EPS
Escola Politècnica
Superior

Projecte/Treball Fi de Carrera

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

Títol: Visualització en temps real de múltiples llums amb la tècnica de Lightcuts

Document: Memòria

Alumne: Guillem González Noguer

Director/Tutor: Gustavo Patow
Departament: Informàtica i Matemàtica Aplicada
Àrea: Llenguatges i sistemes informàtics

Convocatòria: Juny/2009

Índex

1.- Introducció... 4
1.1.- Temporalització del projecte................................ 5
1.2.- Organització del document.................................. 7
2.- El motor gràfic OGRE3D.. 8
2.1.- Introducció als motors gràfics.............................. 8
2.2.- El motor gràfic OGRE3D...................................... 13
2.3.- Classes principals d'OGRE3D................................ 14
3.- Funcionament algoritme Lightcut................................. 23
3.1.- Introducció... 23
3.2.- Treball previ.. 23
3.3.- Notes d'implementació de Lightcuts........................ 24
3.4.- L'enfocament de Lightcuts.................................... 26
3.5.- Implementant l'algoritme lightcuts.......................... 30
3.6.- Càlcul dels errors màxims..................................... 31
3.7.- Càlcul del la iluminació estimada............................ 32
4.- Disseny.. 33
4.1.- Requeriments del sistema..................................... 33
4.1.1.- Requeriments funcionals................................ 33
4.1.2.- Requeriments no funcionals............................ 34
4.1.3.- Casos d'ús... 35
4.2.- Anàlisi del sistema... 39
4.2.1.- Diagrames d'activitat..................................... 39
4.3.- Disseny del sistema... 42
4.3.1.- Disseny de la interfície................................... 42
4.3.2.- Patrons utilitzats... 43
4.3.3.- Diagrama de classes...................................... 47
4.3.3.1.- Classe BoundingBox................................ 49
4.3.3.2.- Classe Ogre::Camera.............................. 50
4.3.3.3.- Classe OGRE::Entity............................... 50
4.3.3.4.- Classe GeneradorArbre............................ 51
4.3.3.5.- Classe OGRE::Light................................. 58
4.3.3.6.- Classe OGRE::Listener............................. 58
4.3.3.7.- Classe ListenerLlums............................... 59
4.3.3.8.- Classe NodeGrup.................................... 65
4.3.3.9.- Classe NodeLlum.................................... 68
4.3.3.10.- Classe NodePuntual............................... 70
4.3.3.11.- Classe OGRE::SceneManager.................. 70
5.- Implementació.. 73
5.1.- Classe GeneradorArbre.. 73
5.1.1.- Constructor GeneradorArbre............................ 73
5.1.2.- Mètode combinarParelles................................. 75
5.1.3.- Mètode distancia.. 76
5.1.4.- Mètode getMillorParella.................................. 77
5.1.5.- Mètode ordenarParelles................................... 77
5.1.6.- Mètode eliminarNoRepresentant........................ 77

5.1.7.- Mètode substituirRepresentant........................ 78
5.2.- Classe ListenerLlums... 78
5.2.1.- Mètode objectQueryLights............................. 79
5.2.3.- Mètode buildLightCut.................................... 80
5.2.4.- Mètode emplenarLlistaLlums........................... 82
5.2.5.- Mètode calcularCota...................................... 84
5.2.6.- Mètode illuminationEstimate........................... 85
5.2.7.- Mètode findWorstErrorBound.......................... 88
5.2.8.- Mètode sonTotFulles..................................... 89
5.2.9.- Mètode computeBoundAndEstimateForChildren.. 89
5.2.10.- Mètode updateTotalIllumination.................... 90
5.3.- Funcionament de CEGUI....................................... 90
5.4.- Altres aspectes de la implementació....................... 96
6.- Anàlisi dels resultats.. 97
6.1.- Comparació percentatges d'error........................... 102
6.2.- Problemes trobats... 107
7.- Conclusions.. 108
8.- Treball futur... 109
9.- Agraïments... 109
10.- Referències bibliogràfiques... 110

1.- Introducció

Avui en dia, un dels problemes més habituals que han d'afrontar els desenvolupadors de videojocs és la iluminació. És ben sabut que el fet de que els videojocs siguin realistes i atractius per a l'usuari és molt important, i un bon sistema d'iluminació juga un paper principal.

