veure apartat x.x.x), el paràmetre diferència (que és el resultat de restar l'altura màxima amb la mínima). Cal dir que l'string diferencia ha de ser també definit amb el mateix nom de variable al fragment program, ja que per contra l'aplicació3D donaria un error.

3.2.1.4 Classe PramatresShader

Per passar els paràmetres que necessita el vèrtex shader creat (veure secció *3.2.2.2 Pluja en temps real*), el qual dóna el moviment adequat als manualsObjects perquè així simuli la caiguda de gotes, s'ha creat aquesta classe. Ja que per evitar que l'aplicació s'aturés de forma incorrecta, s'ha hagut de crear aquesta classe. Altrament, si per exemple es passen els paràmetres

dels shaders en el mètode createScene de MyApplication *(veure apartat anterior 3.2.1.1 Classe MyApplication)*, els objectes MaterialPtr i Technique, necessaris per a passar paràmetres, no es destrueixen de la forma correcta a l'hora de tancar l'aplicació d'Ogre3D. Per això s'ha creat aquesta classe.

Els paràmetres que es passaran en aquesta ocasió, seran els que s'hauran generat en el fitxer normalitzar.bin des de l'editor (*veure apartat 3.1.3.2 Calcular les posicions de pluja*), els quals representaran els extrems de la imatge capturada de l'escenari, és a dir, les coordenades x i y de l'extrem esquerra superior, les coordenades x i y de l'extrem dret de la imagte etc. Com es veurà en l'apartat 3.2.2.2 Pluja en temps real, s'utilitzaran per normalitzar (valors de 0 a 1)les posicions de les gotes en el món real, per així tenir una equivalència entre aquestes posicions del món real d'Ogre i la imatge generada de profunditat, ja que així sabrem exactament cada gota fins a quina altura ha de caure.

Variables:

- Ogre::MaterialPtr esbosMaterial: Guardarà el material que s'està utilitzant.
- Ogre::Technique* esbosTecnica: Guardarà la tècnica que s'està utilitzant.
- float * normalX: Guardarà dos punters a float , els quals seran els valors necessaris per normalitzar les coordenades 3D de l'escenari, així en un futur poder aplicar la imatge de profunditat ja que sabrem a quina x i z pertany el color de gris que podem obtenir del mapa de profunditats.
- float * normalY; Dos valors més per poder obtenir les posicions normalitzades de l'escenari 3D.

Aquesta classe únicament disposa del constructor el destructor i d'un mètode, el qual serà l'encarregat de passar les variables normalX i normalY com a paràmetres del vèrtex shader ja creat. Aquest és el mètode:

Mètodes:

- **paramatresShader(float *a, float *b):** S'Encarregarà de crear els punters a floats normalX i normalY i de cridar el mètode crearParamatres (que s'explicarà a continuació).
- **~paramatresShader():** Destruirà els punters creats.
- **void crearParamatres():** Aquest mètode s'encarregarà de passar els paràmetres normalX i normalY al vèrtex shader. Com s'ha vist ja en anteriors mètodes, primerament assignarem a l'objecte materialPtr el material que estem utilitzant. Tot seguit obtindrem la tècnica que utilitzarem per així poder obtenir el mètode GpuProgramParamatersSharedPtr que s'encarregarà de passar els paràmetres normalX i normalY que està esperant el vèrtex shader. Veiem el codi:

```
esbosMaterial = MaterialManager::getSingleton().getByName("esbos/esbos");
esbosMaterial->load(); // needs to be loaded manually
esbosTecnica = esbosMaterial->getBestTechnique();
GpuProgramParametersSharedPtr vertexParams =
esbosTecnica->getPass(0)->getVertexProgramParameters();
vertexParams->setNamedConstant("normalitzarX",
normalX,sizeof(normalX)/sizeof(normalX[0]));
vertexParams->setNamedConstant("normalitzarY",
normalY,sizeof(normalY)/sizeof(normalY[0]));
```

Com podem veure, per passar els paràmetres l'objecte que s'ha creat anomenat vertexParams es disposa del mètode setNamedConstant, el qual, primerament, se li ha de passar el nom de l'string que ha de ser el mateix que la variable que utilitzarà en el vèrtex shader, en aquest cas serà normalitzarX i normalitzarY (els quals veurem la seva funció en l'apartat *3.2.2.2 Pluja en temps real*). Tot seguit se li ha de passar la variable amb el valor assignat, en aquest cas són punters a floats, i finalment li direm la mida d'aquests punters.

3.2.2 Materials

Per a realitzar la simulació de pluja en temps real, s'han creat varis materials, a continuació els podrem veure:

3.2.2.1 Profunditat

Per crear un material, es necessita un arxiu **.material**, normalment un **.program** i els arxius relacionats que poden ser imatges o programes, com per exemple vèrtex i fragments shaders.

Aquí veiem com s'ha definit el **.material**:

```
material profunditat
{
	technique GLSL
	{
		pass
		{
			vertex_program_ref profunditatVP_GLSL
			{
			}

			fragment_program_ref profunditatFP_GLSL
			{
			}
		}
	}
```

Observem que es defineix que la tècnica utilitzada serà el llenguatge d'OpenGl GLSL. Tot seguit veiem que únicament defineix que s'utilitzaran un vèrtex i un fragment program de nom profunditatVP_GLSL i profunditatFP_GLSL respectivament als quals des del fitxer .material no se'ls hi passarà cap paràmetre.

Aquí veiem com s'ha definit el **.program** que serà l'encarregat de definir el nom del fitxer que guarda els shaders realitzats. Primer veiem el que fa referència al vèrtex shader:

```
vertex_program profunditatVP_GLSL glsl
{
	source profunditat.vert
}
```

Com es veu, el fitxer on guarda el vèrtex shader es diu profunditat.vert.

Aquest és el del fragment shader:

```
fragment_program profunditatFP_GLSL glsl
{
	source profunditat.frag
}
```

Ara podrem veure com s'ha fet el vèrtex i el fragment shader:

- **vèrtex shader**: Primerament es farà la transformació de la posició dels vèrtexs. S'agafarà la matriu, que és la concatenació de la matriu de projecció i la del model 3D que s'està veient. Si aquesta la multipliquem per la posició dels vèrtexs entrants, obtindrem la posició dels vèrtex que veu l'usuari. Si després mirem quina és la posició a l'eix de coordenades z, sabrem quina és la profunditat. Aquesta serà la variable de sortida del vèrtex shader. Finalment s'ha de fer la matriu de transformació, veiem el codi equivalent:

```
varying double depth;

void main()
{
	vec4 viewPos = gl_ModelViewMatrix * gl_Vertex;
	depth = -viewPos.z;

	gl_Position = ftransform();
}
```

Com veiem, la variable de sortida serà un double anomenat depth. Com veurem aquesta variable serà una variable d'entrada del fragment shader.

- Fragment shader: Si dividim el valor de la profunditat, que serà un paràmetre d'entrada, per un valor en concret que representarà el total de profunditat possible (*a l'apartat 3.2.2.2 Pluja en temps real*, es veurà que aquest valor també serà multiplicat per saber la posició en què la pluja s'hagi d'aturar per així no sobrepassar cap objecte de l'escena), obtindrem una escala de valors de

profunditats, si l'assignem a un color obtindrem l'escala de colors de profunditat. Si li assignem el mateix valor per cada component RGB del color, obtindrem una escala de grisos de la profunditat. Aquest n'és el codi:

```
uniform float diferencia;
varying double depth;

void main()
{
        gl_FragColor = vec4((double)(depth/diferencia));
}
```

Com veiem la variable diferència *(que s'ha vist com es passava com a paràmetre del fragment program en l'apartat 3.2.1.3 Classe CalculProfunditat)*, és del tipus uniform, és a dir que serà una constant, per contra la variable depth serà un valor d'entrada de tipus varying és a dir variable, ja que depèn de la posició de la càmera estarem més lluny o més a prop i per tant aquest valor podrà variar (Veure exemple de l'execució d'aquest material creat a la figura 3.19 i 3.20).

3.2.2.2 Pluja en temps real

El material que s'ha creat per aconseguir la simulació de pluja en temps real se l'ha anomenat "pluja". Aquest disposarà d'un **.material**, un **.program**, el mapa de profunditat que hem vist que es creava en l'apartat anterior, i un vèrtex shader que s'ocuparà de controlar tots el moviment de les gotes d'aigua. Veiem el **.material:**

```
material pluja/pluja
  {
   technique
     {
     pass
       {
          scene_blend alpha_blend
          depth_write off

       vertex_program_ref pluja/plujaVp
          {
        param_named_auto time         time_0_x 11.36363636
          }

       texture_unit texture
          {
   texture d.jpg 2d
   filtering none
      }
       }
     }
  }
```

Veiem que a la part de definir els paràmetres de pass es defineix el scene_blend com a alpha_blend i depth_write off. Aquests dos paràmetres s'han seleccionat ja que les gotes tindran un cert valor de transparència, si no els activéssim d'aquesta manera les gotes d'aigua no es mostraria tal i com s'ha explicat en l'apartat 2.2.2. Tot seguit es passarà un paràmetre al vèrtex shader. Concretament serà el temps, i s'ha definit d'aquesta manera ja que time_0_x permet fer un cicle determinat de temps, és a dir, en aquest cas, començarà a zero i anirà augmentant els segons fins arribar als 12.5 indicats, llavors es tornarà a inicialitzar a 0. S'ha fet així ja que, com es veurà a continuació, el moviment de les gotes depenen de la velocitat i del temps. Per tant, com en els vèrtexs shaders els vèrtexs utilitzats sempre són els mateixos, quan les gotes d'aigua ja hagin caigut fins a la profunditat màxima aquestes tornaran a pujar fins a la posició inicial per donar l'efecte de continuïtat. Si no es reiniciés el temps, aquestes sempre caurien fins a l'infinit. A la figura 3.21 es pot veure una imatge de la pluja en temps real creat, que servirà d'exemple per explicar el moviment que faran les gotes gràcies entre altres aspectes (que es veuran a continuació) al cicle del temps.

Figura 3.23, moviment de la pluja en temps real, la qual depèn del valor del temps per comportar-se de la forma correcte.

Un cop definit el cicle de temps que utilitzarà el vèrtex shader, es defineix un texture_unit, al qual veiem que se li passa el nom del fitxer que s'utilitzarà, finalment especifiquem que no es faci filtratge de la imatge (*explicat a l'apartat 2.2.2 Materials a l'Ogre*), per així augmentar la velocitat d'execució de l'aplicació. Aquesta imatge serà el mapa de profunditat, i per tant serà utilitzada per determinar fins a quina altura han de caure les gotes d'aigua.

Tot s'explicarà el **.program** creat:

```
vertex_program pluja/plujaVp cg
{
        source pluja.cg
        entry_point main_vp
        profiles vp40 arbvp1
...
```

Veiem que al final de la definició del .program, s'especifica que el vèrtex Shader que s'està definint està escrit amb el llenguatge cg. Ja a dins del .program, veiem que el nom de l'arxiu on es trobarà el shader utilitzat és el de "pluja.cg", i el nom de la funció del shader serà main_vp. Finalment es declara els profiles amb què s'han d'executar els programes (veiem que el profile del vèrtex shader és bastant modern, és degut a que per controlar la profunditat en què han de caure les gotes s'utilitza la funció tex2D, que veurem a continuació, que només serà acceptada per targetes gràfiques d'última generació).

Finalment, el .program, també passarà un paràmetre declarat com a param_named_auto i serà una matriu worldviewproj_matrix *(veure apartat 2.2.2 Materials a l'Ogre).*

Tot seguit es veurà la creació del vèrtex shader encarregat de simular el moviment de la caiguda de pluja en temps real. Aquí veiem els paràmetres que utilitzarà el vèrtex shader:

Paràmetres:

- **float4 pInitial : POSITION**= Paràmetre d'entrada, guardarà la posició del vèrtex (gota d'aigua).
- **out float2 uv : TEXCOORD0**= vector de dos floats, que guardarà la posició x i y normalitzada.
- **out float4 oPosition : POSITION**= vector de 4 floats, que guardarà la posició final de la gota. És un paràmetre de sortida, per tant serà d'entrada pel píxel shader.
- **uniform sampler2D texture : register(s0)**= És el paràmetre que es passa des del .material, concretament en la secció de texture_unit. És la imatge que guarda el mapa de profunditats. Com es pot veure, és de tipus sampler2D i l'anomenarem texture. El register(s0), és utilitzat per indicar que es tracta de la primera imatge definida en el .material (en aquest cas només hi tenim aquesta, tot i que és necessari declarar aquesta variable d'aquesta manera perquè pugui carregar de forma correcta la imatge desitjada, en aquest el mapa de profunditats).
- **uniform float time=** Paràmetre passat des del .material, representarà el temps.
- **uniform float4x4 modelViewProj**= Paràmetre passat des del .program. És una matriu de tipus worldviewproj_matrix, la qual s'ha explicat en la secció 2.2.2 Materials de l'Ogre.
- **uniform float4 normalitzarX**= Paràmetre que s'ha passat des de la classe MyFrameListener, creant l'objecte paramatresShaders *(veure apartat 3.2.1.2 Classe MyFrameListener i apartat 3.2.1.4 Classe ParamatresShader).* Guarda les coordenades x i y de l'extrem superior esquerra i la coordenada x de l'extrem superior dret.
- **uniform float4 normalitzarY**= Paràmetre que s'ha passat des de la classe MyFrameListener, creant l'objecte paramatresShaders *(veure apartat 3.2.1.2 Classe MyFrameListener i apartat 3.2.1.4 Classe ParamatresShader).* Guarda la coordenada y de l'extrem superior dret i les coordenades x i y de l'extrem inferior.

Implementació:

Al començament del shader es realitzaran les operacions matemàtiques, que són les més costoses en quan a temps (a part del moment en què s'accedirà a la variable texture). Per això, primerament s'obtindrà el valor de les posicions x i y normalitzades, és a dir buscarem el valor del vector de 2 floats declarat anomenat uv.

Si restem el valor de l'extrem esquerra de l'eix de les x (normalitzarX[0]), amb la coordenada x de la gota tractada en el shader, i ho dividim per l'amplada total de l'escenari (normalitzarX[0]-normalitzarX[2], obtindrem el valor de la coordenada x normalitzada, és a dir, obtindrem el valor equivalent de la posició actual en un interval de 0 i 1. Si fem el mateix però calculant l'altura, obtindrem la coordenada y normalitzada. Si tenim que normalitzarX guardarà els següents valors:

normalitzarX[0]	Coordenada x de l'extrem superior esquerra de l'escena.
normalitzarX[1]	Coordenada y de l'extrem superior esquerra de l'escena
normalitzarX[2]	Coordenada x de l'extrem superior dret de l'escena

I normalitzarY guardarà els següents:

normalitzarY[0]	Coordenada y de l'extrem superior dret de l'escena.
normalitzarY[1]	Coordenada x de l'extrem inferior esquerra de l'escena
normalitzarY[2]	Coordenada y de l'extrem inferior esquerra de l'escena

I aquí veiem el codi d'aquesta operació:

```
uv=float2(((normalitzarX[0]-pInitial.x)/(normalitzarX[0]-
normalitzarX[2])),((normalitzarX[1]-pInitial.z)/(normalitzarX[1]-
normalitzarY[2])));
```

Per comprovar si aquests valors normalitzats eren els correctes, s'ha agafat per exemple el valor normalitzat x, i dins del format de color RGB, s'ha assignat aquesta x únicament al valor R (vermell). Els valors de color G i B, s'han igualat a zero.

Per tant per veure millor les gotes, s'ha assignat que plogués en un rectangle que omplís tot l'escenari, i a més a més, s'ha assignat que estiguessin a la mateixa altura, per tant, podrem observar un recuadre que va baixant amb bloc, que si funcionés aniria de negra a vermell (i d'esquerra a dreta), veiem una imatge de la comprovació:

Figura 3.24, imatge capturada al fer proves sobre el bon funcionament de la normalització de posicions, com veiem a l'esquerra, tots els valors de color seran zero, i en mica en mica que es va arribant cap a la dreta de la imatge, es va veient com el color vermell és més intents.

Un cop sabem que el procés de normalitzar les posicions a uns intervals entre 0 i 1 funciona correctament, podrem accedir a la seva posició equivalent a la imatge de profunditats, per així obtenir el seu valor de gris, i després al multiplicar-ho per la profunditat màxima (recordem que en l'apartat *3.2.2.1 Profunditat*, hem vist que es dividia la profunditat (Depth), per la profunditat màxima) sabrem fins a quina altura ha de caure aquella gota. Per tant:

```
float profunditat=(tex2D(texture,uv).x-1)*(-2500);
```

Veiem que s'utilitza la funció tex2D, aquesta funció, serveix per obtenir el color d'un píxel d'una imatge (texture), donat un vector de dos floats que representaran la coordenada x i y (uv) del píxel desitjat.

Per a aconseguir el màxim realisme de la caiguda de les gotes d'aigua de la pluja, s'ha observat prèviament el comportament físic d'aquestes, i hem pogut observar, que pràcticament durant tota la caiguda de les gotes d'aigua baixa a una velocitat constant a causa de la velocitat terminal que arriba a aconseguir *(veure apartat 2.3 Física de la pluja)*, per tant descriurem la caiguda de les gotes d'aigua com una velocitat de caiguda constant, és a dir únicament es multiplicarà la velocitat per el temps, ja que el valor de l'acceleració serà igual a zero.

És a dir partint de la fórmula:

$$Y=Y_0-V_0*t-1/2*a*t$$

Com ja s'ha dit l'acceleració és igual a zero. Per tant la fórmula que descriurà la caiguda de les gotes d'aigua serà aquesta:

```
pInitial.y= pInitial.y - 220 * time ;
```

Un cop les gotes ja estan fent el moviment desitjat, s'hauran de comprovar dos aspectes:

- **Impacte d'una gota amb la superfície**. Quan una gota hagi fet tot el recorregut (sense trobar-se cap obstacle) fins arribar al terra, s'haurà de fer pujar novament fins a l'altura màxima, ja que per contra, a més de que les gotes travessarien el terra, cosa que faria la simulació de pluja molt irreal, es veuria un efecte de caiguda de pluja en bloc molt semblant al de la figura 3.19, ja que les gotes no es renovarien correctament. S'ha fet el següent condicional:

```
if (pInitial.y<0)
        pInitial.y+=2500;
```

  Com es veu, es comprova si la posició de la gota és inferior a la del terra, si és així es torna a pujar a l'altura màxima, en aquest cas 2500. Podem veure-ho a la figura 3.25.

- **Impacte d'una gota amb un obstacle**. Si durant la caiguda d'una gota es troba amb un obstacle (veient l'escenari creat podríem parlar per exemple de l'impacte amb la teulada de la casa), aquesta es col·locarà en un punt indefinit del món tridimensional. Veiem el condicional:

```
if(pInitial.y<profunditat)
        pInitial.y= float(-2.0);
```

  Com veiem se li assigna una posició desconeguda, en aquest cas -2.0. Veiem una imatge del funcionament de l'impacte de les gotes amb la teulada a la figura 3.26.

Aquesta fletxa representa la línia equivalent a la superfície.

Figura 3.25, imatge captada durant l'execució de l'aplicació que mostra que les gotes d'aigua realment no travessen la superfície. En aquesta ocasió s'ha escollit un color de gota més blanc i menys transparent del normal, únicament per poder observar més fàcilment les gotes en la imatge.

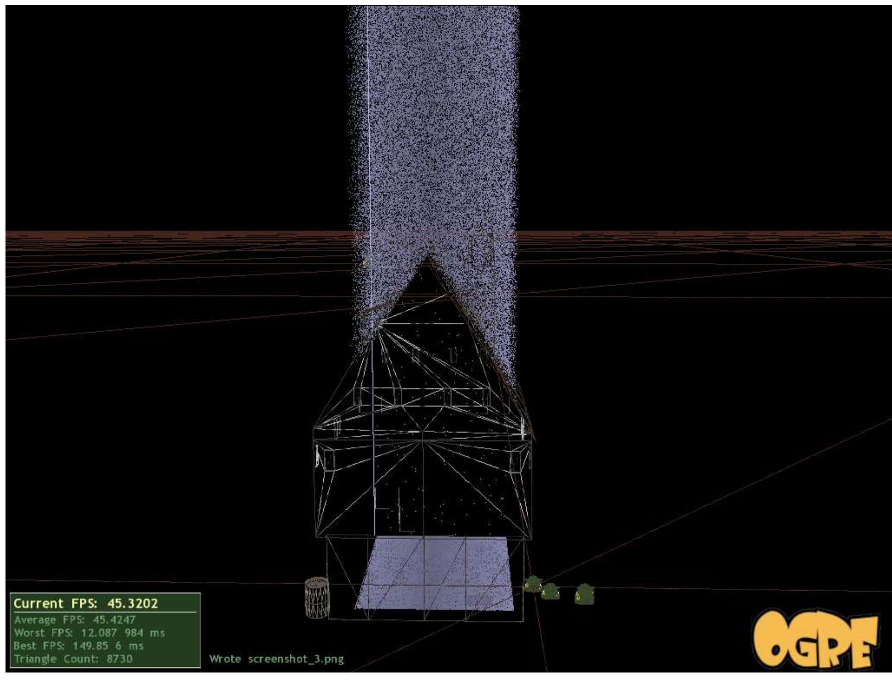


Figura 3.26, Imatge capturada durant l'aplicació (s'observa l'escenari des d'una posició distanciada). S'ha deshabitat mostrar la superfície de la casa per poder observar si les gotes queien o no a dins de la casa. Com podem veure no hi cauen.

Finalment, perquè els vèrtexs es col·loquin a la posició desitjada, s'haurà de multiplicar la posició d'aquests per la matriu modelViewProj.

```
oPosition = mul(modelViewProj, pInitial);
```

Un cop acabat el comportament de les gotes d'aigua durant la simulació, només faltarà posar el color adequat a aquests vèrtexs. Serà un color suau, tirant cap a blanc, però amb un toc de blau. Tindran el component de transparència bastant elevat. Amb el píxel shader, s'ha assignat aquest color a les gotes d'aigua, finalment s'ha assignat aquest color:

```
color = float4(0.7,0.7,1,0.3);
```

Com veiem, s'ha assignat al color vermell i verd el valor de 0.7 (sobre un total de 1), i a la component blava un 1. El quart component és alpha, la component que determina la transparència, se li ha assignat 0.3 (0 seria la màxima transparència). Podem veure una imatge de l'aplicació en funcionament a la figura 3.27.

Figura 3.27, imatge generada durant l'execució de l'aplicació de la simulació de pluja en temps real.

3.3 Visualització de pluja realista

Com ja s'ha explicat a la introducció, inicialment una part substancial del projecte s'ha implementat a la CPU, i no a la GPU com és l'objectiu final. El motiu és poder fer debug més fàcilment i poder comprovar amb més rapidesa si el resultat final que s'obtindria seria el desitjat o no, abans de passar-ho tot a la GPU. Això es deu a que hi ha alguns aspectes que a la GPU es compliquen una mica més que fent-ho directament a la CPU, com per exemple el procés de debug, a causa de les gairebé inexistents eines per a la GPU.

Un cop ja s'han desenvolupat tant l'edició de pluja com el comportament d'aquesta, i funcionen correctament, ens podem centrar en tractar la visualització de la pluja de forma que sembli el més realista possible. Per tant a continuació s'explicaran detalladament tots els passos que s'han implementat fins a arribar al resultat final. Per començar s'explicarà tots els aspectes que s'hagin hagut de tractar des del vèrtex shader i finalment s'explicarà el píxel shader.