La gran contrapartida és la càrrega que aquesta iluminació suposa: el cost de tenir un reduït nombre de llums (parlem de menys de 8 llums en escena) no és massa elevat, però si a això li sumem el treball que han de fer la CPU i la GPU per a calcular altres factors com poden ser la intel·ligència artificial, la simulació de la física, la interacció amb l'usuari, etc. ens trobem que la iluminació queda relevada a un segon lloc i n'hem de disminuir la qualitat per a reduir la càrrega de càlcul.

Això què suposa? Reduir el nombre de llums a l'escena, la qualitat de la iluminació d'aquestes, la qualitat de les ombres que generen... En definitiva, és molt difícil oferir la qualitat gràfica desitjada sense sacrificar molts aspectes.

El que aquest projecte pretén és optimitzar el sistema d'iluminació de manera que la càrrega que representa per al sistema sigui inferior. Això, per suposat, sense haver de renunciar a la qualitat que tindríem sense fer servir aquest sistema.

Els objectius que ens proposem són els següents:

- Entendre i implementar l'algoritme de lightcuts
- Aconseguir optimització en una escena utilitzant l'algoritme de lightcuts.

Lightcuts [Walter 2005] ofereix una solució que compleix aquests 2 punts: optimitzar l'escena i no perdre qualitat. Per a fer-ho, la solució que suggereix és la següent: suposem una escena amb un nombre X de llums. Per cada punt de l'escena calcularem quines d'aquestes llums cal utilitzar per a iluminar-lo. D'aquesta manera farem servir un subconjunt de les llums per cada punt.

Això que sembla tan senzill i obvi comporta la dificultat de decidir quin serà aquest conjunt de llums. Podríem, per exemple, establir un llindar màxim de llums a utilitzar i escollir-les a l'atzar, però es podrien donar situacions on la iluminació per a certs punts donés realment mal resultats. A més, en funció del tipus d'escena de la que estiguessim parlant necessitaríem que l'algoritme escollit fós capaç d'adaptar-se a la situació: en un espai obert generalment disposem d'un gran focus de llum (el sol o la lluna) i potser alguna altra llum

complementària, en canvi en una habitació tancada solem tenir diferents focus de llums més reduides.

La solució de “lightcuts” es pot adaptar a totes les escenes ja que decideix, dinàmicament, quines llums afectaran a cada punt. La solució consta de dos parts bastant diferenciades; una es realitza al preprocessament i l’altra en temps d’execució.

Primer de tot, a l’hora de fer el preprocessament, agrupem les llums de la manera més adequada possible: en aquest cas les ajuntarem en un arbre binari, de dos en dos en funció de diferents paràmetres. Quan hem ajuntat dues llums, decidirem quina de les dues representarà al conjunt, de manera que poguem decidir si utilitzem només la llum representant o volem fer servir les dos llums filles. Així, a dalt de tot de l’arbre hi haurà una sola llum que representaria totes les llums de l’escena, i abaix de tot, a les fulles de l’arbre, hi haurà les llums originals en si mateixes.

El segon pas, que té lloc a l’execució, consisteix en recorrer aquest arbre i trobar el conjunt de llums que satisfaci els nostres requisits. Començarem des de dalt i anirem afegint més llums al conjunt total. Quan creiem que el conjunt obtingut satisfà l’error màxim al que estem predisposats ens aturarem. A aquest conjunt de llums l’anomenarem “lightcut”.

Per a més detalls de l’implementació de l’algoritme, dirigir-se a l’apartat 3 del projecte.

1.1.- Temporalització del projecte

En aquest apartat es pot veure com s’ha dividit el temps en les diferents parts del projecte.

El projecte es va començar al setembre del 2007, però degut a la falta de temps per compaginar-lo amb el món laboral ha anat avançat per fases. A continuació es mencionarà cada tasca realitzada juntament amb el temps aproximat que ha comportat.

- **Aprenentatge de les llibreries d’Ogre**: aprendre el funcionament del motor gràfic emprat va ser la primera tasca. Per a fer-ho es van realitzar gran quantitat de tutorials trobats a la xarxa. Aquesta tasca va durar uns 4 mesos.
- **Estudi de l’article de lightcuts**: a mesura que s’anaven adquirint més coneixements d’Ogre es va fer inevitable començar a introduïr-se en el complicat algoritme de Lightcuts. Per a fer-ho es va llegir l’article repetidament i se’n va realitzar

un resum. Aquesta tasca va durar uns 3 mesos i va començar-se al segon més de l'aprenentatge d'Ogre.

- **Descripció dels requeriments**: quan es va acabar l'aprenentatge de les llibreries d'Ogre i l'estudi de l'algoritme de lightcuts el pas lògic va ser definir els requeriments. Aquest apartat va durar aproximadament un mes, però s'ha anat modificant a mesura que avançava el projecte.
- **Anàlisi del sistema**: un cop finalitzats els requeriments, comença l'anàlisi del sistema. Aquest apartat va durar un mes i també s'ha anat modificant a mesura que ha avançat el projecte.
- **Disseny del sistema**: en aquest apartat es van fer els diagrames pertinents i, el que és més important, es va desenvolupar un diagrama de classes. Es va dur a terme al mateix temps que l'anàlisi del sistema.
- **Implementació**: aquest apartat és el que ha patit més retrassos degut als problemes de compaginació laborals. Es podria dir que ha durat més d'un any, des de la implementació de les classes més senzilles fins a l'interfície gràfica.
- **Proves**: aquest apartat no es pot situar exactament en el temps ja que s'han anat fent proves juntament amb el procés d'implementació, tot i que els últims mesos s'han intensificat molt les proves.
- **Documentació**: tot i que al llarg del projecte s'ha anat escrivint part de la documentació, ha estat durant els últims dos mesos i mig quan realment s'ha dut a terme el gruix d'aquest apartat.

1.2.- Organització del document

A continuació descriurem com està organitzat aquest document, per poder utilitzar-lo de la forma més convenient.

Al primer capítol trobem la introducció amb un resum de la problemàtica que intenta resoldre Lightcuts. A continuació hi ha l'apartat de temporització del projecte: on s'explica el temps dedicat a cada apartat des de l'inici fins al final del projecte.

A continuació, al segon capítol, es detalla el motor gràfic utilitzat, OGRE3D. Dins d'aquest segon capítol tenim la introducció als motors gràfics, on es defineix què és un motor gràfic i un motor de joc, el motor gràfic OGRE3D, on s'expliquen les característiques del motor gràfic utilitzat, i les classes principals d'OGRE3D, on es resum les classes més importants d'Ogre que utilitzarem.

Al tercer capítol, funcionament de l'algoritme lightcuts, presentem l'algoritme de lightcuts. A continuació trobem el treball previ, on s'expliquen altres investigacions que s'han fet en el mateix camp, les notes d'implementació de Lightcuts, que donen detalls tècnics de l'algoritme i, finalment, l'enfocament de Lightcuts, que resum el funcionament de l'algoritme en si mateix.

Al quart capítol, disseny, trobem els apartats requeriments del sistema, on es resum la metodologia utilitzada, requeriments funcionals, requeriments no funcionals i fitxes de cas d'ús. Tot seguit ens trobem amb l'anàlisi del sistema, on s'investiga el problema a resoldre sense entrar en detalls d'implementació, mostrant els diagrames d'activitat. Finalment hi ha l'apartat disseny del sistema on es comenta el procés de creació d'una solució conceptual que permeti implementar els requeriments del sistema desitjat.

Al cinquè capítol ens trobem l'apartat d'implementació, on es comenten els mètodes més importants en detall. A més, comenta com funciona l'interfície gràfica utilitzada.