3.3.1 Transformació dels punts a billboards

Fins ara les gotes d'aigua eren simples vèrtexs. Com sabem, les gotes d'aigua tot i que si les mires des d'una distància considerable poden arribar a semblar, a causa de la persistència de les imatges a la retina, les veiem d'una forma més allargada. Per tant, donat que el nostre sistema treballa amb un framerate constant amb el que cada fotograma representa un cert temps d'apertura de l'obturador d'una càmara virtual, i per guanyar realisme, el primer que s'ha fet ha estat canviar els vèrtexs per billboards que ens mostrin aquest estirament de la imatge. Cada billboard ens representarà una gota.

Per fer-ho primerament cal disposar d'un objecte que encaselli tots els billboards. Ogre ja disposa d'un que realitza aquesta tasca, l'objecte billboardSet, per tant a la classe MyFrameListener s'hi incorpora un nou atribut anomenat bbs el qual és un billboardSet.

```
BillboardSet* bbs;
```

Quan es vulgui crear la pluja s'instanciarà el billboardSet, ja que començarem a crear a partir d'ell tots els billboards per les gotes. Veiem com creem el billboarSet:

```
//creem l'objecte que encapsularà tots els billboards,reservem l'espai
bbs=mSceneMgr->createBillboardSet("conjuntGotes",(int)(2*(tamanyTot/10)));
```

Un cop tenim creat l'objecte que encasellarà tots els billboards, a partir d'aquest objecte crearem tots els billboards que necessitarem, és a dir, tants com gotes haguem de crear. El codi és pràcticament igual a l'explicat en el capítol 3.2.3.2 Classe MyFrameListener. Només canviaria en el moment de crear les gotes

d'aigua, ja que ara seran billboards i abans eren manualObjects en forma de vèrtexs. A continuació es pot veure aquesta part del codi:

```
        //llegint del fitxer especifico les posicions que tindran les gotes
        (billboards),farem un recorregut per tot el fitxer de posicions de gotes.
        while (tamanyFinsAPunter<tamanyTot){
              fread(&posicioPluja[0],sizeof(float),2,FitxerDeLectura);
              //creem un nou billboard, i li especifiquem la posició que li pertoca, i
              amés li calculem un nou punt aleatori entre el cel i la terra. Sera una
              nova gota.
              Billboard* bba = bbs-
        >createBillboard(posicioPluja[0],MINIM + rand() % (MAXIM-
              MINIM),posicioPluja[1],mMinLightColourb);
              tamanyFinsAPunter=ftell(FitxerDeLectura);//bytes
  quee portem llegits.
         }
        mSceneMgr->getRootSceneNode()->createChildSceneNode(
              Vector3(0.0, 0.0, 0.0))->attachObject(bbs);
        …
```

Fins aquí hem creat billboards que representaran les gotes d'aigua, però, com hem vist en detall a l'apartat de fonaments previs, un billboard és un rectangle que sempre és perpendicular a la direcció que uneix el centre i l'observador. Per tant, es rota segons la posició que tingui respecte de la càmera. Això serà molt útil, però faltarà refinar alguna aspecte més, ja que ens interessa que el billboard vagi rotant depenent de la posició de la càmera, però ens interessarà que sempre estigui vertical, és a dir, no ens interessa que quan mirem una gota d'aigua des de sobre d'ella, aquesta roti, ja que llavors veuríem gotes irreals, observaríem gotes molt amples que es mouen en sentit horitzontal.

Per tant cal especificar quin tipus de billboard és: en aquest cas serà un billboard que només rotarà respecte un eix, el vertical.

A la figura 3.28 es pot veure el sistema de coordenades d'Ogre.

Si recordem una gota d'aigua i observem el sistema de coordenades, veurem que cal fixar l'eix Y.

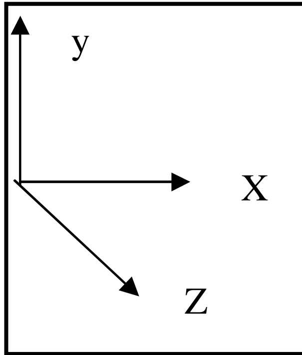


Figura 3.28. sistema de coordenades

Veiem el codi que ens permet fixar un eix als billboards.

Li diem el tipus de billboard que ha de ser, com seran gotes d'aigua han de tenir sempre la mateixa orientació.

```
bbs->setBillboardType(BBT_ORIENTED_COMMON);
//fixem respecte l'eix que únicament ha de rotar la gota, és l'eix y (el
vertical), ja que si ens el mirem des del cel no podem veure una línia
horitzontal (no ha de rotar), sinó que un pun o quelcom semblantt, i quan el
veiem de cara si que hem de veure una línia vertical que representaria la
gota d'aigua.
bbs->setCommonDirection(Vector3::UNIT_Y);
```

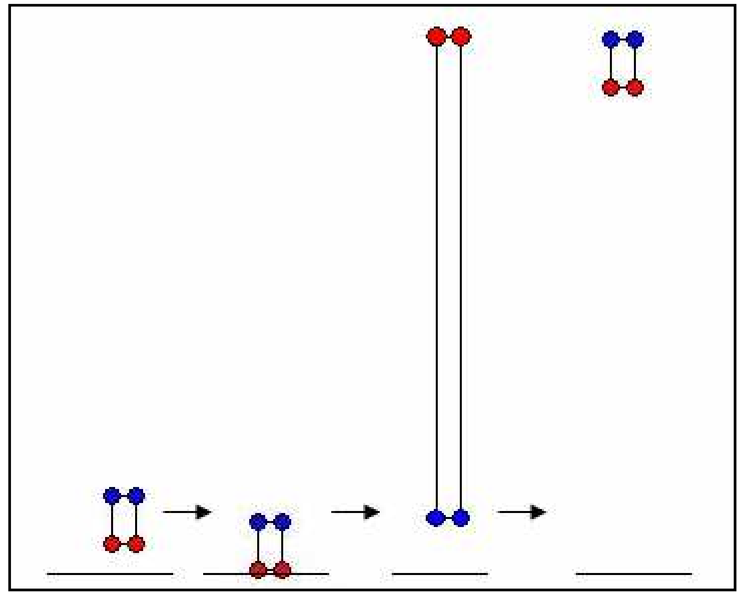
A continuació podem observar una imatge capturada en temps d'execució de l'aplicació en funcionament, en aquest cas és una vista des de dalt al cel, la imatge mostra la caiguda dels billboards sense fixar l'eix y, com podem observar pensant que és una vista des de dalt dels billbaords aquests es veuen horitzontals. Per tant cal fixar l'eix y per fer la rotació només sobre aquest eix, per així aconseguir que els billboards tinguin un sentit vertical, per simular gotes d'aigua.

Figura 3.29, en aquesta imatge, podem observar diferents circumferències de color vermell que marquen que els billboards no tenen la forma desitjada

3.3.1.1 Tractament d'un billboard d'Ogre des de la GPU

Tractar els vèrtexs dels billboards generats per Ogre des de la GPU té algun inconvenient, ja que, al voler fer el mateix que fèiem anteriorment amb els vèrtexs, la implementació ens genera problemes nous. El problema està amb el tractament que fem de les gotes al intersecar amb un objecte o al terra. Fins

ara, quan arribàvem al terra pujàvem la gota fins a la part superior del volum de pluja (el cel). Ara es complica una mica, ja que si ho deixem així, primerament els 2 vèrtexs inferiors del billboard passaran per sota del terra, i per tant, augmentaran la seva posició fins la part superior del volum (2500 unitats al nostre codi). Això vol dir que desplaçarem els vèrtexs inferiors cap al cel, i els altres dos vèrtexs superiors encara estaran a baix, apunt d'impactar amb el terra. Com aquests 4 vèrtexs estan units per un quadrilàter, es veurà un rectangle des de pràcticament el terra fins al cel, és a dir, des dels vèrtexs que ja han pujat al intersecar amb el terra, fins els vèrtexs que encara no han intersecat. Finalment quan els vèrtexs del billboard arribin al terra, aquests també pujaran cap a dalt i es tornarà a veure un billboard normal. Aquest procés produiria un efecte irreal, ja que aniríem veien com cau la pluja però aniríem veient com apareixen línies i rectangles a la pantalla. Aquesta problemàtica que s'acaba d'explicar es pot veure il·lustrat a la figura 3.30.



Figrua 3.30, problemàtica dels billboards al impactar amb el terra o un obstacle: com veiem, quan els vèrtexs inferiors (color vermell) del billboard impacten amb el terra, aquests pugen cap a dalt, mentres que els vèrtexs superiors (color blau) es mantenen a baix crean així una línia molt llarga i molesta. Finalment quan aquests últims acaben impactant a terra també pugen cap a dalt i el billboard queda en la seva forma correcte.

Al principi, com ja hem explicat es va traslladar el codi respecte al moviment de les gotes d'aigua de la GPU a la CPU, en aquí no hi havia aquest problema, ja que només controlàvem la posició del billboard en general, quan aquest arribava a la posició 0, és a dir al terra, assignàvem la nova posició al billboard. Com tractàvem tot el billboard, no teníem el problema que hem trobat a la GPU.

Per tant, com hem vist, la part clau del codi del vèrtex shader és la que controla si la gota interseca amb el terra o un objecte, és a dir aquesta part del codi:

```
//Si la posició de la gota està per sota del terra la pugem
if (pInitial.y<0)
        pInitial.y+=2500;
//si la posició està per sota d'un objecte la posem a un lloc que no es vegi
if(pInitial.y<profunditat)
      pInitial.y= float(-2.0);
```

Es podria solucionar si sapiguéssim, en cada moment, quin vèrtex és dels quatre que hi ha en un Billboard, o més ven dit, necessitem saber si és un vèrtex de la part superior d'un Billboard o un de la part inferior.

Sabent això, per solucionar el problema explicat, als dos vèrtexs superiors a l'hora de comprovar si han travessat el terra o un objecte, se li hauria de restar l'alçada del Billboard. Així aconseguiríem que els quatre vèrtexs tinguessin la mateixa coordenada "y", que és el que ens interessa, ja que així quan els vèrtexs inferiors impactéssin amb el terra els dos vèrtexs superiors també tindrien la mateixa coordenada que els inferiors, i per tant els quatre vèrtexs es comportarien com un de sol. Pujarien tots a la vegada i no es produiria la línia.

Per realitzar això cal passar-li a cada vèrtex la informació sobre si és un vèrtex superior o inferior, però això no es pot fer només amb vèrtex shaders. Es pot fer d'una manera indirecta, assignant un determinat color als vèrtexs superiors i un altre de diferent als inferiors. Així, des del vèrtex shader, podrem consultar de quin color és aquell vèrtex i per tant sabrem si és l'inferior o el superior. A continuació es pot veure el codi del vèrtex shader que implementa el que s'acaba de descriure.

```
//inicialment superior val zero, si finalment és zero voldrà dir que és un
vertex inferior
int superior=0;
//l'atribut c indica el color del vèrtex que s'està tractant, si el valor de  la
component vermella és superior a 0.5 voldrà dir que és un vertex de la part
superior del billboard, i per tant la variable superior valdrà 1, altrament
continuarà valent 0
if (c[0]>0.5)
{
      superior=1;
}
//actualitzem la posició del vèrtexs.
pInitial.y= pInitial.y - 0.5 * time ;
//comprovem si la gota toca al terra o a un objecte, per tant si és un vertex
de la part superior, realitzarem la resta, ja que la variable serà igual a 1, en
el cas del codi, la nova posició del vèrtexs serà la resultant de restar la
posició actual, menys l'alçada del billboard, en aquest cas 14.
if (pInitial.y-(14*superior)<0)
      pInitial.y+=2500;
if (pInitial.y-(superior*14)<profunditat)
      pInitial.y= float(-2.0);
```

Fins aquí s'ha explicat com s'haurà de fer al vèrtex shader. Ara falta assignar a cada vèrtex un color depenent de la posició d'aquest al billboard.

La classe Billboard disposa de la funció:

```
Void setColour (const ColourValue &colour)
```

Però aquesta assigna el color passat per paràmetre a tot el billboard, i no es pot especificar el color que desitges que tingui cada vèrtexs. El que s'ha fet és modificar la classe Billboard i la billboardSet, ja que aquesta és la que realitza el treball de renderitzar cada billboard.

Primerament a la classe Billboard s'han afegit quatre atributs nous, un per cada vèrtex. Cada atribut guardarà el valor del color del vèrtexs en qüestió. Veiem la definició realitzada:

```
//colors dels 4 vertexs del billboard
ColourValue mColourLeftTop;
ColourValue mColourRightTop;
ColourValue mColourLeftBottom;
ColourValue mColourRightBottom;
```

Com veiem els quatre atributs són del tipus ColurValue.

S'ha implementat dues funcions, una per assignar els quatre vèrtexs del billboard, i una altra per obtenir-ne el seu color:

```
//Definim els colors dels 4 vertexs del billboard començant per dalt
d'esquerra a dreta i acabant per baix a la dreta
void setColourVertices(const ColourValue& coloura,const
ColourValue& colourb,const ColourValue& colourc,const
ColourValue& colourd);

//Recuperem el colour de cada vertex del billboard
void getColourVertices(ColourValue& coloura,ColourValue&
colourb,ColourValue& colourc,ColourValue& colourd);
```

BillboardSet disposa de la funció injectBillboard:

```
void BillboardSet::injectBillboard(Billboard& bb)
```

La qual és l'encarregada de definir el billboard que se li passa per paràmetre. Dintre d'aquesta funció es generen els quatre vèrtexs del Billboard. Ho fa a través de la funció genVertices:

```
void BillboardSet::genVertices(const Vector3* const
offsets,Billboard& bb)
```

Dintre d'aquesta funció, es defineixen les posicions que tindrà cada vèrtexs i el seu color. La part corresponent a la posició no cal modificar-la, només ens caldrà modificar la part en la qual tracta el color de cada vèrtexs, ja que inicialment posava el mateix color a cada vèrtexs.

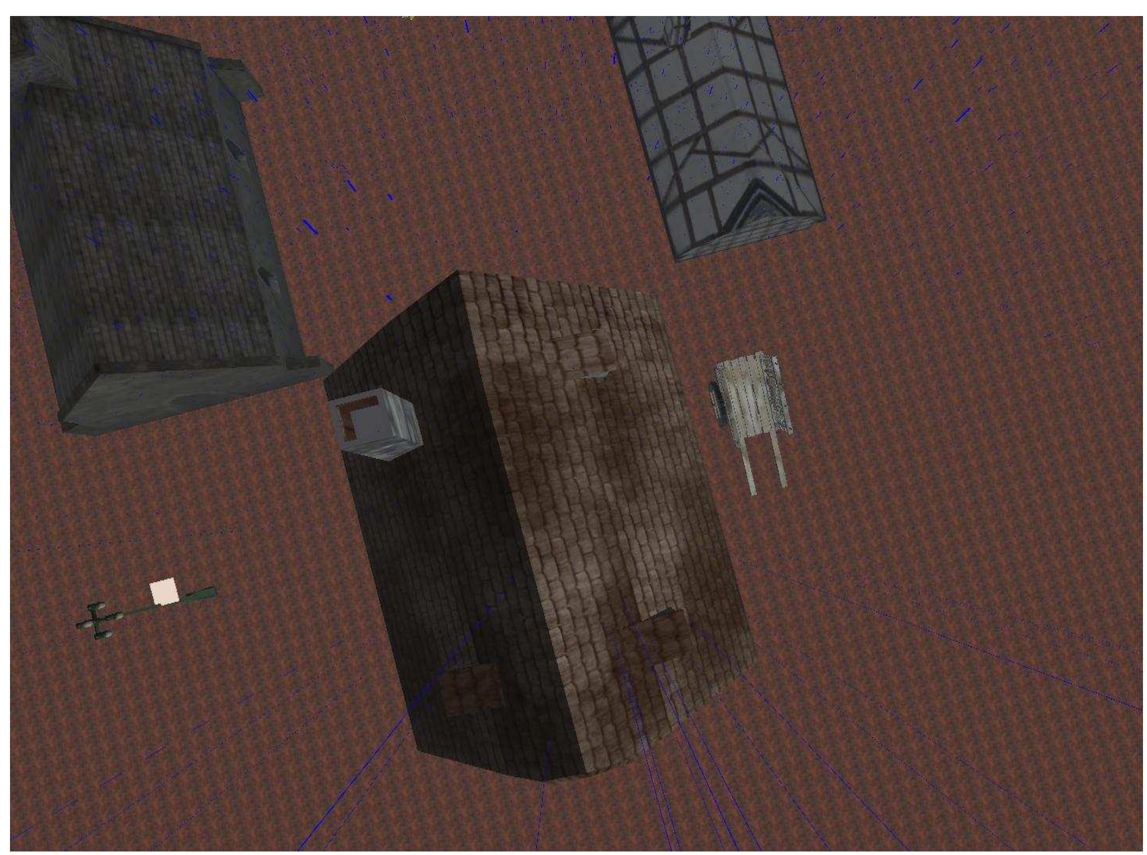
Per cada vèrtex realitza la mateixa tasca. A Ogre hi ha diferents tipus formats de colors, per això hi ha una funció que permet convertir el format de color passat per paràmetre a un altre format que ens permetrà unificar els diferents formats que hi pugui haver de colors, el converteix a uint32*. Feta aquesta conversió ja podrem assignar el color. Veiem el codi:

```
RGBA colourd;
bb.mColourRightBottom.r=0.0;
Root::getSingleton().convertColourValue(bb.mColourRightBottom,
&colourd);
// Convert float* to RGBA*
 pCol = static_cast<RGBA*>(static_cast<void*>(mLockPtr));
*pCol++ = colourd;
```

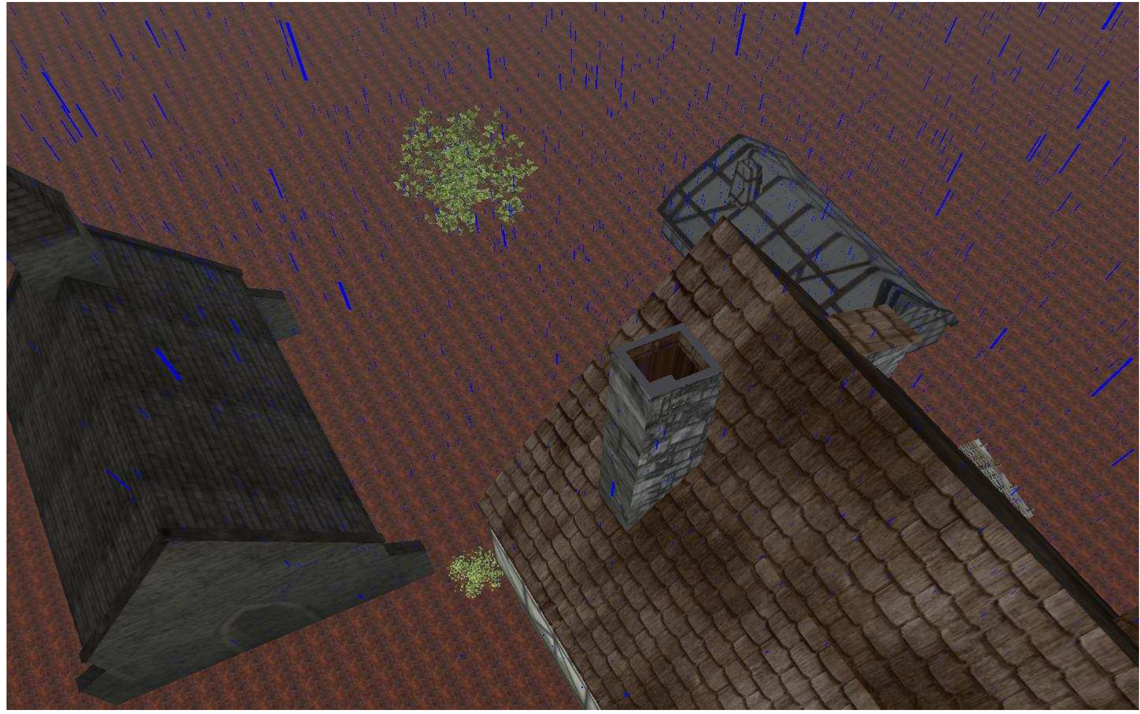
En el codi que s'acaba de mostrar "bb" és el billboard en qüestió i com podem veure a la component red del seu atribut mColourRightBottom, se li assigna un valor de zero. Com veiem el valor d'aquest és inferior a 0.5, que és el que s'havia planejat, els vèrtexs superiors tindran uns valors majors que 0.5 i els inferiors un valor més petit que 0.5. Per acabar de concretar i donar més informació al vèrtex shader, es distingirà de si és un vèrtex superior o inferior, però a més a més si és el vèrtex de la dreta o de l'esquerra, ja que en futurs capítols es necessitarà saber exactament quin vèrtex és, així doncs s'assignarà els següents valors a cada vèrtex del billboard:

- Vèrtex superior esquerra: 0.8.
- Vèrtex superior dret: 0.6.
- Vèrtex inferior esquerra: 0.3.
- Vèrtex inferior dret: 0.

D'aquesta manera les gotes d'aigua, ara en forma de billboards, realitzen tot el cicle d'aigua de forma correcte.



Figrua 3.30, captura de l'aplicació en funcionament. En aquest cas els billboards són de color blau. Es poden observar algunes línies molt allergades, a conseqüencia de la problemàtica explicada. Aquesta imatge mostra el mateix que la 3.29



Figrua 3.31, captura de pantalla amb el problema dels billboards solucionat. Com podem veure les línies blaves de la figura 3.30 aplicant el codi explicat desapareixen.

3.3.2 Càlcul de les textures

En el següent apartat s'explicarà els diferents passos que s'han hagut de desenvolupar per aplicar a cada gota d'aigua la imatge que li pertoca. Primerament explicarem els càlculs necessaris per obtenir els angles que formen la posició dels punts de llum i de l'observador respecte cada billboard, després veurem com calcular el centre del billboard des del vèrtex shader, a continuació assignarem les coordenades de les textures seleccionades, després redimencionarem els billboards segons els valors dels angles obtinguts i finalment, un cop tinguem fet tot el procés que s'acaba de descriure, s'explicarà com fer tot aquest procés amb tants punts de llum com hi hagin a l'escena.

3.3.2.1 Càlcul dels angles

Com s'ha observat a l'apartat de fonaments previs, Secció 2.5, amb l'explicació de l'article en que s'ha basat el projecte, per a determinar la textura que posarem a la gota d'aigua s'haurà de calcular els angles que formen les gotes d'aigua respecte:

- L'alçada de la càmera
- L'alçada d'un punt de llum
- La posició del punt de llum (sense tenir en compte aquest cop l'alçada d'aquest).

A la figura 3.32 veiem el sistema de coordenades que es fa servir, on la gota d'aigua és el centre, amb l'eix de coordenades "y" amb sentit oposat al de la caiguda de la gota i l'eix "x" el qual defineix la projecció de la càmera. Els angles que es poden veure al sistema de coordenades, l'angle "theta" i "phi" són els angles d'elevació i azimut respectivament.