Al sisè capítol analitzem els resultats amb 2 escenes diferents i comparem com influeixen els percentatges d'error en una de les dos. També parlem dels problemes trobats mentre es duia a terme el projecte.

Al setè capítol es comenten els objectius del projecte i els objectius personals.

Al vuitè capítol trobem l'apartat de treball futur. Al novè capítol ens trobem amb els agraïments. La documentació acaba amb les referències bibliogràfiques.

2. El motor gràfic OGRE3D

2.1.- Introducció als motors gràfics

Al món del desenvolupament de videojocs i aplicacions complexes emprant gràfics és molt important no haver-se d'enfrontar directament amb una llibreria gràfica, com poden ser Direct3D o OpenGL. Tot i disposar de molta llibertat a l'hora d'utilitzar-ne una, està clar que ens enfrontarem amb més problemes apart dels gràfics, com poden ser, en el cas dels videojocs, un motor físic, una història i argument, el disseny de personatges, decorats, etc. Així doncs, és lògic que es vulgui treballar amb un llenguatge a més alt nivell: i és aquí on entren en joc els motors gràfics. Són els encarregats de gestionar els aspectes més feixucs de la programació grafica, com la càrrega de textures, models 3D, visualització d'objectes, etc.

A més, cal diferenciar entre dos conceptes: el motor gràfic i el motor de joc ("graphic engine" i "game engine" respectivament).

Un motor gràfic es centra exclusivament en els gràfics, com el seu nom indica. Si volem dotar a l'aplicació de so, motor físic i d'altres característiques que ja hem anomenat, ens farà falta implementar nous mòduls o aconseguir-los a través de llibreries externes.

En canvi, un motor de joc porta incorporats tots els mòduls que es solen requerir a l'hora de desenvolupar un joc.

Aquesta diferència que pot semblar tan sutil és important per poder entendre perquè hem triat un motor gràfic o un altre. A continuació destacarem els motors gràfics i de joc més importants, diferenciant entre els motors gràfics i els motors de joc.

Motors gràfics

- **Ogre3D:** motor gràfic lliure i gratuït, publicat al 2005 sota llicència LGPL. Està programat amb C++ i pot treballar amb DirectX o OpenGL. A més, és compatible amb Windows, Linux i Mac OS X. És orientat a objectes, molt modulable i bastant intuitiu d'utilitzar, permetent a més baixar de nivell si fa falta. La comunitat d'OGRE és bastant extensa i hi ha molta documentació i tutorials. A la figura 2.1.1 podem veure el logotip d'OGRE3D.

Figura 2.1.1 – Logotip del motor gràfic OGRE3D

- **Irrlicht:** motor gràfic lliure i gratuït, publicat sota llicència "zlib license" al 2006. És multi-plataforma, oficialment funciona sota Windows Mac OS X, Linux i Windows CE, i degut a la comunitat que té s'han creat versions per Xbox, PSP, SymbianOS i iPhone. A més, tot i estar originalment creat per a funcionar sota C++, també és compatible amb altres llenguatges com .NET, Java, Perl, Ruby o Python. A continuació podem veure una imatge generada per aquest motor gràfic, a la figura 2.1.2.

Figura 2.1.2 – Imatge generada pel motor gràfic Irrlicht

- **jMonkey Engine:** motor gràfic de codi lliure i gratuït, publicat sota llicencia "BSD license". Està escrit per complet en Java i un dels seus objectius és desenvolupar gràfics que donin un molt bon rendiment primant sobre l'utilització de tecnologies de vanguarda. Tot i que encara està en versió beta funciona sota

OpenGL i accepta entrades de ratolí, teclat i altres dispositius de control. A la figura 2.1.3 es pot veure el logotip de jMonkey.

Figura 2.1.3 – Logotip de jMonkey Engine

- **OpenSceneGraph:** motor lliure i gratuït publicat sota llicència LGPL. Està escrit en C++ i treballa sobre OpenGL, cosa que el fa compatible amb Windows, Linux, Mac OS X, IRIX, Solaris i FreeBSD. S'ha utilitzat en diferents camps: videojocs, modelatge, realitat virtual, visualització científica, etc. Tot i això no disposa d'una gran comunitat.

Motors de joc

- **RealmForge:** és un motor lliure i gratuït sota llicència LGPL, que es va deixar de produïr al 2005 per donar peu al Visual3D.NET Game Engine. Va ser un dels primers motors de joc seriosos creats per a .NET.

- **Visual3D.NET Game Engine:** motor gràfic comercial successor del RealmForge. Està pensat per a funcionar sota Windows i, en un futur, Xbox360. Està escrit en C#.NET i disposa de múltiples mòduls enfocats als videojocs com motor físic, so, scripting o joc en xarxa. No només té una comunitat bastant gran de desenvolupadors sinó que la mateixa companyia ofereix servei d'ajuda al programador. A la figura 2.1.4 es pot veure una imatge d'una escena generada amb Visual3D.NET.

Figura 2.1.4 – Imatge generada amb Visual3D.NET

- **Truevision3D:** motor gràfic comercial del qual se'n va crear la primera versió al 1999. Està programat en Visual Basic6 i C++ i treballa per sobre de l'API DirectX, per tant només és compatible amb Windows. Tot i això, es pot utilitzar amb diversos llenguatges, com C++, C#, Delphi, Visual Basic 6 o Visual Basic .NET. Com pràcitcament tots els motors de jocs descrits és orientat a objectes i disposa dels mòduls més comuns: audio, xarxa, motor físic...

- **Nebula:** motor gràfic de codi lliure i gratuït, publicat sota llicència MIT (Massachusetts Institute of Technology) al 1988. Està escrit en C++ i suporta scripting, és compatible amb Windows i pot ser utilitzat amb C++, Python o Java. Tot i que de moment només és compatible amb DirectX s'està treballant en donar suport a OpenGL i fer-lo compatible amb Linux i Mac OS X.

- **Crystal Space:** motor lliure i gratuït, publicat sota llicència LGPL al 1997. És molt portable (funciona sota Windows, GNU/Linux, Unix i Mac OS X) i està programat en C++ orientat a objectes. Encara que disposa de mòduls de so, colisió d'objectes i un motor físic, és bastant modular i no només s'utilitza per a videojocs. Tot i això ens trobem amb que té una comunitat més aviat petita darrere.

- **Delta3D:** és un motor de jocs lliure sota llicència LGPL. La seva característica principal és que, gràcies al seu disseny modular,

pretén integrar altres projeces de codi lliure com OpenSceneGraph, Cal3D o OpenAL sota una API fàcil d'utilitzar.

En definitiva, els motius que ens han portat a triar OGRE3D són bastant clars: volem un motor gràfic senzill d'utilitzar, gratuït i del qual existeixi molta documentació. Per tant, pràcticament tots els motors de jocs queden descartats, igualment que aquells pels quals s'ha de pagar la llicència.

A més, és molt flexible i té la capacitat de gestionar llums individualment en funció de cada objecte. OGRE no només es pot programar en C++ sinó que s'han fet adaptacions a altres llenguatges, tot i que hem triat aquest per ser el principal. Per acabar, però no menys important, funciona sobre Linux, Windows i Mac OS X, amb DirectX o OpenGL, cosa que el fa el millor candidat.

2.2.- El motor gràfic OGRE3D

OGRE3D és un motor gràfic el nom del qual ve donat per les sigles Object Oriented Graphics Rendering Engine, és a dir, motor de renderització gràfica orientat a objectes.

És important ressaltar que no es tracta d'un motor de joc ja que nomes s'encarrega dels gràfics. Això vol dir que no disposarem, directament, d'altres mòduls necessaris per a desenvolupar videojocs, ja que OGRE ha estat desenvolupat de forma que sigui modular i independent, deixant a elecció del dissenyador la tria d'aquests mòduls. Tot i això, darrere seu hi ha una comunitat que ha desenvolupat una gran quantitat de mòduls especifics per cubrir els aspectes necessaris per a desenvolupar un videojoc (so, entrada/sortida, connexió en xarxa, motor físic...).