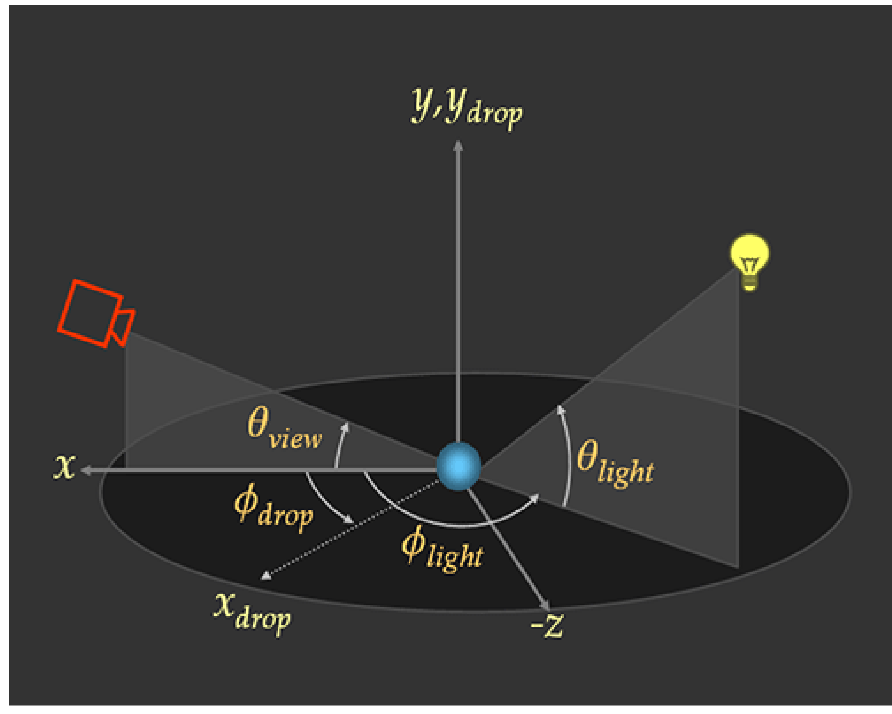


Figura 3.32, sistema de coordenades utilitzat per la base de dades d'imatges del projecte.

Per calcular aquests tres angles utilitzarem en els tres casos el producte escalar:

$$\vec{A} \cdot \vec{B} = |\vec{A}||\vec{B}|cos\theta$$

Si aïllem l'angle podrem obtenir el valor dels angles ja esmentats, podem veure-ho millor en la figura 3.34.

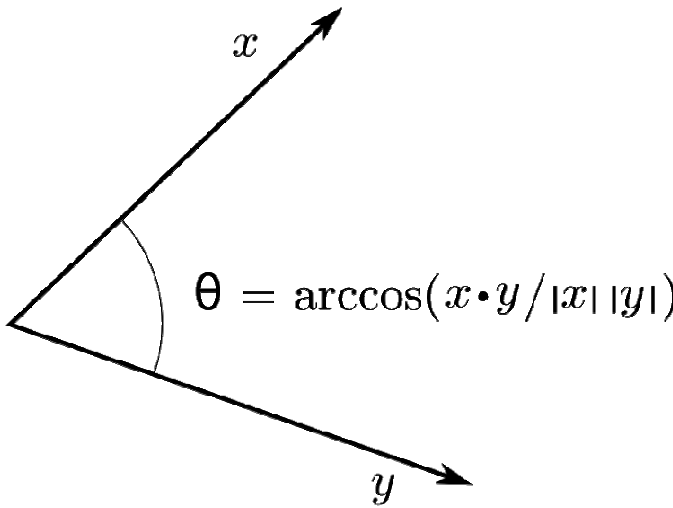


Figura 3.34, obtenció de l'angle a partir del producte escalar

Angle θview: Per calcular l'angle θview necessitem definir els dos vectors que formaran aquest angle, el primer vector és el que va des del centre de la gota d'aigua fins la posició de la càmera. El segon vector és el que va des del centre de la gota d'aigua fins la posició de la càmera sense tenir en compte la coordenada y de la càmera. Un cop disposem d'aquests dos vectors només s'ha d'aplicar la fórmula de la figura 3.33 i obtindrem l'angle. Veiem el codi del vèrtex shader que realitza aquestes operacions explicades:

```
//"camera" guarda la posició de la càmera, i gota la posició del vertex que s'està tractant.
float3 CamPos=camera.xyz-gota;
//un cop calculat el vector CamPos, el segon vector serà "V", que és el mateix que CamPos però amb la coordenada y=0.
float3 V = CamPos;
V.y=0;
//calculem el modul de V i el de CamPos
float mV = sqrt(dot(V,V));
float mCamPos = sqrt(dot(CamPos,CamPos));
//obtenim l'angle
float aCamera = abs(acos(dot(CamPos,V)/(mV*mCamPos)));
```

Angle θlight: En aquest cas, primer de tot calcularem el valor del vector entre la posició de la gota d'aigua i la del punt de llum, l'altre vector que necessitem és aquest mateix vector però amb el valor de y=0. Veiem el codi:

```
//primer vector
lightPos[numLlums].xyz;
//segon vector
float3 L=lightPos[numLlums].xyz;
L.y=0;
//calculem moduls
float mL = sqrt(dot(L,L));
float mlightPosition =
sqrt(dot(lightPos[numLlums],lightPos[numLlums]));
//calculem angle
float alLlum =
acos(dot(lightPos[numLlums].xyz,L)/(mlightPosition*mL));
```

Angle Φlight: Com ja hem vist a la figura 3.33, aquest angle es forma entre el vector que va des del centre de la gota d'aigua fins a la posició de la càmera (amb alçada=0) amb el vector que va des del centre de la gota d'aigua fins al punt de llum (també amb alçada=0). Per tant, observant els dos fragments anteriors de codis, podem aprofitar dos variables els quals defineixen aquests vectors que necessitem, aquestes variables són la "L" la qual és la posició de la llum sense tenir en compte l'alçada, i la "V" que tampoc té en compte l'alçada. Coneixent aquests dos valors i també els seus mòduls: mL i mV respectivament només ens cal aplicar la fórmula:

```
//Calculem angle
float phi=acos(dot(V,L)/(mV*mL));
```

Cal dir que tots aquests resultats són amb radiants, com la base de dades d'imatges de gotes d'aigua estan ordenades amb graus i no amb radiants, ens cal passar aquests resultats de radiants a graus:

```
phi=(phi*180)/3.14159265358979;
alLlum=(alLlum*180)/3.14159265358979;
aCamera=(aCamera*180)/3.14159265358979;
```

3.3.2.2 Càlcul del centre del billboard des del vertex shader

Fins aquí s'ha vist com s'han calculat els angles, però si es vol un resultat realista cal tenir en compte un aspecte: Si repassem el codi anterior, observarem com als diferents càlculs que s'han realitzat sempre s'agafa com a centre del billboard la posició del vèrtex que s'està tractant, que no és correcte ja que la posició d'aquests vèrtexs sempre és la cantonada del billboard.

Tot i que els billboards tenen dimensions reduïdes i la distancia entre un vèrtex del billboard i el centre d'aquest no és molt gran, si aquest inconvenient no es soluciona podríem obtenir un resultat no desitjat, ja que quan estiguéssim veient la pluja com cau, veuríem formes de gotes estranyes i/o irreals: Es podria produir el cas de que quan observéssim una gota d'aigua, com el resultat de calcular l'angle d'una cantonada del billboard amb una altra cantonada d'aquest pot ser diferent, probablement es donaria el cas que una

part del billboard mostrés una part d'una imatge i que l'altre part del billboard en mostrés una altre de diferent, cosa que faria que la similitud amb una gota d'aigua s'esvaís. El cas extrem seria que en un moment donat un billboard intentés mostrar 4 imatges diferents.

Com ja s'ha fet en el capítol anterior per determinar si un vèrtex era de la part superior o inferior del billboard, ens aprofitarem del color de cada vèrtex per donar informació de la distancia que hi ha entre aquest i el centre. Aquesta distància es calcularà, com en l'anterior capítol, a la funció genVertices de la classe billboardSet.

En l'anterior capítol hem utilitzat un canal del color del vèrtex (el vermell) per definir la posició de cada vèrtex al billboard, llavors agafarem els altres tres (verd, blau i alfa) per dipositar la informació que necessitem: la distància que hi ha entre el centre i el vèrtex. Per tant el que es farà serà calcular la distància entre el centre i cada cantonada del billboard i guardar aquest resultat a les tres components del color que ens queden lliures, un cop estiguem al vèrtex shader, per obtenir el centre del billboard sumarem aquesta distància a la posició del vèrtex shader.

Per tant a la component verd (que anomenem g, de "green") del color li assignarem la distancia de la coordenada x, a la blava (b) la distància que hi hagi en l'eix y, i a la component a (d'alpha) la distància en l'eix z. Per assegurar-nos que aquesta distància està compresa entre 0 i 1 el resultat es divideix entre 10, ja que com coneixem les dimensions del billboard, sabem que com a molt tindran valors d'entre 1 i 9, si aquests valors resultants els dividim entre 10 ens assegurem obtenir un valor entre 0 i 1. No es divideix per un valor superior a 10 per no perdre precisió, ja que al tractar amb decimals es perd exactitud. Veiem el codi:

```
//Calculo la component vermella a partir de la funció calcularRed
bb.mColourLeftTop.r=0.8;
calcularRed(bb.mColourLeftTop.r,Cx-(pt.x + bb.mPosition.x),Cz-
(pt.z + bb.mPosition.z));
//pt.x+bb.mPosition.x representa el centre del billboard per el que fa a la
component x.
bb.mColourLeftTop=ColourValue(bb.mColourLeftTop.r,abs((Cx-
(pt.x + bb.mPosition.x)))/10.0,abs((Cy-(pt.y +
bb.mPosition.y)))/10.0,abs((Cz-(pt.z +
bb.mPosition.z)))/10.0);
```

En aquest codi, pt.x+bb.mPosition.x representa el centre del billboard pel que fa a la component x.

Ja des del vèrtex shader s'ha de sumar la distància guardada a les diferents components del color, a la posició del vèrtex:

```
float3 center=pInitial.xyz;
//Calculo el centre del billboard, que és C=V+D, on V el coneixem, ja que és el punt del VS, i D esta guardat per els parametres del color d'entrada c[1]=x, c[2]=y i c[3]=z. D=distancia entre centre i vertex calculat a la CPU, genvertices. Ja que al billboardSet, es divideix per 10, ja que poden sortir expresions de lestil de 9.01..i al passar-ho al vertex el color es entre 0 i 1
center.x=c[1]*10.0+pInitial.x;
center.y=c[2]*10.0+pInitial.y;
center.z=c[3]*10.0+pInitial.z;
```

Com es pot veure en el fragment de codi, s'utilitza la funció calcularRed, aquesta és necessària per a poder obtenir un resultat correcte. Sense aquesta funció pot semblar que el resultat de calcular el centre del billboard seria el correcte, però si ens fixem en el moment d'assignar el color al vèrtex del billboard:

```
bb.mColourLeftTop=ColourValue(bb.mColourLeftTop.r,abs((Cx-
(pt.x + bb.mPosition.x)))/10.0,abs((Cy-(pt.y +
bb.mPosition.y)))/10.0,abs((Cz-(pt.z +
bb.mPosition.z)))/10.0);
```

Observem que a l'hora de calcular la diferència entre un vèrtex i el centre del billboard, s'agafa el resultat amb valor absolut, per tant deixem de saber si la diferència entre un vèrtex del billboard i el centre d'aquest és negativa o positiva. Això s'ha de solucionar, ja que sinó en comptes d'apropar-nos a la posició del centre del billboard, en alguns casos, ens allunyaríem d'aquest.

Així doncs la funció calcularRed, ens a serveix per donar informació sobre la distància entre el centre del billboard i els diferents vèrtexs d'aquest. Concretament donarà informació de si les distàncies en els eixos x i z, són positives o negatives. Aquesta informació s'acumularà al valor que ja tenia la component vermella, per tant la component vermella de cada vèrtex del billboard ens donarà la següent informació:

Segons el valor ens indicarà quina cantonada del billboard representa.
Depenent del valor que s'hi hagi acumulat, ens informarà de si les distàncies especificades en les altres 3 components del color, han de ser tractades com a positives o negatives, ja que l'atribut que defineix el color en els vèrtex i píxel shaders sempre tenen els quatre valors (rgba) positius.

Aquest és el codi:

```
void calcularRed(float & valIni,float x,float z)
{
      if (x<0 && z>=0)
            valIni=valIni+0.05;
      else if (x>=0 && z<0)
            valIni=valIni+0.1;
      else if (x<0 && z<0)
            valIni=valIni+0.15;

}
```

Per tant, un cop sabem quina cantonada del billboard és i les distàncies hi ha fins el centre, podem calcular quina posició té el centre del billboard. Aquest és un tros del codi del vèrtex shader que s'utilitza per a calcular el centre del billboard, només es mostra el cas a l'hora de tractar el vèrtex de dalt a l'esquerra ja que per la resta de vèrtexs seria molt similar:

```
//si es compleix la condició voldrà dir que és una de les 2 cantonades de la
part de dalt
if (c[0]>0.5)
 {
      superior=1;
      c[2]=c[2]*(-1);
}
//calcul posicio centre
if (c[0]>0.78)//vertex dalt esquerra
{
      //distancia x negativa i z positiva
      if (c[0]>0.83 && c[0]<0.87)
            c[1]=c[1]*(-1);
      //distancia x positiva i z negativa
      else if (c[0]>0.88 && c[0]<0.92)
            c[3]=c[3]*(-1);
      //distancia x negativa i z negativa
      else if (c[0]>0.93)
      {
            c[1]=c[1]*(-1);
            c[3]=c[3]*(-1);
      }
      //els altres casos volen dir que les distàncies x i z són positives
}
...
```

Un cop s'ha tractat tots els vèrtexs, ja es pot calcular el centre del billboard:

```
float3 center=pInitial.xyz;
center.x=c[1]*10.0+pInitial.x;
center.y=c[2]*10.0+pInitial.y;
center.z=c[3]*10.0+pInitial.z;
```

Ara que ja coneixem quina és la posició del centre dels billboards, els resultats dels càlculs dels angles seran els correctes i cada cantonada obtindrà els mateixos angles després de fer els càlculs. Per tant, ara que ja disposem dels angles, es pot començar a calcular quines imatges li toca a cada billboard.

3.3.2.3 Assignació de textures

Com ja s'ha vist al capítol: 2.5 cada imatge de la base de dades ve definida per 3 angles concrets, per exemple, aquesta imatge:

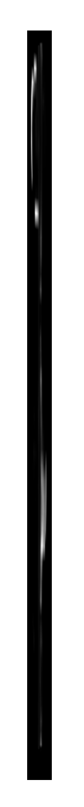


Figura 3.34, imatge d'una gota d'aigua, extreta de la base de dades d'imatges. Concretament és la imatge: cv20_v10_h10_osc3

S'hauria de visualitzar quan la càmera estigui a un angle respecte la gota d'aigua de 20 graus, i el punt de llum a 10 graus tant per el que fa a l'alçada com a l'angle azimut. Recordem que la base de dades d'imatges està distribuïda segons aquests tres angles, i els valors que poden prendre són els següents:

- Angle de la càmera respecte la gota d'aigua: Les imatges podran estan definides per cinc valors diferents, aquests valors aniran des del 0 al 80 (ambdós inclosos) amb intervals de 20. És a dir de 0 a 80º amb intervals de 20,o sigui: 0º, 20º, 40º etc.

- Angle del punt de llum respecte la gota d'aigua: Les imatges estaran definides per deu valors diferents, aquests valors aniran des del -90º als 90º (ambdós inclosos) amb intervals de 20.
- Angle azimut: Les imatges podran està definides per nou valors diferents, aquests valors aniran des dels 10º als 170º (ambdós inclosos) amb intervals de 20.

Per tant, com coneixem els angles que formen en cada moment el billboard respecte els punts de llum i la càmera, haurem d'agafar aquests valors i trobar, per cada un dels tres angles, quin angle dels possibles s'aproxima més. Un cop es tinguin aquests tres angles, ja sabrem quina imatge s'ha d'utilitzar per aquell billboard.

Primerament calcularem quins angles li pertoquen a cada angle calculat. Per aconseguir-ho, per cada un dels tres angles, hauríem de restar-li el seu valor per un dels especificats, el que tingui la diferencia més petita serà el que més s'hi aproximi i per tant el valor que agafarem per a seleccionar la imatge.

Per a realitzar aquests càlculs al vèrtex shader es podria fer amb tres bucles, un per cada angle, per estalviar recursos i per tant augmentar velocitat en l'execució, s'han combinat els tres bucles per fer-ho amb un de sol. A continuació s'explicarà la funció calculaLongImatge per parts:

```
void calculaLongImatge(out int longI,out int longF,out int pC,float aCamera,float alLlum,float phi)
{
        //ara que ja tenim els angles, hem de saber per cada gota, quin és l'angle que li correspon, per fer això hem de comprobar, en quin dels possibles angles hi ha menys diferències amb el que ha sortit.
        int pA=-90,pPhi=10,iA=-1,iPhi=-1;
        int iC=-1,millorPa=10,millorPc=0,millorPphi=10;
        pC=0;
        float rC,rA,rPhi,millorC=200,millorA=200,millorPhi=200;
        bool trobatC=false,trobatA=false,trobatPhi=false,acabat=false;
...
```

Per millorar amb velocitat, només es crea un bucle, ja que si se'n crea un per cada angle, faries moltes més repeticions que si se'n fa un de sol, per tant, aniria més lent i per això es fa un bucle. En aquest es busca, per cada angle, quin li pertoca. Com el que hi ha més angles possibles és el de l'alçada de la llum, el mentre, es farà fins que no s'hagi examinat tots els angles d'aquest, o fins que no haguem trobat tots els angles. Això passarà quan haguem trobat per cada angle, una diferència que sapiguem que ja no n'hi haurà cap de més petita.

```
while (pA<91 && !acabat)
       {
              //si de cada angle no hem trobat el que li toca, augmentem la seva i.
              if (!trobatC)
                     iC++;
              if (!trobatA)
                     iA++;
              if (!trobatPhi)
                     iPhi++;

              //calculem la diferencia amb un angle possible, marcat per pX.
              rC=abs(aCamera-pC);
              rA=abs(alLlum-pA);
              rPhi=abs(phi-pPhi);
...
```

Comprovem si la diferencia que acabem de trobar, és inferior o no a la diferència més petita que s'ha trobat, això es farà únicament si no s'ha trobat arribats a aquest punt l'angle que li pertoca. Vegem-ho:

```
       if (rC<millorC && !trobatC)
       {
              millorC=rC;millorPc=pC;
       }
       if (rA<millorA && !trobatA)
       {
              millorA=rA;millorPa=pA;
       }
       if (rPhi<millorPhi && !trobatPhi)
       {
              millorPhi=rPhi;millorPphi=pPhi;
       }
       //posarem a true que s'ha trobat l'angle si el resultat de millor és inferior a
       10
       if (millorC<=10)
              trobatC=true;
       if (millorA<=10)
              trobatA=true;
       if (millorPhi<=10)
              trobatPhi=true;
       //haurem acabat si hem trobat els 3 angles.
       if ((trobatC && trobatA && trobatPhi))
              acabat=true;
...
```

Finalment augmentem els diferents comptadors.

```
//augmentem l'angle de la camera
            if (pC<81 && !trobatC)
                  pC=pC+20;
            //i el de phi, finalment el de l'alçada de la camera, en aquell no cal
            comprobà si s'ha trobat o no, ja que ja es fa al while.
            if (pPhi<171 && !trobatPhi)
                  pPhi=pPhi+20;

            pA=pA+20;
      }
      //com les imatges totes tenen la mateixa amplada, podem trobar la posició
      que pertoca fent les multiplicacions amb iA i iPhi, que les hem trobat al
      bucle
      longI=(144*iA)+(16*iPhi);
      longF=longI+16;
}
```

Aquesta funció que s'acaba de veure apart de calcular quin és l'angle que li pertoca segons els calculats, també ja calcula a quin píxel de l'atles començaria la imatge seleccionada i en quin acabaria, només ho calcula per l'amplada, per l'alçada es calcularà fora d'aquesta funció. Aquesta és la part del codi anterior que ho realitza:

```
      longI=(144*iA)+(16*iPhi);
      longF=longI+16;
```

Per entendre aquest pas, cal recordar com s'ha generat anteriorment l'atles on es guardaran totes les imatges de les gotes d'aigua. Multipliquem 144*iA (iA dins de l'interval -90 a 90, indica quin angle és, per tant si iA=0, voldrà dir que agafarem una imatge que tingui un angle del billboard respecte del punt de llum de -90, si és 1 voldrà dir que l'angle serà de -70 i així successivament), ja que coneixem l'estructura de l'atles, i coneixem que cada 144 píxels d'amplada l'angle d'alçada entre el punt de llum i el billboard s'augmenta en 20 º. Per acabar de conèixer a on comença la imatge que li pertoca li sumarem 16*iPhi, ja que cada imatge té una amplada de 16 píxels i iPhi marca el valor de l'angle azimut de la imatge que li pertoca al billboard. Com l'angle azimut pot anar des de 10 fins a 170 si iPhi té un valor de 0, voldrà dir que l'angle de la imatge seleccionada té d'angle azimut 10º, si iPhi val 1, d'angle azimut tindrà 30º i així amb la resta de valors. Es pot veure més gràficament a la imatge 3.35, on podem veure una petita part de l'atlas generat amb indicacions per poder entendre els càlculs que s'acaben de descriure.

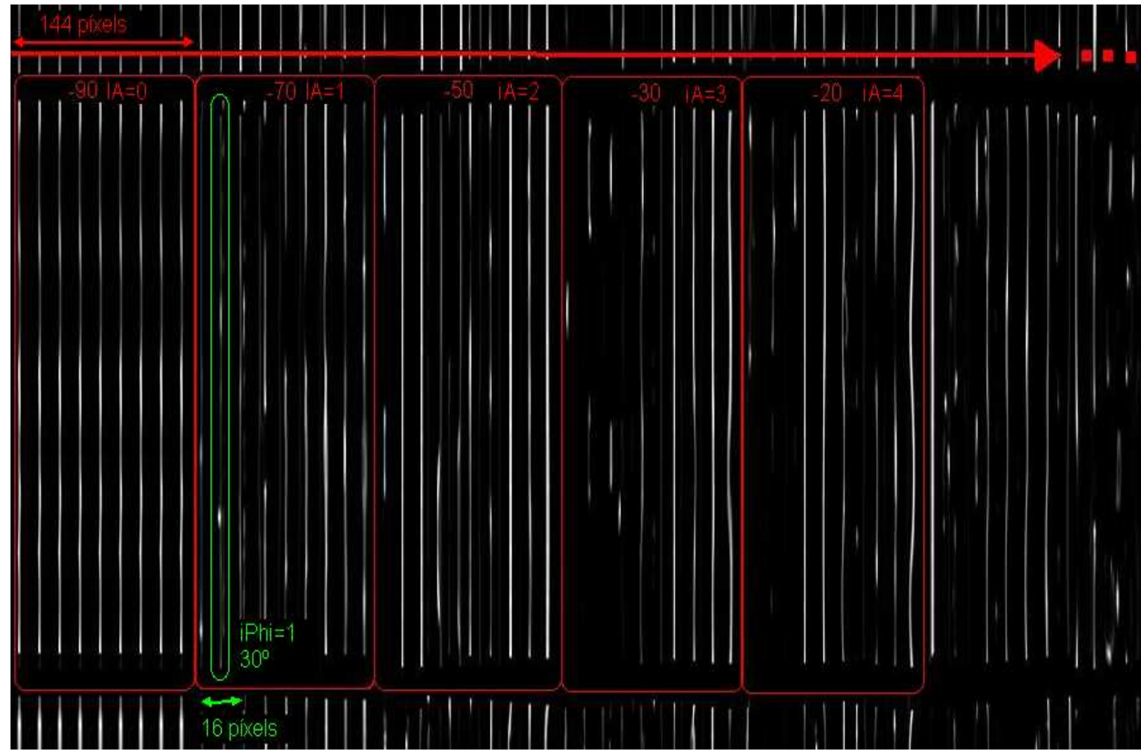


Figura 3.35, podem veure la representació gràfica dels càlculs realitzats per poder trobar el píxel inicial de la imatge que li pertoca segons els angles resultants. Amb color verd podem veure la informació sobre la variable iPhi, iPhi marcarà l'angle azimut que tindrà la gota d'aigua ja que totes les imatges que estiguin dins d'un mateix bloc tindrà el mateix angle respecte la camera i també el mateix angle respecte el punt de llum, però diferent angle azimut. Aquests valors van de 10 a 170º i estan ordenats de forma ascendent. El color vermell mostra tota la informació pel que fa a la variable iA, la qual indicarà l'angle del billboard respecte el punt de llum. Com veiem hi ha fins a 10 angles diferents possibles. Sabem que dins de cada un d'aquests blocs hi ha 9 imatges de 16 píxels d'amplada (les quals tindran el mateix angle de camera, de punt de llum, però diferent angle azimut), per tant podem saber l'amplada de cada un d'aquests deu blocs: 16*9=144 píxels. Sabent aquesta informació serà fàcil determinar la posició dins de l'atlas de qualsevol imatge.

Ara ja sabem a quins angles s'aproximen els anteriorment calculats, també coneixem on a l'atles comença la imatge seleccionada i on acaba per el que fa a l'amplada. Per tant, falta saber a quin és el píxel inicial i el final per el que fa a l'alçada.

Per cada angle θview que pugui tenir la imatge, li pertocarà una alçada diferent, per tant això dificulta poder calcular les posicions inicials i finals dels píxels amb una fórmula matemàtica. Les imatges tenen 5 angles θview possibles, i com ja hem dit 5 alçades diferents per cada imatge que tingui un angle θview diferent. A continuació es mostra el llistat dels 5 angles possibles amb les dimensions de les imatges:

- Amb angles θview= 0º es seleccionaran imatges de mida: 16*525 píxels.
- Amb angles θview= 20º es seleccionaran imatges de mida: 16*494 píxels.
- Amb angles θview= 40º es seleccionaran imatges de mida: 16*405 píxels.
- Amb angles θview= 60º es seleccionaran imatges de mida: 16*272 píxels.
- Amb angles θview= 80º es seleccionaran imatges de mida: 16*108 píxels.

Com realitzar una fórmula matemàtica era més costós que per el cas de l'amplada, s'ha assignat la posició inicial i final dels píxels d'alçada de la imatge seleccionada, amb cinc condicionals. Aquest és el codi del vèrtex shader:

```
//segons el valor que hagi quedat a pC (el de la diferencia mínima amb
l'angle sortint), li assignarem un parell de alçades o un altre parell.
if (pC==0){
      alsadaI=0;
      alsadaF=525;
}
else if (pC==20){
      alsadaI=525;
      alsadaF=1019;
}
else if (pC==40){
      alsadaI=1019;
      alsadaF=1424;
}
else if (pC==60){
      alsadaI=1424;
      alsadaF=1696;
}
else if (pC==80){
      alsadaI=1696;
      alsadaF=1804;
}
```

Un cop ja es coneix els diferents píxels inicials i finals de la imatge que li pertoca mostrar al billboard, cal assignar-li al billboard què haurà de mostrar. Per fer-ho s'ha aprofitat una característica del llenguatge Cg, on es definim un texCoord (dos floats), en el qual per cada vèrtex del billboard, li assignarem dos reals entre 0 i 1. Aquests dos reals significaran el píxel de la coordenada x (és a dir d'amplada) i el píxel de la coordenada y (és a dir de l'alçada). Ja que un billboard disposa de quatre vèrtexs, un per cada cantonada, li assignarem a cada cantonada el píxel que li pertoca de l'atles. Com necessitem passar-li valors enter 0 i 1, els valors que abans s'han calculat s'han de normalitzar. El primer píxel de l'atles tindria el valor 0 i l'últim tindria el valor 1. Per posar un exemple podem imaginar que en un moment donat volem assignar a un billboard tota la imatge i no només una part com fem en aquest cas, en el que seleccionem només una part de l'atles. Veiem una imatge amb les coordenades que assignaríem a cada vèrtex:

Figura 3.36, aquest exemple ens indica els valors que s'hauríen d'assignar a la variable de tipus texcoord perquè un billboard mostrés tota aquesta imatge.

Per normalitzar els valors que s'assignaran perquè tinguin valors entre 0 i 1, els valors calculats sempre es dividiran pel màxim de la imatge. L'atles, d'amplada fa 1440 i d'alçada 1804, per tant sempre dividirem per aquests valors per poder mostrar de forma correcte la gota que li pertoca mostrar al billboard.

Com es pot veure a la figura 3.36, per assignar unes coordenades o unes altres cal primer conèixer de quin vèrtex del billboard es tracte. Gràcies al codi implementat i explicat en apartats anteriors, podem conèixer quin és el vèrtex que s'està tractant mitjançant condicionals. Veiem el codi d'aquestes assignacions: la variable uvB és una variable de tipus texcoord. El vector c, com ja s'ha vist en anteriors ocasions, indica el color del vèrtex, per tant serà aquesta variable la que ens ajudarà a reconèixer quin vèrtex és:

```
if (c[0]>0.78)//dalt esquerra
      uvB=float2(longI/1440.0,alsadaI/1804.0);
else if(c[0]>0.58 && c[0]<0.77)//dalt dreta
      uvB=float2(longF/1440.0,alsadaI/1804.0);
else if(c[0]>0.28 && c[0]<0.47)//baix esquerra
      uvB=float2(longI/1440.0,alsadaF/1804.0);
else if (c[0]<0.17)//baix dreta
      uvB=float2(longF/1440.0,alsadaF/1804.0);
```

Com ja s'ha explicat al principi d'aquest capítol disposem de dos tipus d'atles, el primer altes guardarà totes les imatges de gotes d'aigua corresponents a la visualització de les gotes d'aigua tenint en compte las diferents direccions de llum que hi pugui haver. Al segon atles hi podem trobar totes les imatges de gotes d'aigua corresponent a la llum ambiental. Fins ara tot el que s'ha fet és assignar les coordenades de l'atles per localitzar la imatge que li pertoca segons la posició del punt de llum i de la càmera. Ara falta assignar les coordenades al segon atles per seleccionar la imatge de la gota segons la llum ambiental, que únicament dependrà de la posició de la càmera respecte la gota d'aigua.

Com només depèn de l'angle θview, i ja que els càlculs els tenim realitzats, només ens cal assignar el resultat a una variable de tipus texcoord. A continuació es mostrarà les modificacions al codi anterior afegint l'assignació de la 'imatge de la gota d'aigua amb la llum ambiental, en aquest cas, la variable s'anomena uvC (la part del codi amb negreta és la part del codi que encara no s'havia mostrat):

```
        if (c[0]>0.78){//dalt esquerra
                uvB=float2(longI/1440.0,alsadaI/1804.0);
                uvC=float2(0,alsadaI/1804.0);

        }
        else if(c[0]>0.58 && c[0]<0.77){//dalt dreta
                uvB=float2(longF/1440.0,alsadaI/1804.0);
                uvC=float2(1,alsadaI/1804.0);
        }
        else if(c[0]>0.28 && c[0]<0.47){//baix esquerra
                uvB=float2(longI/1440.0,alsadaF/1804.0);
                uvC=float2(0,alsadaF/1804.0);
        }
        else if (c[0]<0.17){//baix dreta
                uvB=float2(longF/1440.0,alsadaF/1804.0);
                uvC=float2(1,alsadaF/1804.0);
        }
```

Així ara ja es tenen assignades la informació de les textures al vèrtex shader, aquestes variables s'utilitzaran al píxel shader per tractar el color de cada píxel dels billboards.

3.3.3 Redimencionar el tamany dels billboards

Com s'ha pogut veure en el capítol anterior, segons l'angle θview que s'hagi obtingut, les imatges seleccionades tindran una mida o una altra. Si es tracten tots els billboards com si tinguessin les mateixes dimensions, s'obtindria un resultat que no és el desitjat, ja que les proporcions entre alçada i amplada de la imatge de la base de dades variaria al mostrar-se a l'escena i per tant deformaria la seva forma inicial al il·lustrar-se al billboard i baixaria la realitat de la visualització de la gota d'aigua.

Per redimensionar la mida dels billboards, s'ha establert un mida inicial d'aquest, ja que així ja coneixerem quin és la mida del billboard al tractar un vèrtex al shader. S'ha establert que la mida inicial del billboard sigui corresponent a la proporció de la imatge amb dimensions més grans, és a dir les imatges amb angle θview=0, les quals tenen una mida de 16*525. Si dividim 16 entre 525 obtindrem una proporció entre alçada i amplada de 0.0304.

Després de realitzar vàries proves de visualització s'ha establert que una alçada de 14 unitats (unitats arbitràries d'Ogre3D) del billboard podia ser una bona alçada màxima per als billboards que s'utilitzaran per la pluja. Per tant, a la creació dels billboards des del frameStarted de la classe frameListener s'ha establert que la mida inicial i màxima dels billboards serà de 0.426 unitats d'amplada per 14 d'alçada. Com podem comprovar té la mateixa proporció entre l'amplada i l'alçada del billboard, 0.0304.

3.3.3.1 Fórmula per redimensionar la mida del billboard

Sabem que cada vegada que l'algorisme del vèrtex shader tracti cadascun dels 4 vèrtex, inicialment, tindrà sempre la mateixa distància entre ells. També sabem que al principi d'aquest algorisme, per simular el moviment de la gota d'aigua es col·loca cada vèrtex a una nova posició seguint la fórmula ja explicada en capítols anteriors (Veure capítol 3.2.5.2 Pluja en temps real):

```
pInitial.y= pInitial.y - 0.5 * time ;
```

Per tant cada vegada que es tracta un dels vèrtexs del billboard al vèrtex shader, situem el vèrtex a la posició que li pertoca, això seria cert si tots els billboards tinguessin la mateixa mida, l'inicial, el definit des del frameListener al moment de crear-lo. Segons l'angle θview caldrà retocar una mica la posició de cada vèrtex per així fer que el billboard tingui la mida que li pertoca. Si l'angle θview=0, la mida del billboard serà la mateixa que la definida al crear al billboard, per tant en aquest cas no s'haurà de retocar la posició del vèrtex.

A l'hora de reduir la mida dels billboards cal tenir en compte que cada vèrtex s'haurà de fer càrrec de la meitat del total de la reducció d'aquell billboard, ja que els vèrtexs superiors disminuiran una mitja part i els inferiors l'altre part. Veiem un exemple hipotètic de com s'hauria de modificar la mida dels billboards amb la següent imatge, en aquest exemple la mida inicial del billboard és de 10 unitats, el segon billboard s'ha de reduir en 2 unitats, per tant restem la posició dels vèrtexs superiors en una unitat i als inferiors li sumem també una unitat, per tant la nova alçada del billboard és de 8 unitats. El mateix passa amb el tercer billboard, el més petit de tot, en el que la posició dels vèrtexs superiors se li restaria 2 unitats i als inferiors se li sumarien 2 unitats de les posicions inicials, així l'alçada final d'aquest billboard es quedaria amb 6 unitats, com veiem a la imatge 3.37.

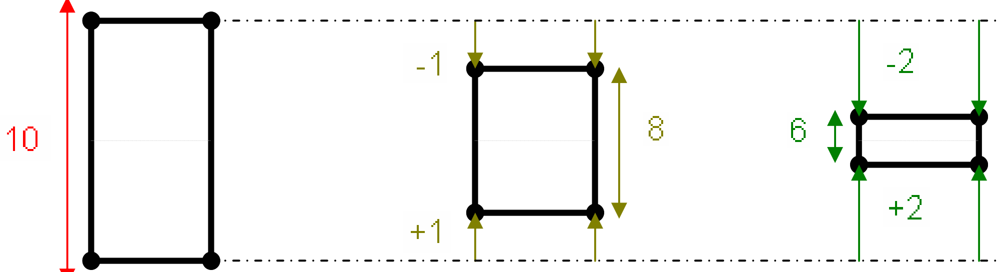


Figura 3.37, exemple que mostra com es reduirà el tamany dels billboards, les dades són de forma hipotètica.

En resum, el que s'ha fet és primer de tot calcular l'alçada que tindria el billboard aprofitant les variables alçadaI i alçadaF ja calculades. Després s'ha calculat la proporció d'aquesta alçada amb els 16 píxels que sabem que té d'amplada cada imatge. Al conèixer l'amplada del billboard la qual és de 0.426 unitats es podrà trobar l'alçada que haurà de tenir aquest billboard. Per acabar només cal restar a aquesta alçada l'alçada inicial del billboard, la qual ja coneixem. Amb això obtenim la diferència que hi haurà entre el billboard inicial i el que s'ha de modificar. Si es divideix per 2 aquest valor, s'obtindrà l'increment que haurà de patir cada vèrtex per aconseguir la mida del billboard desitjat, aquest és el codi corresponent a la fórmula que s'acaba d'explicar:

```
float alsadaInicial=c[2]*2*10.0;//d'aqui obtenim
l'altura inicial del billboard.
//calcul alçada
float alsadaGotes=alsadaF-alsadaI;
incrementGotes=(((alsadaGotes*0.426)/16)-alsadaInicial)/2.0;
```

Finalment, cal incrementar o decrementar, depenent de si és un vèrtex superior o inferior del billboard. Aquest codi es dipositarà quan es comprovi quin vèrtex és:

```
pInitial.y=pInitial.y+incrementGotes;
```

Amb aquest codi els billboards sempre tindran la mateixa proporció que la imatge pel que fa a la proporció amplada/altura, per tant la visualització que podrà obtenir cada gota d'aigua serà excel·lent.

3.3.4 Visualització amb diferents punts de llum a l'escenari

Fins ara només s'ha tractat que la pluja que pugui caure en un escenari només tingui en compte un únic punt de llum. El més normal és que en un escenari hi hagi més, per tant si es vol obtenir una pluja realista cal tenir en compte tots aquests punts de llum i així veure els diferents efectes que tindran aquests punts de llum sobre la pluja.

Per tant el que es necessitarà serà poder obtenir tants vectors de 2 reals de tipus texcoord com punts de llum tingui l'escena. És a dir, s'haurà de repetir el càlculs dels angles, menys el càlcul de l'angle θview, així com l'assignació de les textures, tantes vegades com punts de llum tingui l'escena. El càlcul de la selecció de la imatge de les gotes amb la llum d'ambient es mantindrà igual, ja que només dependrà del punt de llum que realitzi la llum ambiental. Veiem la declaració al vèrtex shader tant del vector de texcoords com dels diferents punts de llum, en aquest cas dos punts de llum.

```
…
out float2  uvB[2] : TEXCOORD0,
uniform float4 lightPosition0,
uniform float4 lightPosition1,
…
```

Un cop feta la declaració del vèrtex shader, es crearà manualment un vector de punts de llum, vegem-ho:

```
float4 lightPos[2];
lightPos[0]=lightPosition0;
lightPos[1]=lightPosition1;
```

Algunes parts del codi vistes durant els últims capítols es repetiran dins d'un bucle amb tantes iteracions com números de punts de llum tingui l'escenari. Aquest bucle començarà un cop s'hagi calculat el centre del billboard.

la modificació de l'alçada dels billboards i l'assignació de les coordenades de l'atles de la imatge de llum ambiental només es realitzarà una sola vegada, ja que no depèn dels diferents punts de llum. S'ha implementat una altra funció anomenada calculaLongImatgeB que realitza exactament la mateixa tasca que calculaLongImatge però únicament no calcula a quin angle θview s'aproxima més dels angles possibles, ja que només cal calcular-l'ho una sola vegada, la qual es calcularà a la primera passada del bucle amb la funció calculaLongImatge. Podem veure el codi complert a l'annex.

Per poder tractar els punts de llum i per tant rebre informació, entre altres coses de la seva posició, ha calgut informar al vèrtex shader que hi haurien aquests paràmetres, des del fitxer pluja.material s'han hagut d'afegir varis paràmetres: a més a més del temps i de la càmera podem veure com s'ha declarat els dos punts de llum (si a l'escenari s'hi col·loquessin més punts de llum, caldria introduir-los també aquí). Veiem com s'ha modificat l'arxiu pluja.material:

```
material pluja/pluja
 {
   technique
   {
     pass
     {
       vertex_program_ref pluja/plujaVp
       {
         param_named_auto time time_0_x 5000
         param_named_auto camera camera_position
         param_named_auto lightPosition0
      light_position_object_space 0
         param_named_auto lightPosition1
      light_position_object_space 1
       }
…
```

Amb tots aquests aspectes que s'han tractat fins al moment, s'entrega al píxel shader una informació que serà molt profitosa per aconseguir una visualització el més realista possible.

3.3.5 Aportació del píxel shader

En els darrers apartats, entre altres aspectes, s'ha explicat com calcular i assignar les coordenades de les imatges de gotes d'aigua que tocaven segons la posició de la càmera i els diferents punts de llum que hi pugui haver a l'escenari respecte cada billboard. Ara cal aplicar aquestes coordenades per assignar un color a tots els píxels que formin els diferents billboards que s'hagin creat.

Primer de tot, al píxel shader l'hi caldrà saber de quines imatges s'haurà d'obtenir les textures que venen especificades per les coordenades guardades en les variables de tipus texcoord. Això s'ha d'especificar des del material, és a dir des del fitxer pluja.material. A la primera imatge, la qual serà l'atles de gotes d'aigua corresponents a la base de dades d'imatges puntuals, se l'anomenarà al píxel shader imatgePuntuals. A la segona imatge, corresponent a l'atles generat per les imatges de la base de dades d'imatges de llum ambiental, se li ha assignat el nom de imatgeAmbient. Veiem aquestes definicions:

```
texture_unit imatgePuntuals
      {
            texture atles.jpg
      }

      texture_unit imatgeAmbient
      {
            texture atlesAmbient.jpg
      }
```

També ha calgut definir una sèrie de paràmetres que seran claus per l'aparença i el comportament final de cada billboard. S'ha especificat que la variable depth_write estigui a off, ja que com les gotes d'aigua contenen un

percentatge de transparència, si no es posiciona a off les gotes d'aigua no tindran cap transparència i per tant no seran realistes. També existeix una variable per els materials d'Ogre en el que s'especifica el comportament del color en relació al color de l'ambient i el del material que s'està tractant, en aquest cas els billboards. Aquesta variable és l'anomenada scene_blend, la qual s'ha definit que sigui de tipus Add, la qual significa que el color de renderització del material tractat, és afegit a l'escena. Veiem aquestes definicions dipositades també al .material:

```
      depth_write off
      scene_blend add
```

Al píxel shader com s'explicarà a continuació es necessitarà saber el color de la llum que generi els diferents punts de llum, tant la difusa com l'ambiental. Es definiran tants punts de llum difusos com n'hi hagin a l'escena, apart d'aquests també es definirà la llum d'ambient, el qual com ja s'ha explicat en els capítols anteriors només n'hi haurà una. Veiem com s'han definit els paràmetres de que disposarà el píxel shader:

```
fragment_program_ref pluja/plujaPS
{
      param_named_auto lightdiffuse0 light_diffuse_colour 0
      param_named_auto lightdiffuse1 light_diffuse_colour 1
      param_named_auto lightambient ambient_light_colour
}
```

Aquest últims paràmetres que s'acaben d'afegir i els quals s'utilitzaran al píxel shader, es defineixen aquí ja que són els que tenen relació directa amb Ogre. En el cas del codi anterior, el primer llum definit des de les classes del projecte cedirà la informació del color de la seva llum d'ambient al píxel shader, el segon objecte de llum definit a Ogre cedirà la seva informació al paràmetre lightdiffuse1 i així successivament. Amb aquesta declaració, Ogre ja farà aquesta tasca de forma automàtica. Ara que ja s'ha definit el material, i per tant alguns dels paràmetres que necessitarà el píxel shader, podem veure amb gran quantitat de comentaris quins seran els paràmetres del píxel shader que s'han necessitat:

```
void main_ps(
        //inputs
        //vector de float2s en el que s'hi guardaran les coordenades
        seleccionades des del vertex shader per les imatges puntuals
        float2 uv[2]: TEXCOORD0,
        //Guardarà les coordenades de la imatge seleccionades de l'atles de
        llum d'ambient.
        float2 uvC: TEXCOORD5,
        //imatge: atles d'imatges puntuals
        uniform sampler2D imatgePuntuals : register(s1),
        //imatge: atles d'imatges d'ambient
        uniform sampler2D imatgeAmbient : register(s2),
        //color de la llum difusa que genera la primera llum creada a
        l'escenari utilitzat
        uniform float4 lightdiffuse0,
        //color de la llum difusa que genera la segona llum creada a
        l'escenari utilitzat
        uniform float4 lightdiffuse1,
        //color de la llum ambiental que genera la primera i principal llum
        creada a l'escenari utilitzat
        uniform float4 lightambient,
        //outputs
        //color resultant, els billboards agafaran el color definit en aquesta
        variable
        out float4 color : COLOR
)
```

Com ja s'ha explicat, la única finalitat del píxel shader, és determinar quin serà el color final que agafaran els billboards perquè es puguin visualitzar com a gotes el màxim de realistes possible. Coneixent els paràmetres dels que disposem, observant el codi anterior, l'objectiu és que gràcies a les coordenades calculades al vèrtex shader s'obtingui primer la imatge de l'atles d'imatges puntuals i que cada píxel de la imatge s'il·lustri a la posició que li pertoca del billboard. Un cop es disposi del color corresponent al billboard, es multiplicarà el valor de cada component del color per el de la llum difusa del punt de llum de l'escena. Finalment per acabar d'adequar el color es multiplicarà per un factor. Aquest procés s'haurà de repetir tantes vegades com punts de llum hi hagi a l'escena sumant els valors resultants entre ells per obtenir un únic color. Aquest serà el color que li pertoca agafant les imatges puntuals i la llum difusa generada per totes les imatges puntuals. Per acabar, aprofitant la variable uvC s'haurà d'obtenir la imatge de l'atles d'imatges d'ambient, i com s'ha fet en les imatges puntuals, se li multiplicarà el color de la llum d'ambient i un factor. El color final de la gota d'aigua vindrà determinat per la suma del color calculat per les llums puntuals i les llums d'ambient.