Seguint el concepte de l'orientació a objectes, OGRE permet abstreure els detalls d'utilització de llibreries gràfiques com Direct3D o OpenGL, proporcionant una interfície a més alt nivell.

OGRE es gratuït i lliure, i està publicat sota llicència LGPL (GNU Lesser General Public License). És multi-plataforma i la última versió estable disponible és la 1.6.2, disponible per a Linux, Windows o Mac OS X.

OGRE s'utilitza per a desenvolupar diferents tipus de jocs comercials, com Anarchy Online (figura 2.2.1) o Earth Eternal (figura 2.2.2), o d'altres de lliures, com OpenFrag (figura 2.2.3), i fins i tot un framework per desenvolupar jocs massius multijugador de rol online (MMORPG) anomenat WorldForge.

Fig 2.2.1 - Anarchy Online

Fig 2.2.2 - Earth Eternal

Fig 2.2.3 - OpenFrag

2.3.- Classes principals d'OGRE3D

A continuació es farà un resum de les classes necessàries per a introduir-se en l'OGRE. A la figura 2.3.1 es pot veure un diagrama amb les classes principals d'OGRE i les com estan relacionades entre elles. Només apareixen les classes més bàsiques i importants, i tot i així moltes d'elles no seran comentades ja que no són necessàries per entendre la implementació del projecte.

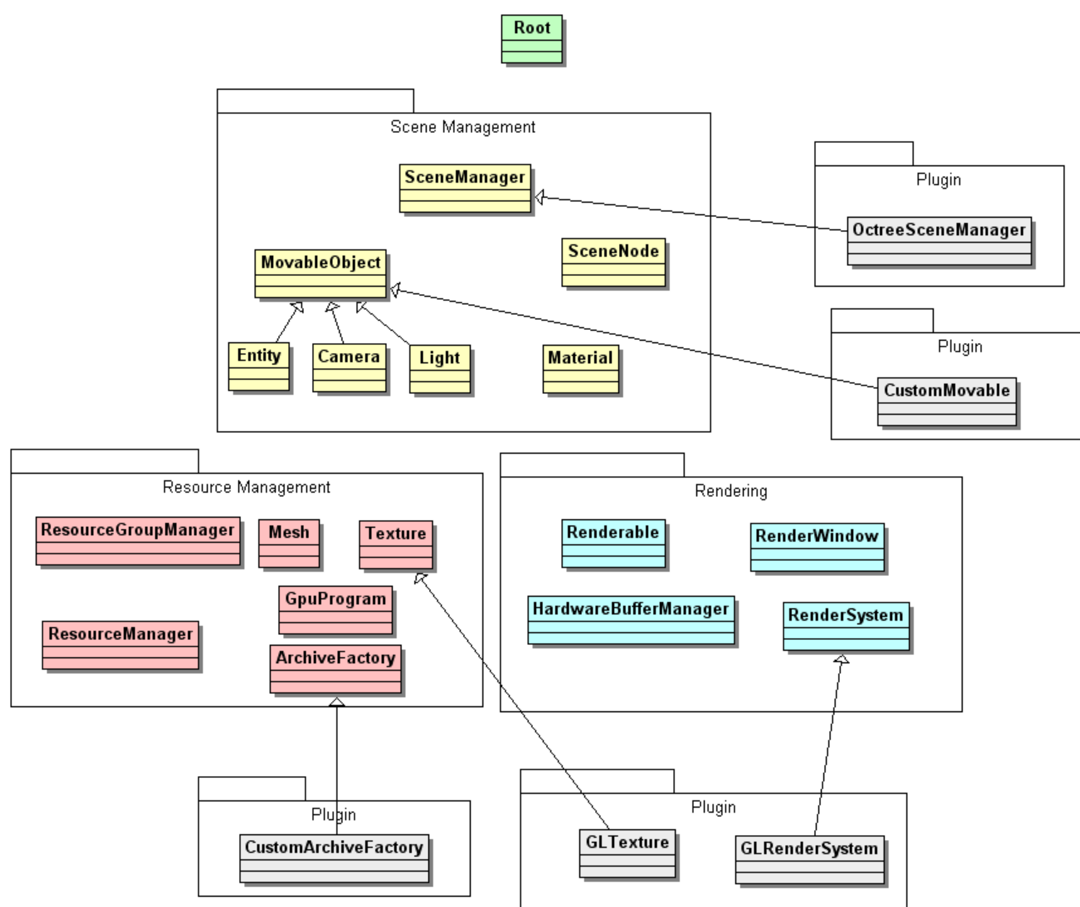


Figura 2.3.1 – Diagrama de classes d'OGRE3D

2.3.1.- Classe Root

És la classe arrel d'OGRE, tal i com el seu nom indica, i representa un punt de partida per al client de l'aplicació.

Des d'aquí l'aplicació pot accedir a parts fonamentals del sistema, com els diferents sistemes de renderització disponibles, el sistema de logging, el control de les configuracions guardades, i altres classes del sistema.

És lògic per tant que l'objecte Root sigui el primer que cal instanciar per a poder treballar amb OGRE. Es pot accedir a la instància des de qualsevol punt. Ja que es tracta d'una classe que implementa el patró Singleton només en podrem tenir una instància creada en tot moment.

A més permet configurar aspectes importants de l'OGRE com la mida de la finestra, la targeta gràfica, la resolució, etc. També permet decidir si volem o no que ens obri el quadre de diàleg de configuració d'OGRE (figura 2.3.1.1), on es poden escollir aquestes opcions.

Figura 2.3.1.1 - Quadre de diàleg de configuració d'OGRE3D

2.3.2.- Classe SceneManager

Una altra classe molt important dins de l'OGRE. És la encarregada d'organitzar pràcticament tots les aspectes d'una escena. Des de càmeres a llums passant per qualsvevol objecte que es pugui visualitzar, des de simples punts o plans fins a objectes molt més complexes representats per malles o "meshes". Aquest tipus d'objecte "visible" s'anomena "Entity" i més endavant en parlarem amb més detall. Per a relacionar un objecte amb l'escena l'hem de vincular a aquesta a través un "SceneNode", és a dir, un node de l'escena.

A més a més OGRE ens proporciona diferents tipus de "SceneManagers" depenent de l'escena que necessitem renderitzar. En el nostre cas la solució per defecte és la més eficient, però si volguessim renderitzar grans quantitats de terreny amb més o menys extensió, o interiors amb moltes parets, podríem trobar un "SceneManager" més adequat pels nostres propòsits.

2.3.3.- Classe SceneNode

"SceneNode" és la classe utilitzada per a organitzar els objectes dins de l'escena. Cada node emmagatzema un objecte (és a dir, una llum, una càmera, una "Entity"...) i cada node forma part d'un mateix arbre de nodes. Una entitat no serà renderitzada fins que aquesta no ha estat vinculada a un "SceneNode". A més, és en aquesta classe on es guarda la posició i l'orientació de l'objecte. Com és lògic, un "SceneNode" no té representació visual, només la de l'objecte que conté, si aquest és visible.

A més, a un mateix "SceneNode" hi podem guardar tots els objectes que volguem (per exemple, suposem que volem, en un mateix punt, guardar-hi una persona, una llum i una càmera).

D'altra banda, podem relacionar dos "SceneNode" entre si jeràrquicament, de manera que la posició del fill és relativa a la del pare: si el pare està situat a la posició (0,1,1) i el fill a la (1,3,5) vol dir que respecte a l'escena el fill estarà a la posició (0,1,1) + (1,3,5) = (1,4,6).

2.3.4.- Classe RenderSystem

Aquesta és una classe abstracta que defineix una interfície per l'API gràfica que tenim per sota; és a dir Direct3D o OpenGL, per citar els dos exemples més coneguts. S'encarrega d'enviar operacions de renderització a l'API i configurar totes les opcions possibles d'aquesta. Una de les subclasses existents és D3DRenderSystem, que com el

seu nom indica és l'encarregada de comunicar-se amb l'API de Direct3D.