Per assignar a cada píxel, el píxel que li pertoca, s'ha utilitzat una funció pròpia del llenguatge Cg, la funció anomenada tex2D, la qual passant-li com a primer paràmetre un atribut de tipus sampler2D, com és el cas dels dos atles, i passant-li com a segon paràmetre una coordenada de textura, obtindrem el color de la imatge que s'ha especificat en aquestes coordenades. Per tant, per

calcular primerament el color que li pertoca de les imatges puntuals i agafant el color que generen els diferents punts de llum, s'ha realitzat un bucle en el que s'anirà repetint el mateix procediment tantes vegades com punts de llum hi hagi. El procés que es repetirà serà el de l'obtenció del píxel de l'atles d'imatges puntuals amb l'ajuda de la funció tex2D, multiplicar aquest valor per el color de la llum difusa del punt de llum. I finalment multiplicar aquest color per un factor. A cada iteració s'anirà acumulant el color obtingut, veiem el codi descrit:

```
//es preparan les variables per poder fer el bucle
        float4 lightDif[2];
        lightDif[0]=lightdiffuse0;
        lightDif[1]=lightdiffuse1;
        //inicialitzo el color
        color=float4(0,0,0,1);
        //es realitzarà el mateix procés tantes vegades com punts de llum hi hagi
        for (int numLlums=0;numLlums<2;numLlums++)
        {
              color=color+tex2D(imatgePuntuals,uv[numLlums])*lightDif[
              numLlums]*0.15;
        }
```

D'aquesta manera ja es tindrà el color de les imatges puntuals, falta sumar-li la corresponent a la llum d'ambient, veiem el codi a on es veu com es calcula el color d'ambient i es suma al color final:

```
        //calculem el valor del color corresponent a la llum ambiental
        float4 ambient=tex2D(imatgeAmbient, uvC);
        ambient=ambient*lightambient*0.015;
        //sumem el color de les imatges puntuals amb el d'ambiental, això serà el
        color final
        color=color+ambient;
```

Amb el vèrtex shader i el píxel shader ja dissenyats, podem veure la següent imatge la qual mostra el resultat obtingut:

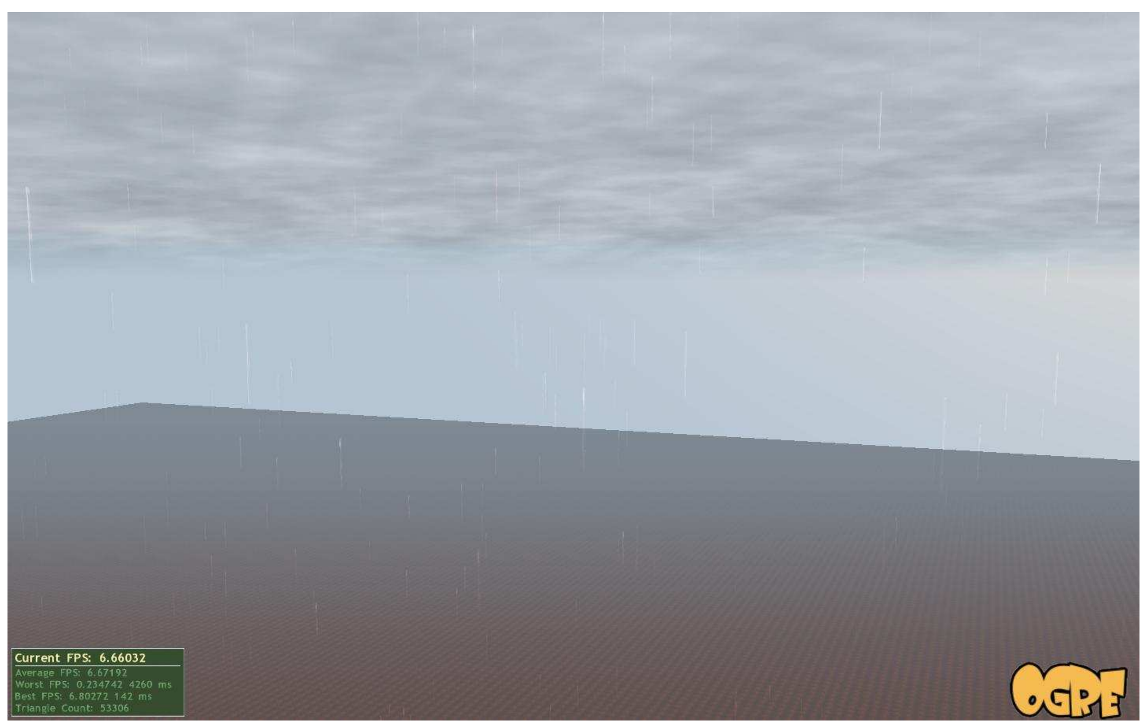


Figura 3.38, mostra d'execució de l'aplicació en funcionament.

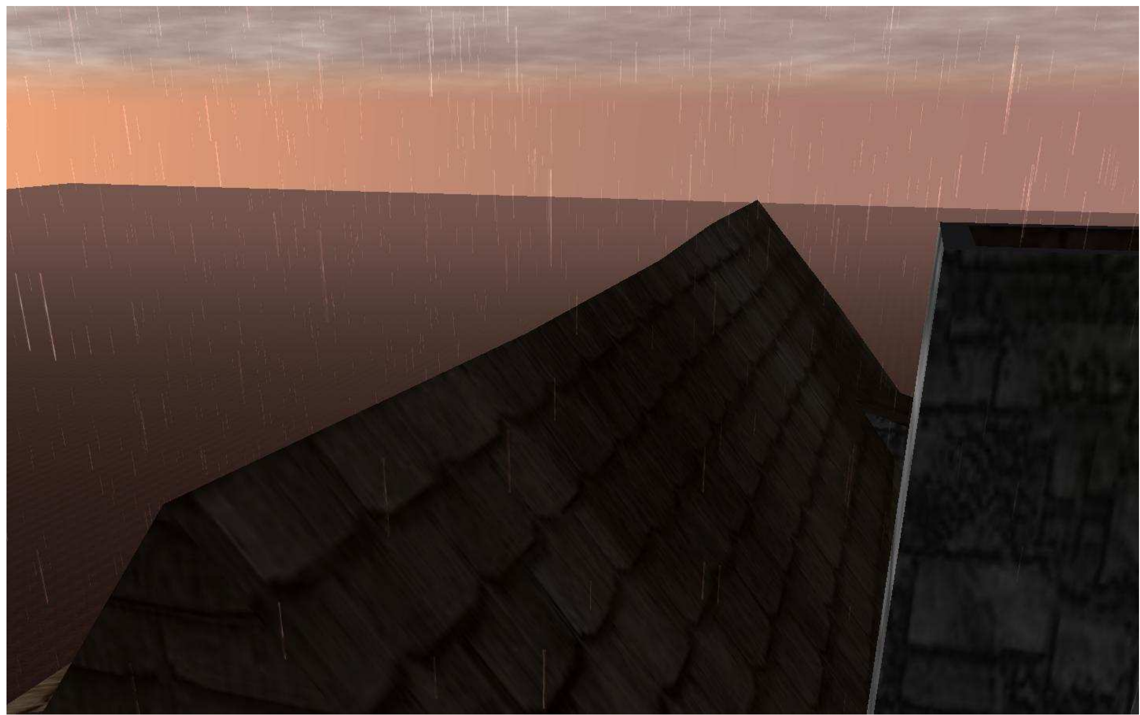


Figura 3.39, mostra d'execució de l'aplicació en funcionament.

3.3.5.1 Tractament de la visualització segons el tipus de punt de llum

Fins ara s'han tractat tots els punts de llum de manera iguala, però alguns punts de llum simulen la llum de fanals, cotxes etc. En aquests casos la llum no es reflecteix en tot l'escenari amb la mateixa intensitat, sinó que com en la realitat, en un determinat angle i una distancia respecte el focus de llum, aquesta es veurà amb un color diferent (si és que s'hi observa la llum) que per exemple un altre angle i una distància més llunyana. Aquest tipus de llum, Ogre els defineix com spotlights. En aquest apartat es tractarà com es pot realitzar perquè sigui realista i per tant, si s'incorpora un fanal a l'escenari no afecti a totes les gotes de l'escenari sinó que només a les que passin per la zona d'on el fanal reflecteix la llum.

S'ha afegit un llum de tipus spotlight, per així poder comprovar si la implementació que s'està explicant funciona de forma correcte. Com es pot veure a la figura 2.29 els llums de tipus spotlight tenen franges amb intensitats diferents, la primera és una franja de llum la qual tindrà la intensitat màxima, la definida al llum. Aquesta franja serà definida per un angle interior. Un cop es superi aquest angle la intensitat de la llum s'anirà disminuint, fins que arribats a un angle determinat deixi de tenir intensitat. Aquests dos angles s'anomenen angle interior i exterior respectivament. Veiem com s'ha definit que el segon llum de l'escenari sigui de tipus spotlight i també com s'han establert els diferents angles que tindrà el llum.

```
Light *lightb = mSceneMgr->createLight("Light1");
lightb->setType(Light::LT_SPOTLIGHT);
lightb->setPosition(Vector3(-700,550, -50));
lightb->setDiffuseColour(1.0, 0.0, 0.0);
lightb->setSpecularColour(1.0, 0.0, 0.0);
lightb->setDirection(0,-1,0);
lightb->setSpotlightRange(Degree(50), Degree(65));
```

La posició a la qual s'ha afegit aquest llum s'hi ha dipositat un objecte en forma de fanal per així reconèixer on és el llum spotlight i poder observar quin és el comportament de les gotes d'aigua al passar per la zona del fanal. Veiem el fanal a la figura 3.40.

Ara que ja s'ha incorporat un llum de tipus spotlight tractarem el procés de l'impacte de les gotes d'aigua amb el llum. Per fer-ho més realista i exacte, no només tractarem si cada gota cal ser il·luminada per el fanal sinó que també controlarem cada fragment de la gota, és a dir, pot ser que uns fragments de la gota estiguin il·luminats per aquell fanal i pot succeir que n'hi hagi d'altres que no, o bé per una intensitat més baixa. Per aconseguir això necessitem conèixer la posició en 3D de cada píxel.

Caldrà afegir dos paràmetres al vèrtex shader, un d'entrada i un altre de sortida. El d'entrada serà una matriu de 4*4 la qual ens permetrà realitzar la transformació per passar la posició des del vèrtex shader fins al píxel shader al mon. El paràmetre de sortida serà la posició resultant.

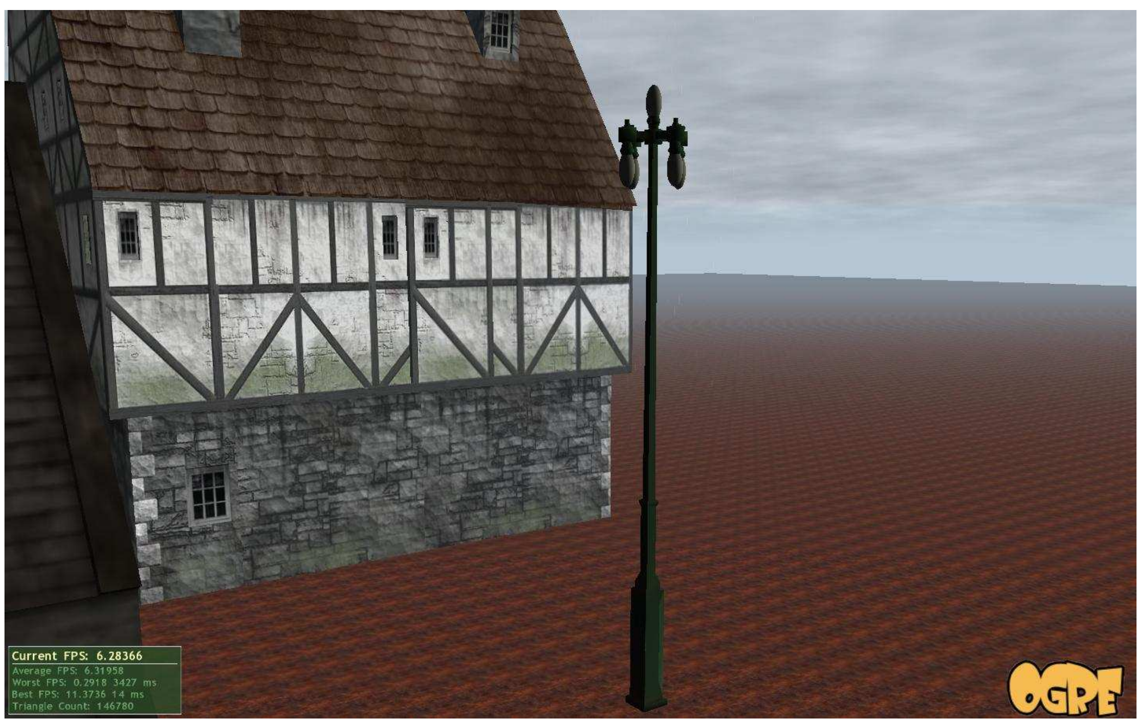


Figura 3.40, imatge de l'escenari en la qual hi podem veure el fanal que s'hi ha afegit. A la part superior del fanal, a la zona dels llums, és on hi ha el punt de llum de tipus spotlight.

Primer de tot s'ha definit al fitxer pluja.material, que el vèrtex shader necessitarà aquesta matriu, la qual l'Ogre calcula automàticament. Definint-la aquí el vèrtex shader ja podrà obtenir aquesta matriu com un paràmetre d'entrada més, veiem aquesta definició:

```
param_named_auto modelWorld world_matrix
```

També afegim el paràmetre de sortida al vèrtex shader, en aquesta ocasió serà un semàntic TEXCOORD i no POSITION ja que el píxel shader no deixa rebre paràmetres de tipus POSITION, a causa de que aquests tenen la posició respecte l'objecte, i al píxel shader es defineix respecte les coordenades del mon. Veiem com s'han afegit els dos paràmetres al vèrtex shader:

```
…
out float3 oWorldPos : TEXCOORD4,
uniform float4x4 modelWorld,
...
```

A continuació calculem la posició, s'agafarà la posició del vèrtex en coordenades respecte l'objecte i es multiplicarà per la matriu que s'acaba de passar per paràmetre, així realitzarem la transformació i des del píxel shader coneixerem la posició 3D de cada fragment. Veiem el codi de la transformació explicada:

```
oWorldPos = mul( modelWorld, pInitial ).xyz;
```

Ara que ja coneixem la posició en 3D de cada fragment de la gota d'aigua, necessitem saber la direcció de la llum, la posició del llum, i l'angle interior i exterior del llums de tipus spotlight, a la figura 2.9 de l'apartat 2.2.1 Objectes principals d'Ogre, podem veure un exemple d'un llum de tipus spotlight, observant la figura podem veure que a la part interior la llum és més forta que a l'exterior, la part interior és la que ve definida per l'angle interior. Aquests tres paràmetres també els proporciona Ogre sense que els tinguis que passar expressament des del codi, per tant els definirem des del pluja.material, vegem-ho:

```
param_named_auto lightPosition1 light_position_object_space 1
param_named_auto lightDirection1 light_direction_object_space
1
param_named_auto spotLightParams1 spotlight_params 1
```

El paràmetre de tipus light_direction_object_space indica que passarem al píxel shader la direcció de la llum indicada en el llum especificat en el següent paràmetre, en aquest cas, com veiem, el següent paràmetre és 1, ja que sabem que la llum de tipus spotlight és la número 1. Com veiem també sabrem la posició del llum amb l'atribut lightPosition1.

El paràmetre de tipus spotlight_params també dóna informació de la llum indicada en el pròxim paràmetre, com veiem també és 1, ja que ens referim al mateix llum. Aquest paràmetre és un vector de quatre reals. Si la llum de la qual ha de donar informació és de tipus spotlight, el primer element del vector donarà el resultat de fer el cosinos de l'angle interior dividit per 2. El segon element donarà el valor de fer el cosinos de l'angle exterior dividit per 2. El tercer element és sempre 0 i el quart 1.

Si la llum de la qual ha de donar informació no és de tipus spotlight, el vector de quatre reals tindrà els següents valors: (1,0,0,1).

Ara que ja disposem de tota la informació per el que fa tant al píxel com al llum, ja podem calcular la intensitat de llum que aportarà el fanal a cada píxel de la gota d'aigua.

Per fer-ho necessitem saber l'angle que forma el vector que va des del punt de llum fins a la gota d'aigua amb el vector de direcció de la llum. Veure la figura 3.41 per observar el gràfic de l'angle que formen els dos vectors.

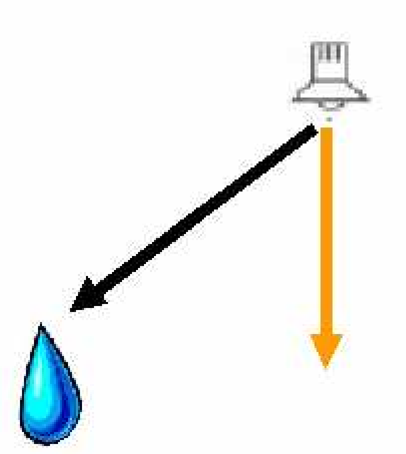


Figura 3.41, l'angle que busquem és el que formen entre ells els dos vectors. El negre representa el vector que va des del punt de llum fins la gota, i el taronja és el corresponent a la direcció de la llum del punt de llum.

També necessitem saber l'angle de llum del punt de llum, concretament la meitat de l'angle. Només necessitem saber la meitat de l'angle ja que com hem vist a l'anterior figura, l'angle es calcula respecte el vector de direcció del punt de llum, per tant s'ha d'agafar aquest mateix vector com a referència per calcular l'angle interior i exterior.

Un cop s'ha calculat l'angle de la gota respecte el punt de llum, i un cop coneixem els angles interiors i exteriors en els quals hi ha intensitat de llum, si l'angle de la gota respecte el punt de llum és superior a l'angle de la intensitat del punt de llum, voldrà dir que aquesta gota d'aigua no ha de ser il·luminada per aquest punt de llum. Per contra serà il·luminada, després caldrà observar en quina intensitat és il·luminada, si l'angle és inferior a l'angle interior serà il·luminada per la intensitat màxima, altrament per una intensitat entre la màxima i la mínima. Veiem el pseudocodi:

```
Si angle > angleGran
   fragment de gota no il·luminat per aquesta llum
altrament si angle > anglePetit
  Fragment il·luminat amb llum mitja, entre la intensitat de
  llum màxima i una    de inferior a la màxima.
altrament
   Fragment il·luminat per la llum sencera màxima
```

Podem veure el codi de tots els càlculs explicats en el següent fragment de codi:

```
        //calcul de la direcció de la llum
        float3 dirLLumPunt = normalize(posPuntDelShader-
        lightPosition1.xyz);
        float cosDirection= dot(dirLLumPunt,lightDirection1.xyz);
        float angle=acos(cosDirection);
        //calcul angles/2 spotlight
        float innerAngle=acos(spotLightParams1.x);
        float outerAngle=acos(spotLightParams1.y);
        //en aquest cas posarem que el llum spotlight és a la posició 1, si es varies la
        posició cal canviar aquest valor
        if (angle>outerAngle)
        {
                //no iluminat per aquesta llum
                lightDif[1]=float4(0,0,0,1);
         }
        else if(angle>innerAngle)
        {
                //fragment il·luminat amb llum mitja
                lightDif[1]=lightdiffuse1*smoothstep(innerAngle,outerAng
                le,angle);
        }
        else
        {
                //fragment il·luminat per la llum sencera
                lightDif[1]=lightdiffuse1;
        }
```

Com podem veure lightDif[1], guardarà la intensitat de llum que afecta a cada fragment de la gota d'aigua, a diferència de fins ara a cada fragment de la gota d'aigua afectarà d'una manera en particular. Fins ara a tots els fragments de les gotes d'aigua, lightDif[1] tenien el mateix valor cosa que el feia irreal.

Al codi que s'acaba de mostrar s'ha utilitzat la funció smoothstep, aquesta és una funció pròpia del llenguatge Cg. Com s'ha pogut veure, rep tres paràmetres i retorna un real, s'ha utilitzat per calcular la intensitat de llum del fragment que està entre l'angle petit del punt de llum i el gran. Aquest és el pseudocodi d'aquesta funció:

```
        Smoothstep (min,max,x)

        Si (x<min)
              Retorna 0
        Si x>=max
              Retorna 1
        Altrament
        Retorna l'anoemat smooth Hermite interpolation, el qual és un
        valor entre 0 i 1 donat per la següent fórmula:
        -2*((x-min)/(max-min))^3 + 3*((x-min)/(max-min))^2
```

Es pot veure una gràfica de la funció smoothstep a la figura 3.42.

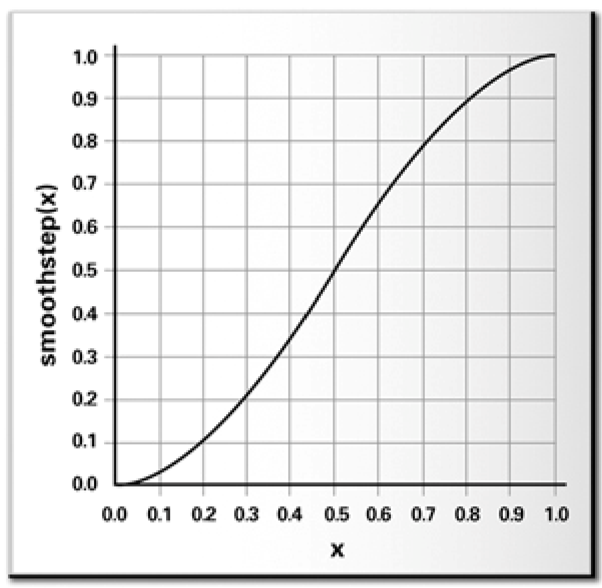


Figura 3.42, gràfica de la funció smoothstep

Per observar si l'algorisme explicat funciona, s'ha substituït momentàniament les gotes d'aigua per quadrats. Els fragments de cada quadrat han de prendre un color o un altre segons les llums que rep. Si només rep la llum general els fragments seran de color negre. Si per contra apart de la llum general el fragment és il·luminat per la llum de l'angle interior generat pel fanal, aquest fragment agafarà el color verd. Si el fragment rep llum del fanal, però és una llum inferior a la central, és a dir l'angle exterior de la llum generada per el fanal, aquest fragment se li assignarà el color vermell. Si no apliquéssim l'algorisme descrit, en aquesta prova tots els quadrats tindrien color verd. Veiem la diferència aplicant l'algorisme a les següents figures:

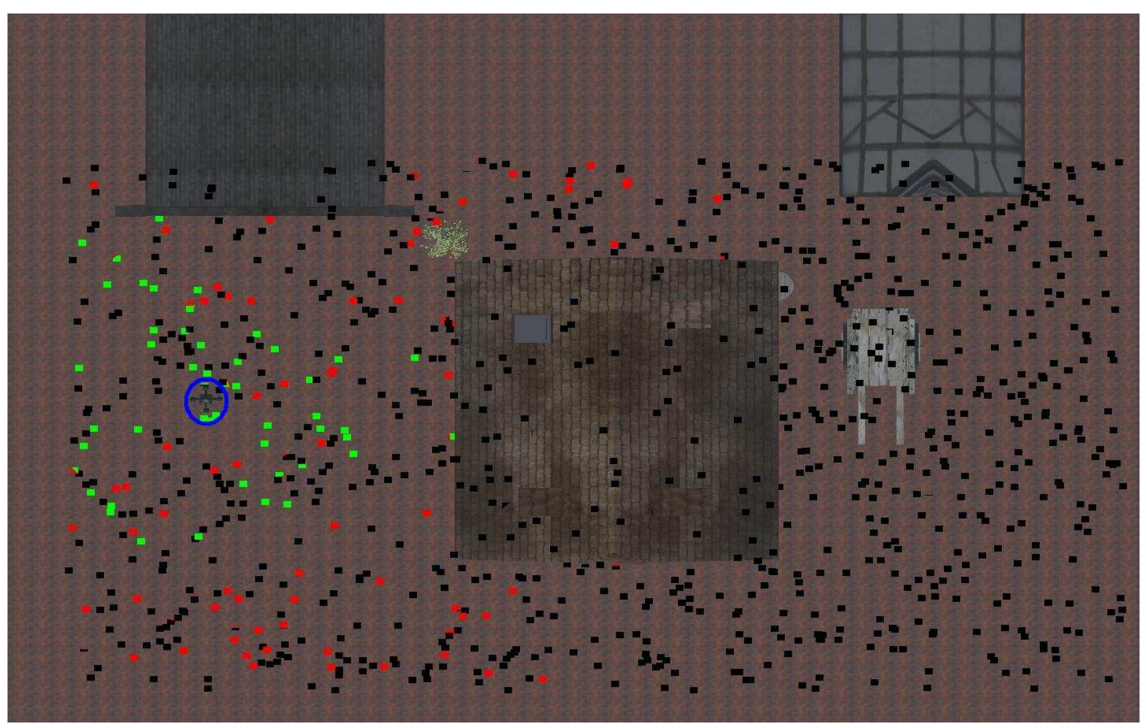


Figura 3.43, imatge aèria on podem observar els diferents colors que s'han assignat als quadrats. A la banda esquerra, veiem dibuixada una circumferència la qual marca la posició del fanal. Com podem observar la majoria dels quadrats pròxims al fanal agafen el color verd, els que estan una mica més allunyats al vermell. Finalment els que estan ja a una distància llunyana han agafat el color negre. Hi ha quadrats pròxims al fanal que han agafat el color negre, cosa que indica que aquests quadrats estan a una alçada més elevada que la del punt de llum.

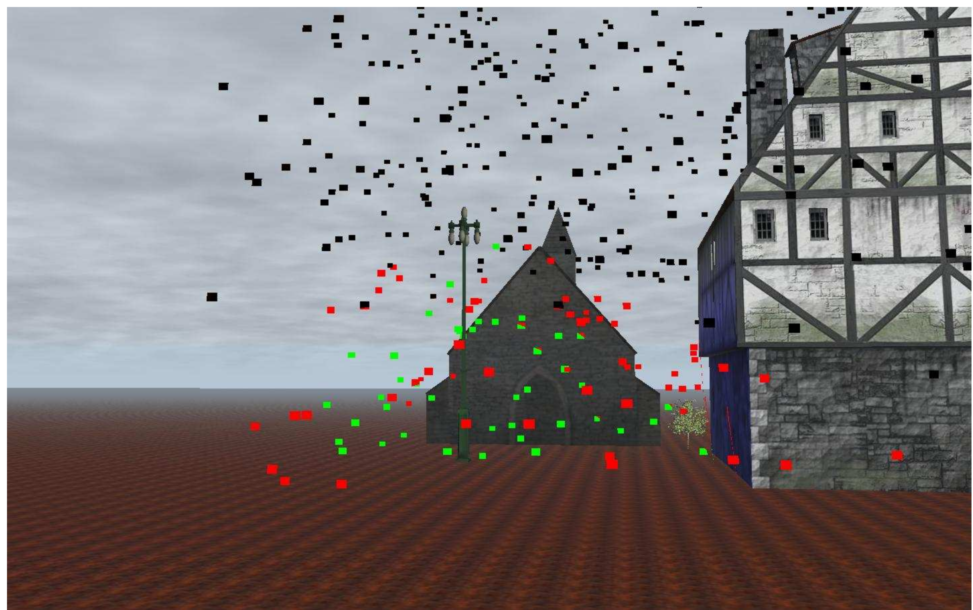


Figura 3.44, imatge en la que podem observar com els quadrats que estan a una alçada superior a la del fanal agafen el color negre, això indica que a aquestes gotes no rebrien la llum generada pel fanal. Podem observar els quadrats verds i vermells, els quals ens informen de les diferents intensitats en que la llum arriba a les gotes.

Un cop s'ha comprovat que l'algorisme funciona correctament, s'ha substituït de nou els quadrats per les gotes d'aigua. A continuació podem observar els resultats:

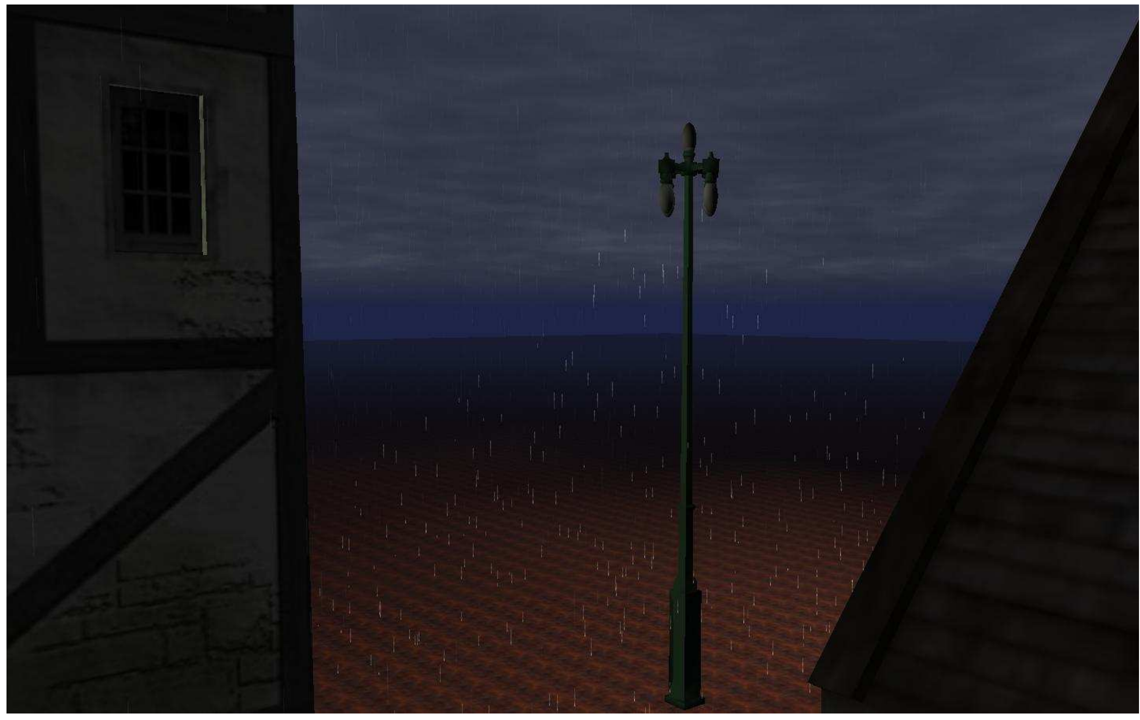
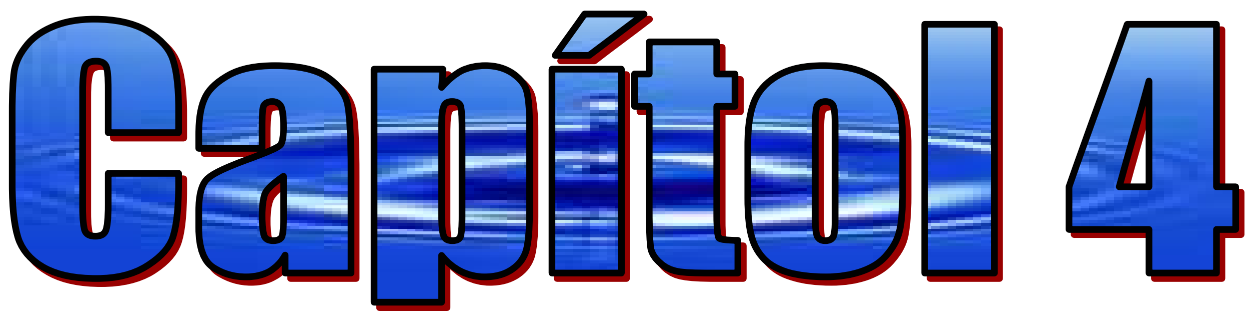


Figura 3.45, imatge de l'aplicació en funcionament, on es pot observar que les gotes més pròximes al fanal són més visibles a causa que el fanal les il·lumina.

Finalment veiem una imatge de l'aplicació en funcionament amb les gotes d'aigua amb la seva forma correcta.

Figura 3.45, imatge de l'aplicació en funcionament.



-Resultats i conclusions-

4.1 Resultats

Com ja s'ha dit al començament de la present documentació (*apartat 1.3 Abast*), l'aplicació realitzada es divideix en dos sistemes, un editor de pluja, i un sistema de visualització en temps real:

- **Editor de pluja:** Ja que un dels objectius del projecte era implementar una aplicació que permetés al sistema de visualització en temps real, ploure en les zones seleccionades per l'usuari, aquest editor s'ha implementat buscant sempre que l'usuari aconseguís fer ploure exactament a les zones que ell desitgés. Per tant, per buscar aquesta exactitud, s'ha hagut de buscar sempre algorismes que permetessin obtenir el resultat més exacte possible. És el cas per exemple de l'algorisme Cumulative Density Function (CDF)(*veure apartat 3.1.3.2 Calcular les zones de pluja*), que ens ha permès distribuir les gotes de la manera desitjada. Sempre recordant que també s'ha tingut de buscar una nova manera de generar números aleatoris perquè aquest algorisme fos realment eficaç (llibreria Mersenne Twister). Perquè la funció de generar números aleatoris que porta el visual c++ es va comprovar que tenia insuficient precisió.

  Aquí veiem algunes imatges de l'editor en funcionament:

Figura 5.1, Podem veure l'editor que ha carregat una imatge. En aquest cas és la imatge de l'escenari vist des del cel.

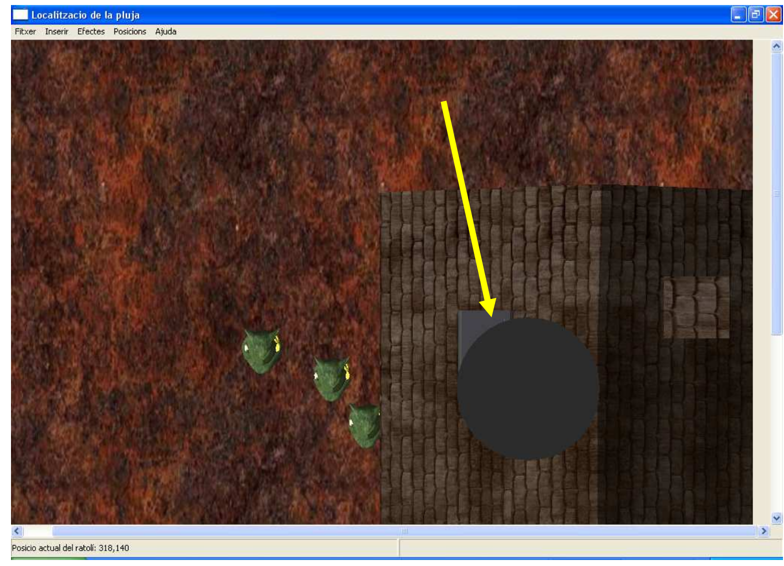


Figura 5.2, veiem que l'usuari vol que plogui sobre la teulada, ha començat dibuixant un cercle sobre la imatge.

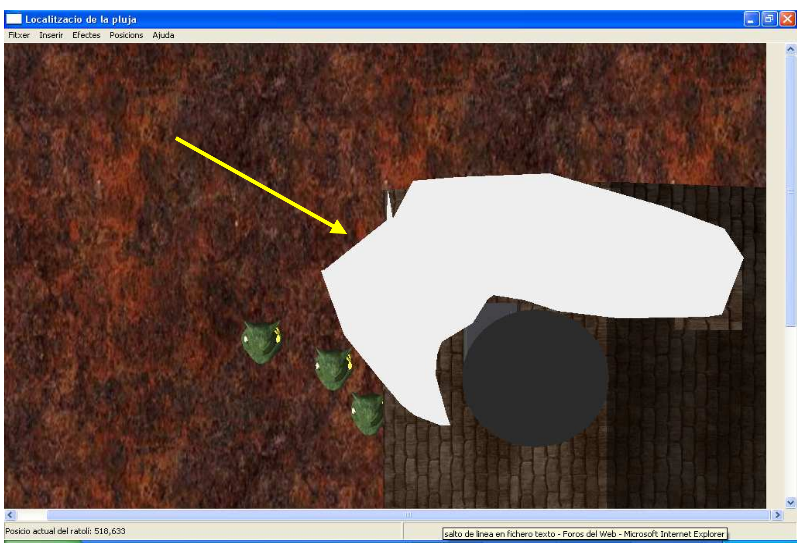


Figura 5.2, La Figura mostra que l'usuari desitja ampliar la zona de pluja, però en aquest cas de forma més suau.

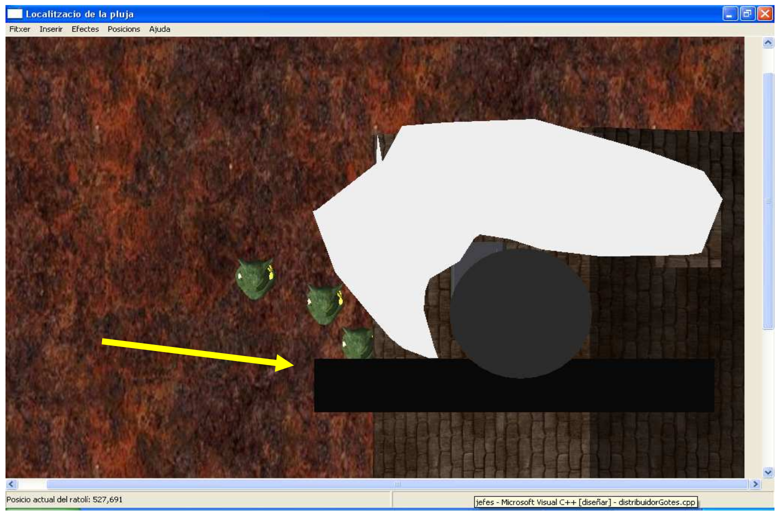


Figura 5.3, veiem que l'usuari ha dibuixat un rectangle amb un color molt proper al negre. Serà indicador de que vol que en aquesta zona plogui amb una gran intensitat.

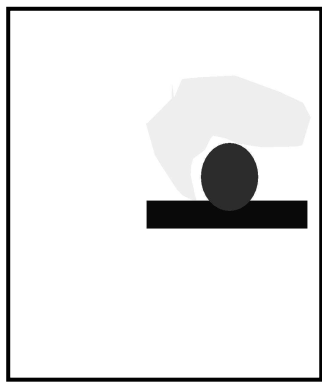


Figura 5.4, imatge del total de figures generades per l'usuari.

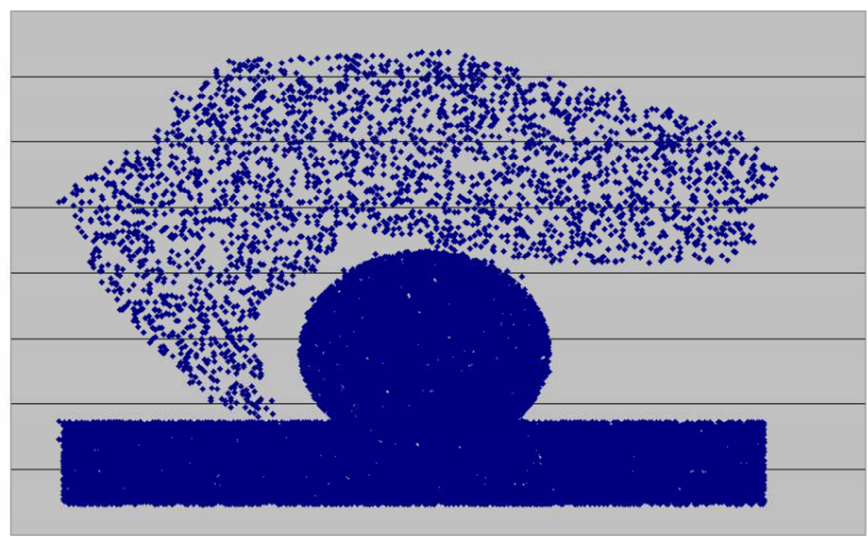


Figura 5.5, Aquesta seria la distribució de 50000 gotes d'aigua que hauria generat l'editor, podem comprovar que s'han distribuït de la forma desitjada.

Figura 5.6, I aquesta seria la imatge de la pluja queient des de l'aplicació en 3D, veiem que la cauen exactament a la posició desitjada.

Per buscar més realitat hi ha la possibilitat de suavitzar els canvis d'intensitat de pluja. Per això s'ha creat la difuminació de la zona de pluja (veure apartat *3.1.3.1 Difuminar les zones de pluja*). Veiem aquí el resultat d'aplicar la difuminació sobre les figures que s'han estat veient fins ara en aquest apartat:

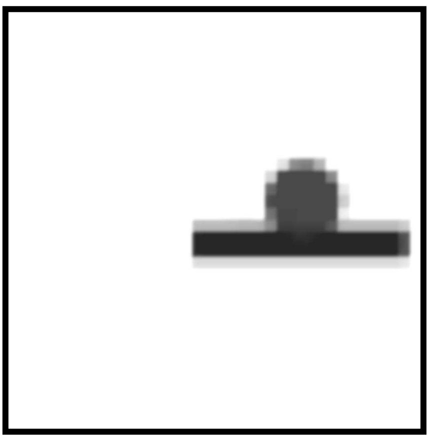


Figura 5.7, resultat d'aplicar la difuminació de la imatge.

Com s'ha pogut veure a l'apartat 3.1.4 Generació d'atles, l'editor permet generar els atles tant d'imatges de llum ambiental com de llums puntuals, aquests s'utilitzaran a l'aplicació en temps real per a la visualització de la pluja. Podem observar l'atles a la figura 5.8:

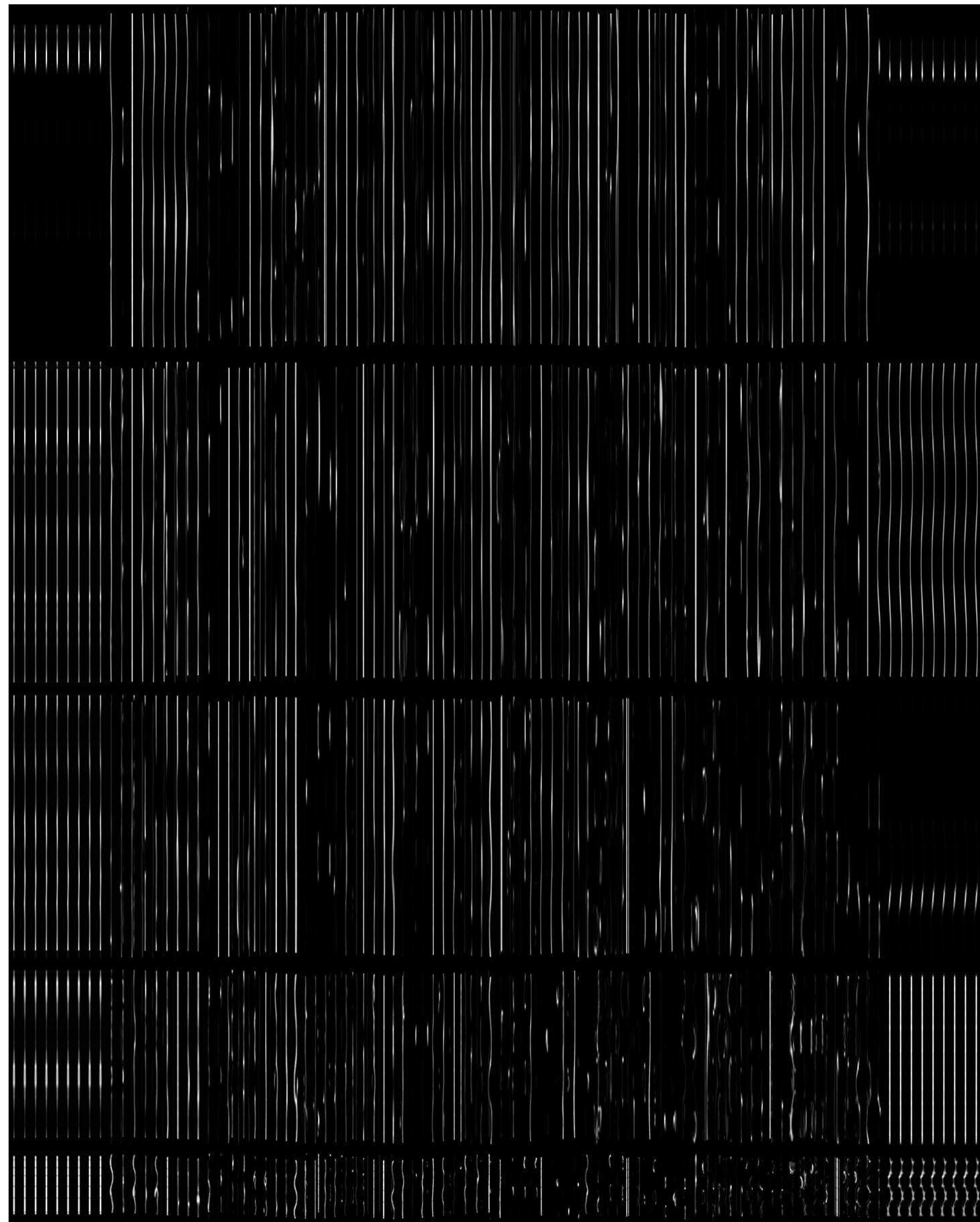


Figura 5.8, atles de gotes de pluja de llums puntuals.

- **Visualització en temps real**: Aprofitant l'exactitud a l'hora de distribuir les gotes d'aigua per les zones dibuixades per l'usuari amb l'editor desenvolupat, a l'aplicació en temps real. S'ha implementat un sistema de pluja que compleix els fonaments físics de la pluja *(veure apartat 2.3 Física de la pluja).* S'ha aconseguit capturar la imatge de l'escenari d'una forma ràpida i eficaç. L'aplicació permet aconseguir un mapa de profunditats de l'escena amb una gran exactitud. Això s'ha aprofitat per

impedir que les gotes d'aigua impactin amb els diferents objectes i així fer encara més real la caiguda. Podem veure els resultats amb imatges:

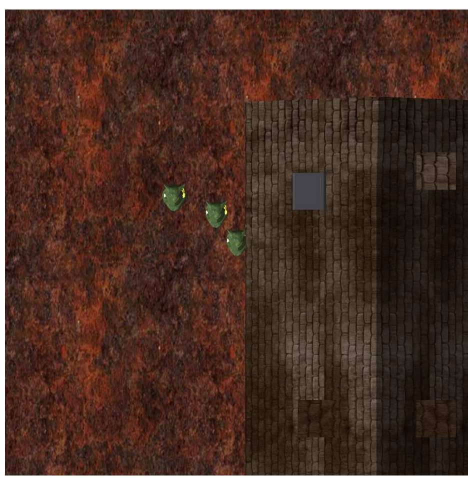


Figura 5.9, imatge capturada des del cel.

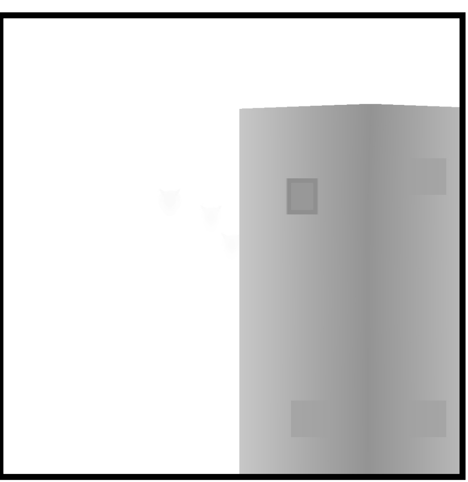


Figura 5.10, mapa de profunditat de l'escena capturada.

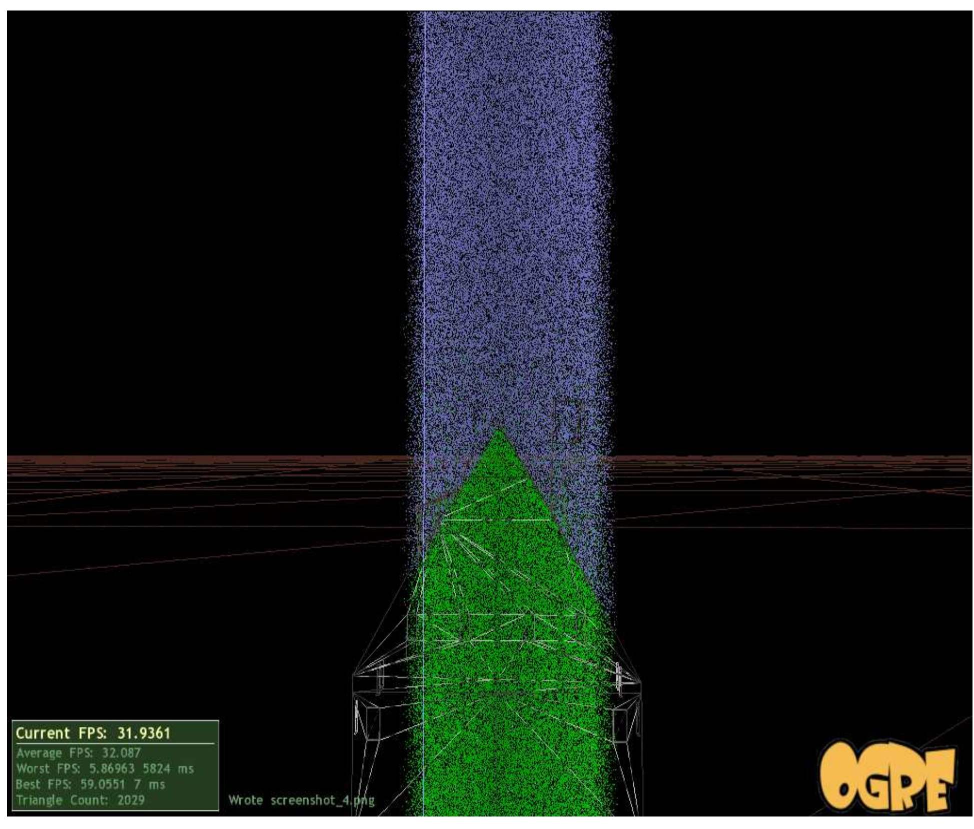


Figura 5.11, Resultat de les proves realitzades per observar més clarament l'eficiència de l'impacte de les gotes amb els obstacles. Com es pot comprovar, les gotes al impactar amb un obstacle canviaven de blau a verd, hem fet aquest canvi per així observar més clarament el resultat.

Figura 5.12, figura que mostra que les gotes no cauen més enllà de la superfície.

L'aplicació en temps real gràcies als shaders, els quals s'ajuden de l'atles on hi ha guardades les diferents textures de gotes de pluja, aconsegueix donar realisme a la visualització de la pluja. Es pot observar a les següents captures:

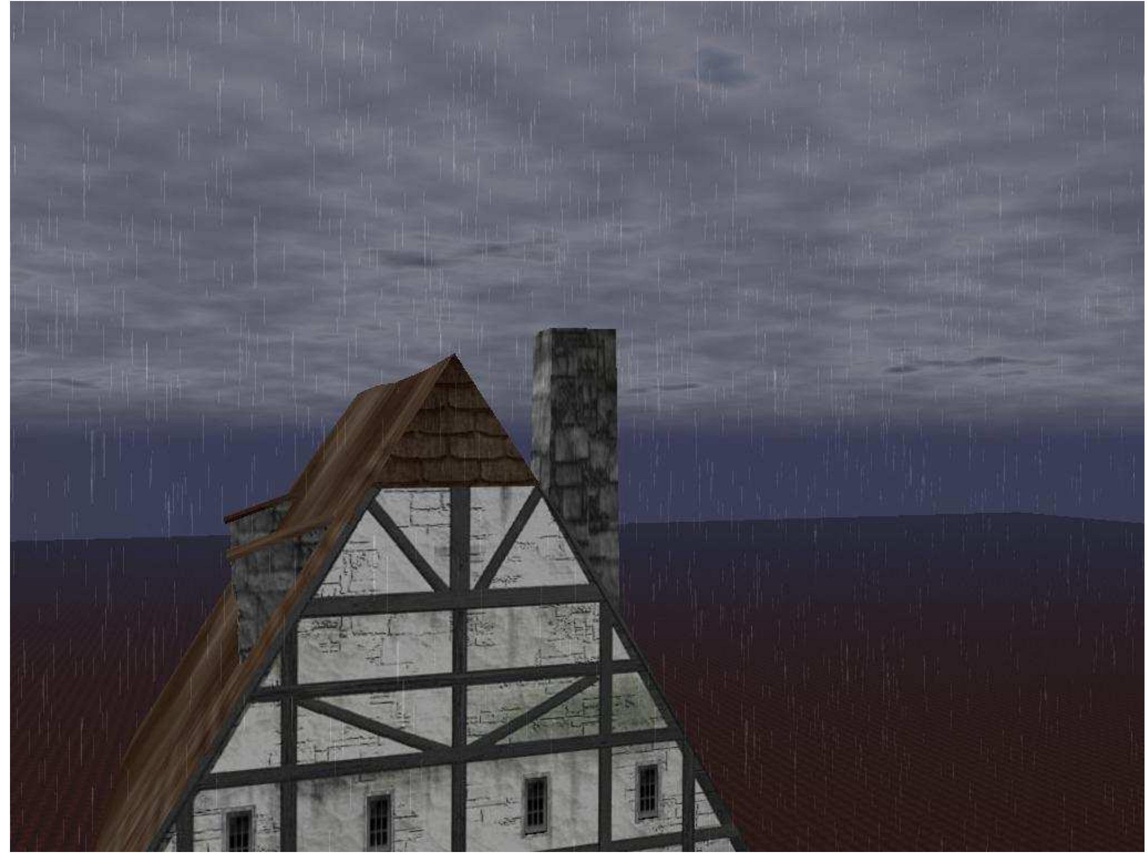


Figura 5.13, figura en la que podem observar el resultat final. Observem una gran quantitat de gotes vistes des de la distància

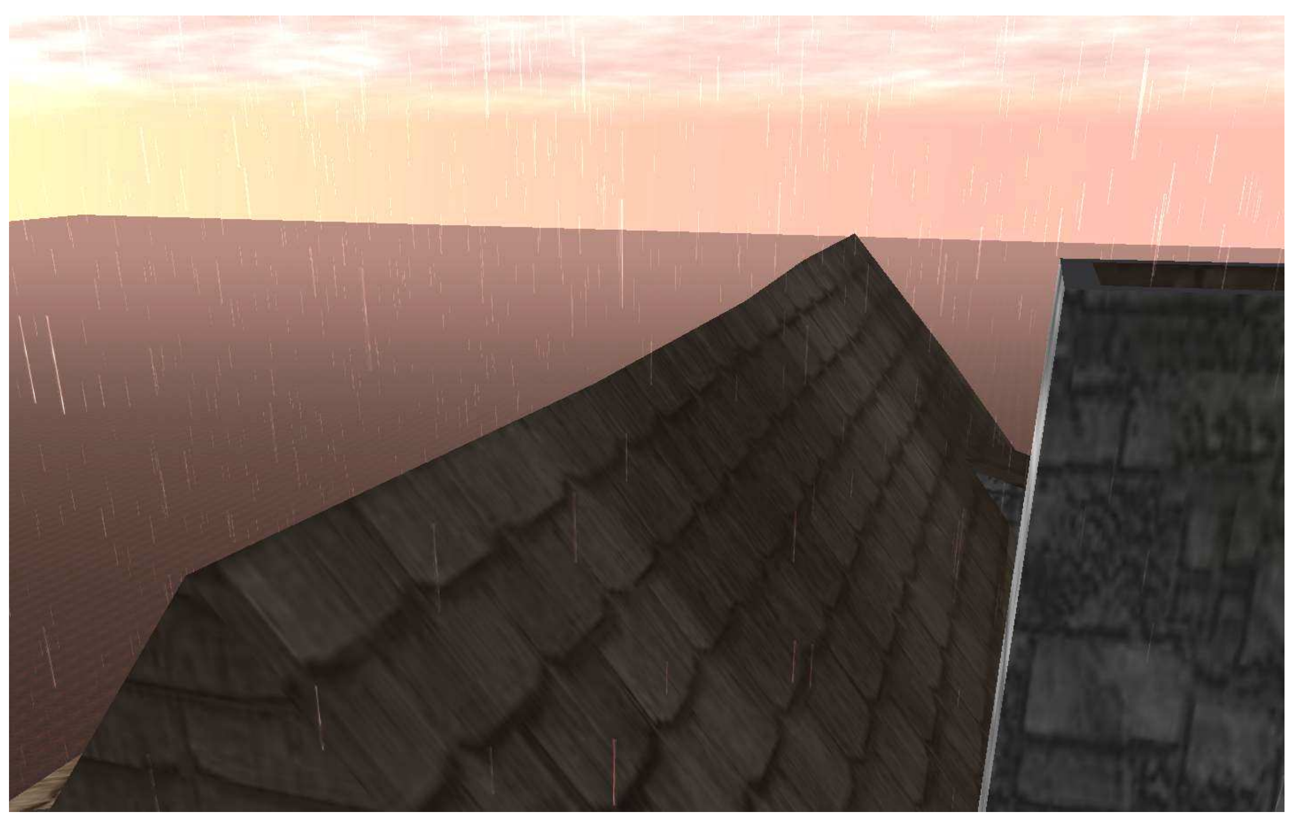


Figura 5.14, observem la visualització de la pluja quan el sol es pon a l'escenari.

Figura 5.15, observem com plou en un escenari molt ennuvolat.

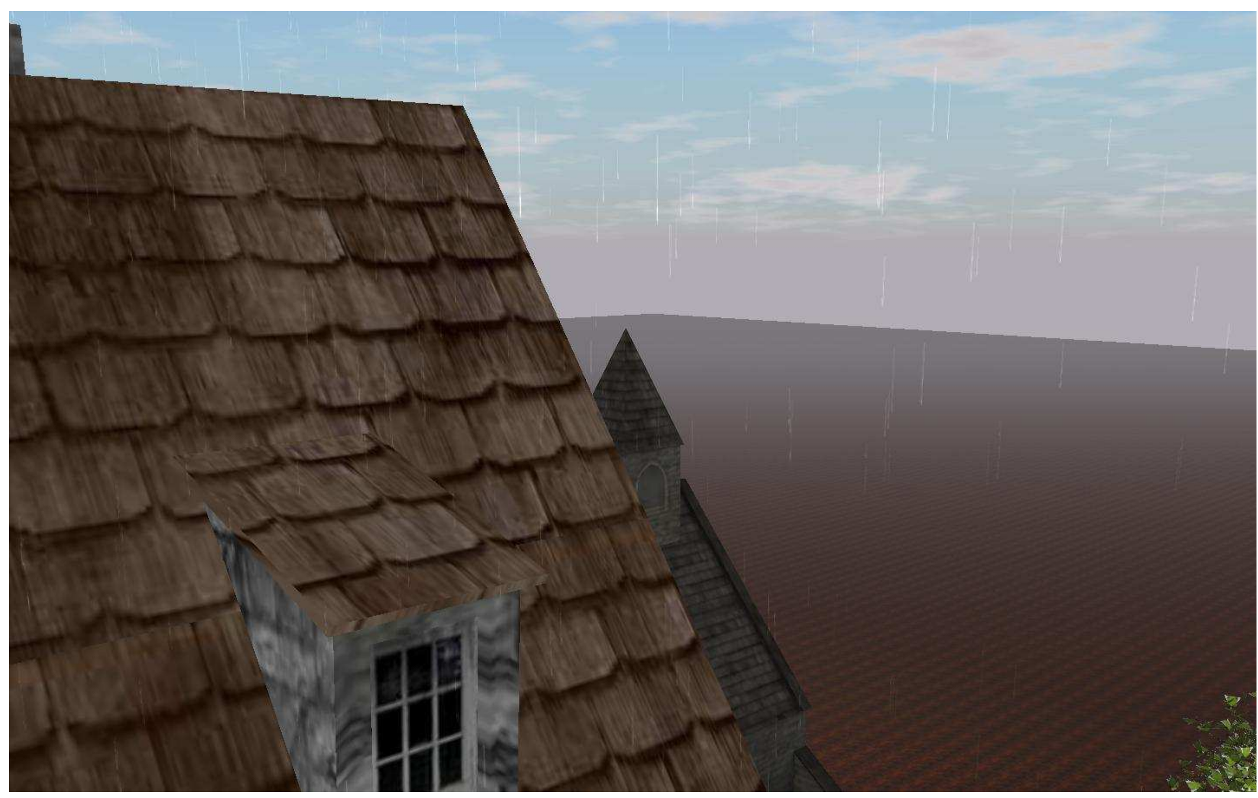


Figura 5.16, observem la pluja amb un cel aclarat.

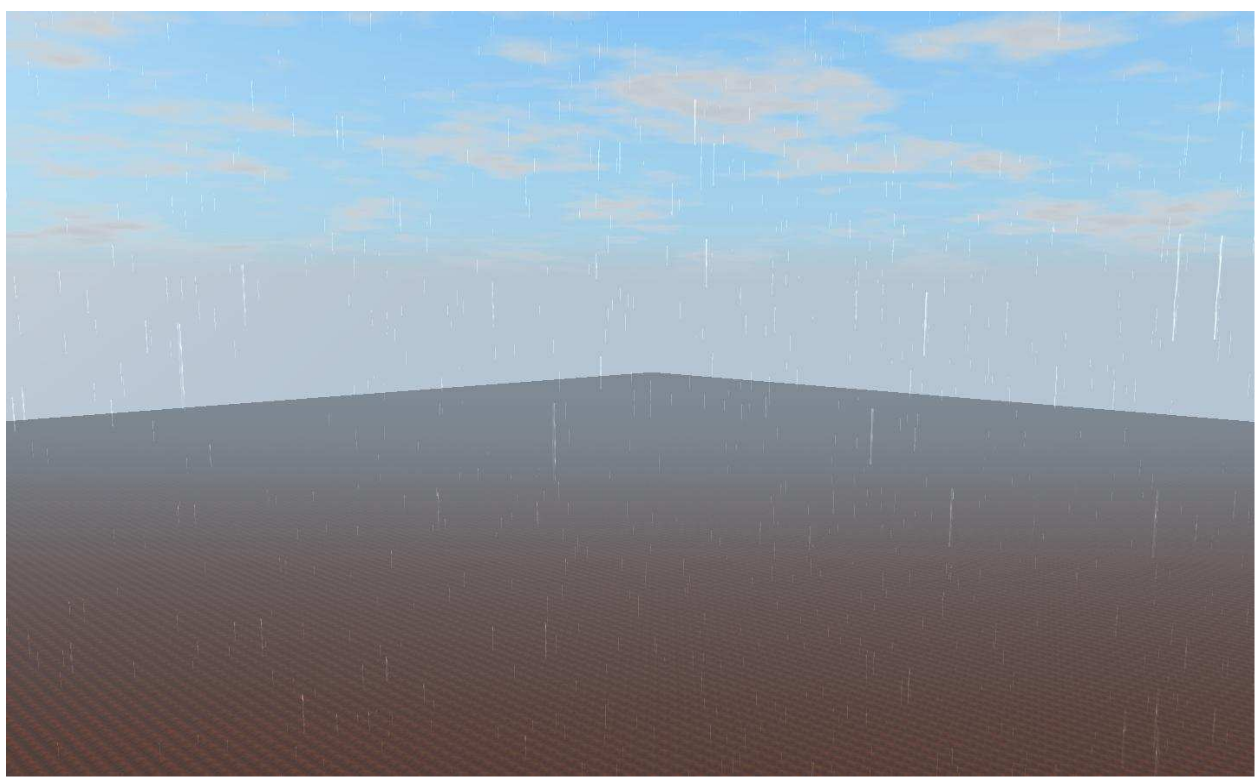


Figura 5.17, observem la pluja amb el cel aclarat.

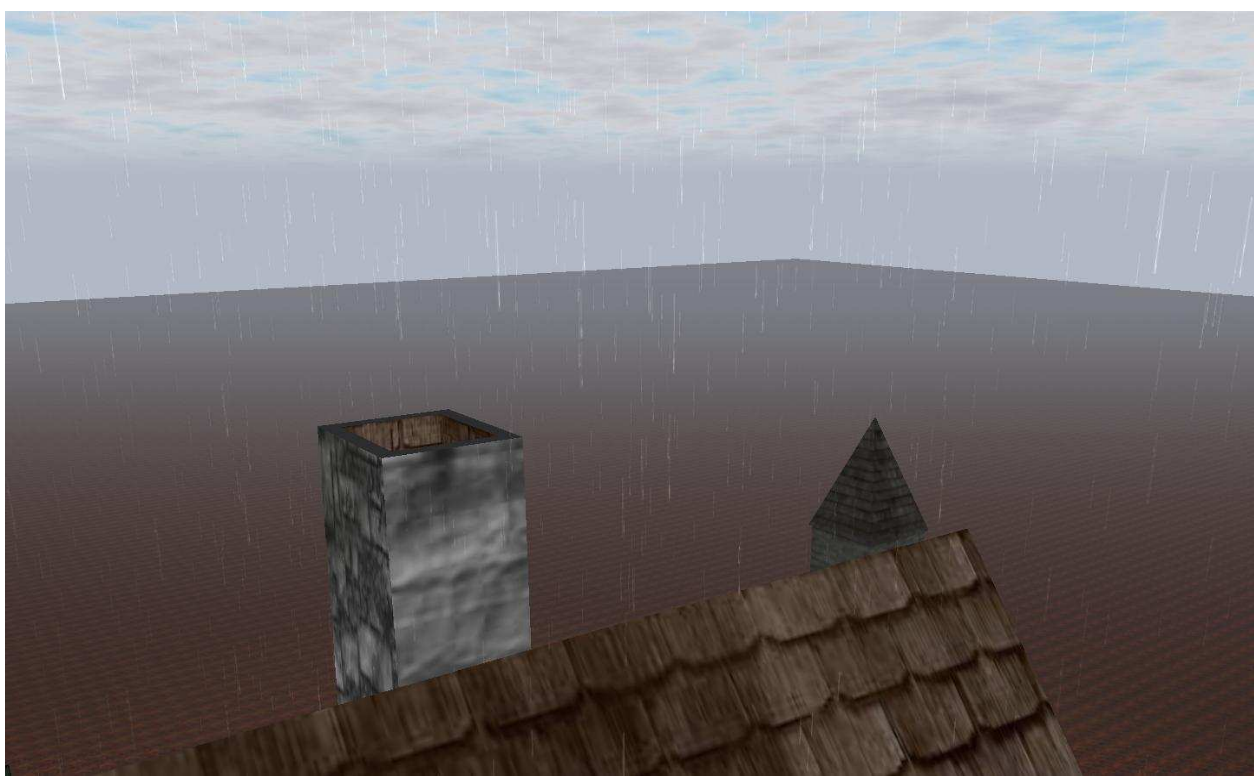


Figura 5.18, observem com plou en un escenari amb alguns núvols.

Com s'ha pogut anar veient durant la documentació, l'aplicació en temps reals visualitza les gotes d'aigua depenent dels diferents punts de llum que tingui l'escenari. Si és un llum de tipus spotlight, és a dir un fanal, o uns llums d'un vehicle etc, aquesta llum no ha d'afectar a la resta de gotes de l'escenari, només a les gotes que passin per aquest flux de llum. A continuació es poden observar algunes imatges en que es mostra com el fanal de l'escenari il·lumina únicament les gotes del seu voltant, cosa que produeix que ressalti aquelles gotes en comparació amb la resta.

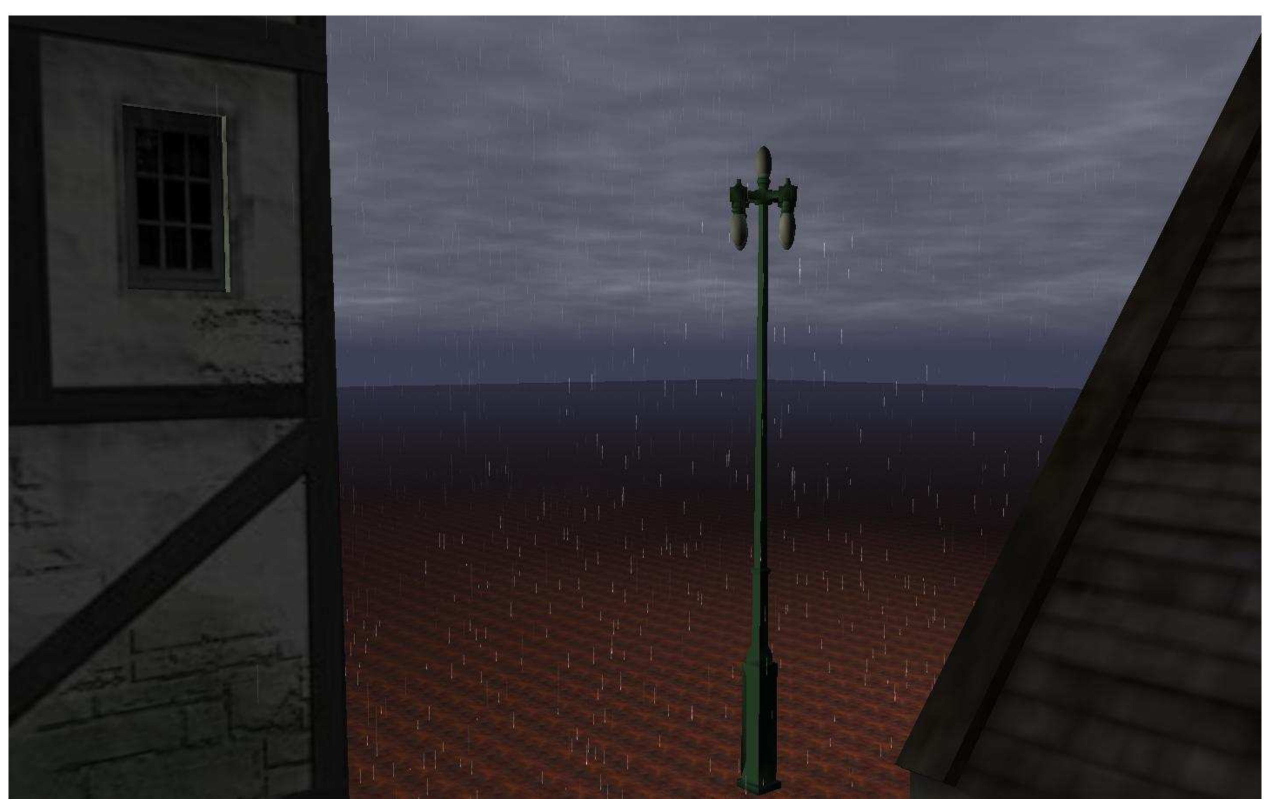


Figura 5.19, com es pot veure a la imatge a partir del fanal les gotes estan més il·luminades que la resta.

Figura 5.20, la imatge mostra com les gotes són més blanquinoses gràcies a l'aportació de la llum del fanal.

Figura 5.21, la imatge mostra com les gotes inferiors al fanal són més blanques a conseqüència del fanal que la resta, en aquest moment generava una llum blanca.

Figura 5.22, la imatge mostra el fanal com en aquesta ocasió genera una llum blava, es pot observar com les gotes il·luminades han agafat un color blau.

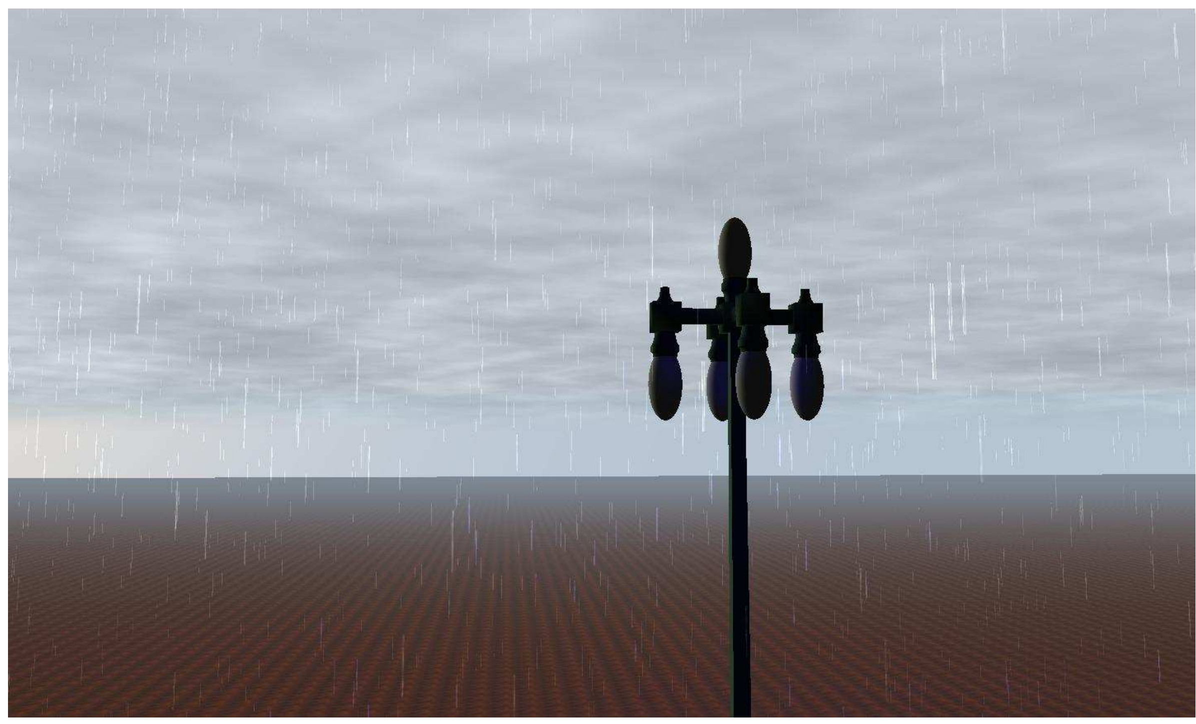


Figura 5.23, a la imatge podem observar com plou, les gotes inferiors al fanal són més blavos que la resta ja que el fanal genera una llum blava.

4.2 Conclusions

Al llarg de la realització d'aquest projecte, s'ha buscat que la distribució de pluja fos la que desitgés i dibuixés l'usuari a l'editor. Creiem que, veient els resultats, s'ha aconseguit aquest objectiu, ja que l'editor permet:

- Carregar i guardar els formats més usats que hi ha actualment, és a dir, els formats jpg, gif, bmp, png, tiff etc. Això permet que l'usuari tingui la possibilitat de carregar un gran repertori d'imatges siguin del format que siguin.

- Amb les figures cercle, rectangle i polígon que pot generar l'editor, l'usuari aconsegueix omplir amb figures tot l'escenari carregat. Aquestes figures es podran dibuixar amb 255 intensitats diferents de pluja, cosa que fa que l'editor sigui una eina que permeti generar zones de pluja amb gran detall.

- S'ha aconseguit fer caure les gotes d'aigua a la posició escollida, respectant la distribució de pluja segons les intensitats marcades per l'usuari amb exactitud.

- S'ha aconseguit augmentar el realisme dels actuals sistemes de pluja (com per exemple el que disposa l'Ogre3D), ja que per exemple les diferents intensitats de pluja varien suaument i per tant fa menor el contrast entre la zona en què plou i la que no.

- S'ha permès generar un atles d'imatges per englobar les diferents textures de la base de dades, les quals poden prendre les diferents gotes de pluja.

Pel que fa als objectius marcats abans de realitzar el sistema de pluja, també podem dir que s'han complert en escreix, ja que s'ha aconseguit un sistema de pluja en temps real molt realista, ja que aquest:

- Compleix els fonaments físics de la pluja i per tant s'aconsegueix un realisme elevat en quant al comportament de les gotes.
- Ha estat desenvolupat aprofitant el hardware gràfic, cosa que fa que les CPUs es lliurin d'executar algorismes costosos.
- A diferència de la majoria de sistemes de pluja, contempla la profunditat de l'escenari, per tant no travessarà objectes (siguin teulades, taules o senzillament persones).
- S'ha aconseguit una visualització realista de les gotes d'aigua les quals tindran una visualització diferent depenent de les diferents propietats dels punts de llum de l'escena i de la posició de l'observador.

Per tant s'ha aconseguit implementar un simulador de pluja. Que permetrà als desenvolupadors de videojocs escollir la zona que volen què caigui pluja en

temps real i de forma realista. A més a més, la visualització s'ha desenvolupat a partir d'una base de dades realista d'imatges de gotes d'aigua, amb la qual cosa s'ha aconseguit maximitzar el realisme. S'han afegit efectes de moviment i il·luminació realista, aconseguint un alt grau de fotorrealisme en el resultat final.

Capítol 5

-Treballs futurs-

A continuació es presentaran certes millores que es tenen pensades realitzar en un futur:

- Distribuir la pluja al voltant de l'observador. Actualment es distribueix per tota la zona seleccionada per l'usuari, en canvi una millora podria ser distribuir la pluja únicament per una zona pròxima a l'observador.

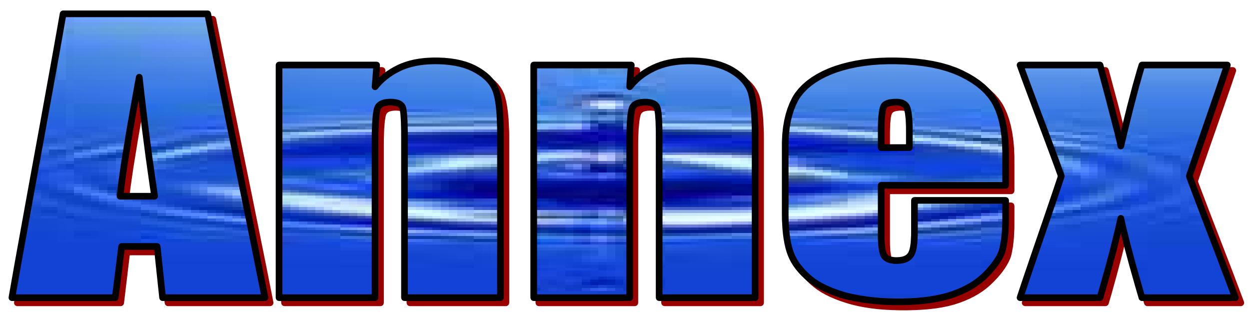
- Generació dels billboards a la GPU utilitzant Geometry Shadows: reben 1 vèrtex per cada gota i generen un billboard. Per tant passaríem de tenir un vèrtex shader i un píxel shader a tenir:
    - Geometry shaders: generen billboards per cada gota.
    - Vèrtex shader: posiciona gota.
    - Píxel shader: donen aparença visual.

- Un altre treball que es té previst fer en el futur, que té una relació directa amb el desenvolupat en aquest projecte de final de carrera, és el d'afegir el comportament del sistema de pluja que s'ha creat en una situació de vent.

Bibliografia

Aquí podem observar la bibliografia que s'ha necessitat per desenvolupar el present projecte:

- Randima Fernando, Mark J. Kilgard; "*Cg Tutorial, The: The Definitive Guide to Programmable Real-Time Graphics*", Addison Wesley Professional 1era edició, febrer del 2003.
- Woo, Mason & Neider, Jackie & Davis, Tom; "*OpenGL Programming Guide*"; Second Edition; Addison-Wesley, July 1997.
- Kempf, Retane & Frazier, Chris; "*OpenGL Reference Manual*"; Second Edition; Addison-Wesley, December 1996.
- J. Cros, J. A. Roca; *"3D Studio - Animador Pro - Animador Studio"*; Edición de imágenes Multimedia"; Primera edició; Inforbook's ediciones;1998.
- Bjarne Stroustrup; *"The C++ Programming Language"* ; Third Edition; Addison-Wesley;
- George Shepherd; "*Programming with Microsoft® Visual C++® .NET";* Sisena Edició; Editorial McGraw-Hill
- Gregory Junker; *"Pro OGRE 3D Programming"*; Primera Edició; Editorial Apress; 2006.
- Manual del motor en 3D Ogre3D; Manual d'internet; Accessible des de www.ogre3d.org.
- Manual de la llibreria wxWidgets; Manual d'internet; Accessible des de www.wxwidgets.org.
- Manual de la lliberia Devil; Manual d'internet; Accesible des de http://openil.sourceforge.net/.
- Kshitiz Garg and Shree K. Nayar. Photorealistic Rendering of Rain Streaks, SIGGRAPH 2006.
- LOMAS, A. 2005. The Matrix Revolutions. Personal Communication.
- REEVES, W. T. 1983. Particle System- a Techinque for Modeling a Class of Fuzzy Objects. ACM Trans. Graphics, 91–108.
- STARIK, K., AND WERMAN, M. 2003. Simulation of Rain in Videos. Texture Workshop, ICCV.
- WANG, N., AND WADE, B. 2004. Rendering Falling Rain and Snow. SIGGRAPH (sketches 0186).


Annex

En el següent apartat es podrà observar tot el codi implementat als shaders.

Codi de la funció calculaLongImatge utilitzada al vèrtex shader:

```
void calculaLongImatge(out int longI,out int longF,out int pC,float
aCamera,float alLlum,float phi)
{
//ara que ja tenim els angles, hem de saber per cada gota, quin és l'angle que li
correspon, per fer això hem de comprobar, en quin dels possibles angles hi ha
menys diferències amb el que ha sortit.
	int pA=-90,pPhi=10,iA=-1,iPhi=-1;
	int iC=-1,millorPa=10,millorPc=0,millorPphi=10;
	pC=0;
	float rC,rA,rPhi,millorC=200,millorA=200,millorPhi=200;
	bool trobatC=false,trobatA=false;
	bool trobatPhi=false,acabat=false;
//per millorar amb velocitat,només es crea un bucle, ja que si se'n un per cada
angle, com seria més llarg, aniria més lent, per tant es fa un bucle, i en aquest es
busca, per cada angle, quin li pertoca. Com el que hi ha més angles possibles és el
de l'alçada de la llum el mentre, es farà fins que no s'hagi examinat tots els angles,
o fins que no haguem trobat tots els angles, que això serà quan haguem trobat, per
cada angle, una diferència que sapiguem que ja no n'hi haurà cap de més petita,
que és <10.

	while (pA<91 && !acabat)
	{
//si de cada angle no hem trobat el que li toca, augmentem la seva i.
		if (!trobatC)
			iC++;
		if (!trobatA)
			iA++;
		if (!trobatPhi)
			iPhi++;

//calculem la diferencia amb un angle possible, marcat per pX.
		rC=abs(aCamera-pC);
		rA=abs(alLlum-pA);
		rPhi=abs(phi-pPhi);

//comprobem si la diferencia que acabem de trobar, és inferior o no,  a la diferència
més petita que ja es tenia, això es farà únicament si no s'ha trobat ja l'angle que li
pertoca.
		if (rC<millorC && !trobatC)
		{
			millorC=rC;millorPc=pC;
		}
		if (rA<millorA && !trobatA)
		{
			millorA=rA;millorPa=pA;
		}
		if (rPhi<millorPhi && !trobatPhi)
		{
			millorPhi=rPhi;millorPphi=pPhi;
		}
...
```

```
…
//posarem a true que s'ha trobat l'angle si el resultat de millor és inferior a 10
            if (millorC<=10)
                  trobatC=true;
            if (millorA<=10)
                  trobatA=true;
            if (millorPhi<=10)
                  trobatPhi=true;
//haurem acabat si hem trobat els 3 angles.
            if ((trobatC && trobatA && trobatPhi))
                  acabat=true;
//augmentem l'angle de la camera
            if (pC<81 && !trobatC)
                  pC=pC+20;
//i el de phi, finalment el de l'alçada de la camera, en aquell no cal comprobà si s'ha trobat o no, ja que ja es fa al while.
            if (pPhi<171 && !trobatPhi)
                  pPhi=pPhi+20;

            pA=pA+20;
      }
//com les imatges totes tenen la mateixa amplada, podem trobar la posició que pertoca fent les multiplicacions amb iA i iPhi, que les hem trobat al bucle
      longI=(144*iA)+(16*iPhi);
      longF=longI+16;
}
```

Codi de la funció calculaLongImatgeB utilitzada al vèrtex shader:

```
void calculaLongImatgeB(out int longI,out int longF,float
aCamera,float alLlum,float phi)
{
      int pA=-90,pPhi=10,iA=-1,iPhi=-1,iC=-1
      int millorPa=10,millorPc=0,millorPphi=10;
      float rA,rPhi,millorA=200,millorPhi=200;
      bool trobatA=false,trobatPhi=false,acabat=false;
      while (pA<91 && !acabat)
      {
            if (!trobatA)
                  iA++;
            if (!trobatPhi)
                  iPhi++;

            rA=abs(alLlum-pA);
            rPhi=abs(phi-pPhi);

            if (rA<millorA && !trobatA)
            {
                  millorA=rA;millorPa=pA;
            }
            if (rPhi<millorPhi && !trobatPhi)
            {
                  millorPhi=rPhi;millorPphi=pPhi;
            }
            if (millorA<=10)
                  trobatA=true;
...
```

```
            if (millorPhi<=10)
                  trobatPhi=true;
            if ((trobatA && trobatPhi))
                  acabat=true;
            if (pPhi<171 && !trobatPhi)
                  pPhi=pPhi+20;

            pA=pA+20;
      }
      longI=(144*iA)+(16*iPhi);
      longF=longI+16;
}
```

Codi del vèrtex shader:

```
void main_vp(float4 pInitial : POSITION,
                 out float2  uvB[2] : TEXCOORD0,
                 out float2  uvC : TEXCOORD5,
                 out float3 oWorldPos : TEXCOORD4,
                out float4 oPosition : POSITION,
                out float4 color     : COLOR,
               float4 c: COLOR,
           uniform sampler2D texture : register(s0),
           uniform float    time,
           uniform float4 camera,
           uniform float4 lightPosition0,
           uniform float4 lightPosition1,
           uniform float4x4 modelViewProj,
           uniform float4x4 modelWorld,
            uniform float4 normalitzarX,
            uniform float4 normalitzarY,
            uniform float altura
           )
{
float4 lightPos[2];
lightPos[0]=lightPosition0;
lightPos[1]=lightPosition1;
float2 uv=float2(((normalitzarX[0]-pInitial.x)/(normalitzarX[0]-
normalitzarX[2])),((normalitzarX[1]-pInitial.z)/(normalitzarX[1]-
normalitzarY[2])));

float profunditat=(tex2D(texture,uv).x-1)*(altura*-1);
int superior=0;
float alsadaInicial=c[2]*2*10.0;//d'aqui treiem l'altura inicial del billboard.
 //si es compleix la condició voldrà dir que és una de les 2 cantonades de la part de
dalt
 if (c[0]>0.5)
 {
      superior=1;
      c[2]=c[2]*(-1);
 }
  pInitial.y= pInitial.y - 0.5 * time ;
  if (pInitial.y-(14*superior)<0)
      pInitial.y+=2500;
  if (pInitial.y-(superior*14)<profunditat){
      pInitial.y= float(-2.0);//li assigno aquesta posició
      }
...
```

```
...
//calcul posicio centre
if (c[0]>0.78)//vertex dalt esquerra
{
//distancia x negativa i z positiva
	if (c[0]>0.83 && c[0]<0.87)
		c[1]=c[1]*(-1);
	//distancia x positiva i z negativa
	else if (c[0]>0.88 && c[0]<0.92)
		c[3]=c[3]*(-1);
	//distancia x negativa i z negativa
	else if (c[0]>0.93)
	{
		c[1]=c[1]*(-1);
		c[3]=c[3]*(-1);
	}
	//els altres casos volen dir que les distàncies x i z són positives
}
else if (c[0]>0.58)//vertex dalt dreta
{
	if (c[0]>0.63 && c[0]<0.67)
		c[1]=c[1]*(-1);
	else if (c[0]>0.68 && c[0]<0.72)
		c[3]=c[3]*(-1);
	else if (c[0]>0.73)
	{
		c[1]=c[1]*(-1);
		c[3]=c[3]*(-1);
	}
}
else if (c[0]>0.28)//vertex baix esquerra
{
	if (c[0]>0.33 && c[0]<0.37)
		c[1]=c[1]*(-1);
	else if (c[0]>0.38 && c[0]<0.42)
		c[3]=c[3]*(-1);
	else if (c[0]>0.43)
	{
		c[1]=c[1]*(-1);
		c[3]=c[3]*(-1);
	}
}
else //vertex baix dreta
{
	if (c[0]>0.03 && c[0]<0.07)
		c[1]=c[1]*(-1);
	else if (c[0]>0.08 && c[0]<0.12)
		c[3]=c[3]*(-1);
	else if (c[0]>0.13)
	{
		c[1]=c[1]*(-1);
		c[3]=c[3]*(-1);
	}
}
int pC,longI,longF,alsadaI=0,alsadaF=525;
float incrementGotes=0;
float3 pos=pInitial.xyz;
…
```

```
...
float3 center=pInitial.xyz;
//Calculo el centre del billboard, que és C=V+D, on V el coneixem, ja que és el punt
del VS, i D esta guardat per els parametres del color d'entrada c[1]=x, c[2]=y i
c[3]=z. D=distancia entre centre i vertex calculat a la CPU, genvertices.
//Ja que al billboardSet, es divideix per 10, ja que poden surtir expresions de lestil
de 9.01..i al passar-ho al vertex el color es entre 0 i 1
center.x=c[1]*10.0+pInitial.x;
center.y=c[2]*10.0+pInitial.y;
center.z=c[3]*10.0+pInitial.z;
for (int numLlums=0;numLlums<2;numLlums++)
{
	//calcul de phi
	float3 CamPos=camera.xyz-center;
	float3 L=lightPos[numLlums].xyz;
	L.y=0;
	float3 V = CamPos;
	V.y=0;

	float mV = sqrt(dot(V,V));
	float mL = sqrt(dot(L,L));
	float mlightPosition =
sqrt(dot(lightPos[numLlums],lightPos[numLlums]));
	float mCamPos = sqrt(dot(CamPos,CamPos));

	float phi=acos(dot(V,L)/(mV*mL));

	phi=(phi*180)/3.14159265358979;
	//calcular de angle alçade de la llum
	float alLlum =
acos(dot(lightPos[numLlums].xyz,L)/(mlightPosition*mL));
	alLlum=(alLlum*180)/3.14159265358979;

	//calcular angle alçada camera (cal fer valor absolut)
	float aCamera = abs(acos(dot(CamPos,V)/(mV*mCamPos)));
	aCamera=(aCamera*180)/3.14159265358979;
	//busco la coordenada que li pertoca
	if (numLlums==0){
		calculaLongImatge(longI,longF,pC,aCamera,alLlum,phi);

//segons el valor que hagi quedat a pC (el de la diferencia mínima amb l'angle
sortint), li assignarem un parell de alçades o un altre parell.
		if (pC==0){
			alsadaI=0;
			alsadaF=525;
		}
		else if (pC==20){
			alsadaI=525;
			alsadaF=1019;
		}
		else if (pC==40){
			alsadaI=1019;
			alsadaF=1424;
		}
		else if (pC==60){
			alsadaI=1424;
…
```

```
...
                alsadaF=1696;
            }
            else if (pC==80){
                alsadaI=1696;
                alsadaF=1804;
            }
            //calcul alçada
            float alsadaGotes=alsadaF-alsadaI;
            incrementGotes=(((alsadaGotes*0.426)/16)-
alsadaInicial)/2.0;
        }
        else
        calculaLongImatgeB(longI,longF,aCamera,alLlum,phi);

            if (c[0]>0.78)//dalt esquerra
            {
                uvB[numLlums]=float2(longI/1440.0,alsadaI/1804.0);
                if (numLlums==0){
                    uvC=float2(0,alsadaI/1804.0);
                    pInitial.y=pInitial.y+incrementGotes;
                }
            }
            else if(c[0]>0.58 && c[0]<0.77)//dalt dreta
            {
                uvB[numLlums]=float2(longF/1440.0,alsadaI/1804.0);
                if (numLlums==0){
                    uvC=float2(1,alsadaI/1804.0);
                    pInitial.y=pInitial.y+incrementGotes;
                }

            }
            else if(c[0]>0.28 && c[0]<0.47)//baix esquerra
            {
                uvB[numLlums]=float2(longI/1440.0,alsadaF/1804.0);
                if (numLlums==0){
                    uvC=float2(0,alsadaF/1804.0);
                    pInitial.y=pInitial.y-incrementGotes;
                }
            }
            else if (c[0]<0.17)//baix dreta
            {
                uvB[numLlums]=float2(longF/1440.0,alsadaF/1804.0);
                if (numLlums==0){
                    uvC=float2(1,alsadaF/1804.0);
                    pInitial.y=pInitial.y-incrementGotes;
                }
            }
        }
        oPosition = mul(modelViewProj, pInitial);
        oWorldPos = mul( modelWorld, pInitial ).xyz;
}
```

Codi del píxel shader:

```
void main_ps(
	//inputs
	float2 uv[2]: TEXCOORD0,
	float2 uvC: TEXCOORD5,
	float3 posPuntDelShader: TEXCOORD4,
	//outputs
	uniform sampler2D imatgePuntuals : register(s1),
	uniform sampler2D imatgeAmbient : register(s2),
	uniform float4 lightdiffuse0,
	uniform float4 lightdiffuse1,
	uniform float4 lightambient,
	uniform float4 lightPosition1,
	uniform float4 lightDirection1,
	uniform float4 spotLightParams1,
	out float4 color : COLOR
)
{
	float4 lightDif[2];
	lightDif[0]=lightdiffuse0*0.18;
	//si es vol ressaltar encara més les gotes reflexades per el fanal es pot
augmentar el seguent factor
	lightDif[1]=lightdiffuse1*0.18;

	//calcul impacte gota amb spotlight
	//calcul de la direcció de la llum
	float3 dirLLumPunt = normalize(posPuntDelShader-
lightPosition1.xyz);

	float cosDirection= dot(dirLLumPunt,lightDirection1.xyz);
	float angle=acos(cosDirection);
	//calcul angles spotlight
	float innerAngle=acos(spotLightParams1.x);
	float outerAngle=acos(spotLightParams1.y);
	//en aquest cas posarem que el llum spotlight és a la posició 1, si es varies la
posició cal canviar aquest valor
	if (angle>outerAngle)
	{
//no iluminat per aquesta llum
		lightDif[1]=float4(0,0,0,1);
	}
	else if(angle>innerAngle)
	{
//fragment il·luminat amb llum mitja, si lightDiffuse1 no és la llum spotlight cal
canviar aquesta variable per la que ho es
		lightDif[1]=lightdiffuse1*smoothstep(innerAngle,
outerAngle, cosDirection);
	}
	else
	{
		//fragment il·luminat per la llum sencera
		lightDif[1]=lightdiffuse1;
	}
…
```

```
	//inicialitzo el color
	color=float4(0,0,0,1);

	for (int numLlums=0;numLlums<2;numLlums++)
	{
		color = color+tex2D(imatgePuntuals, 
uv[numLlums])*lightDif[numLlums];
	}
	//calculem el valor del color corresponent a la llum ambiental
	float4 ambient=tex2D(imatgeAmbient, uvC);
	ambient=ambient*lightambient*0.015;
	//sumem el color de les imatges puntuals amb el d'ambiental, això serà el 
color final
	color=color+ambient;
}
```