Com és lògic la classe encarregada de decidir quin sistema de renderització volem és la classe "Root".

De totes maneres no es sol necessitar manipular aquesta classe directament, ja que des de les classes "SceneManager" o "Material" solem tenir disponibles la gran majoria d'opcions de renderització.

2.3.5.- Classe MovableObject

Classe abstracta que defineix un objecte mòbil dins d'una escena. Les instàncies d'aquesta classe s'hauran de vincular a un "SceneNode" per a poder-ne definir la posició i, si són visibles, per a renderitzar-les.

2.3.6.- Classes Entity, Mesh i Material

La classe "Entity" és una instància de la classe "MovableObject" que representa qualsevol objecte que es pugui posar en una escena. Les "Entity" també es creen des del "SceneManager", donant-li un nom únic que l'identificarà, per si es necessita accedir-hi en el futur, i el nom de la malla que el representarà. A la figura 2.3.6.1 es pot veure el diagrama de classes d'UML que representa la relació entre aquestes 3 classes.

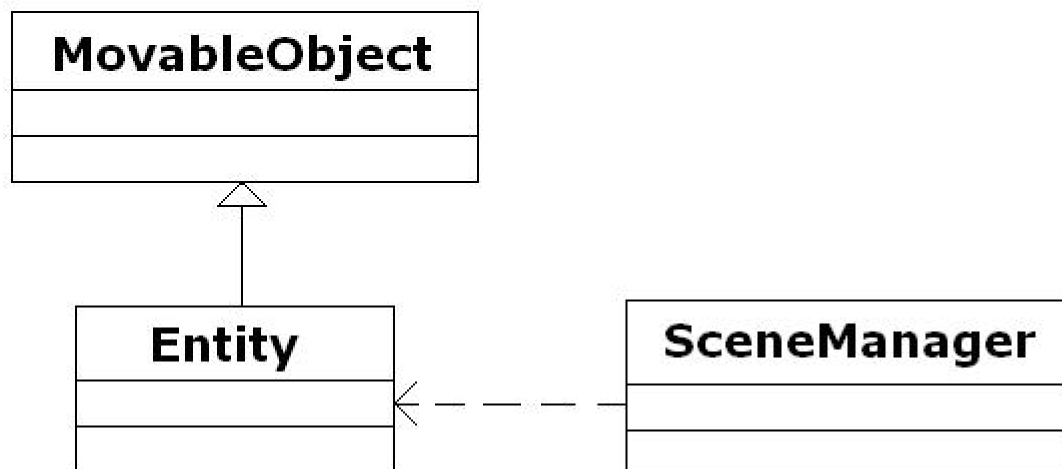


Figura 2.3.6.1 – Diagrama de classes amb les relacions entre Entity, MovableObject i SceneManager

Les entitats es representen gràficament amb una malla o “Mesh”: una estructura gràfica en 3D composta de vèrtexs, eixos i cares. Quan carreguem una “Mesh” se li assigna un “Material” per defecte. La classe “Mesh” representa la geometria d’un objecte determinat. Per tant, si volem tenir vàries còpies d’una mateixa malla necessitarem un sol objecte “Mesh” i tantes instàncies d’objectes “Entity” com necessitem.

La classe Material determina quin aspecte té una superfície quan és renderitzada. Podem assignar-li textures, colors, activar o desactivar la llum i la generació d’hombres, etc. Un dels aspectes atractius d’OGRE és que permet definir les propietats dels materials des de codi font o a través de fitxers, evitant així la repetició de codi i independitzant aquest aspecte.

Una “Mesh” pot tenir més d’un “Material”, per a poder decidir quin aspecte volem que tingui cada part d’una “Mesh”. És per això que existeix la classe “SubMesh”. Una “Mesh” està composta per diferents “SubMesh” i cada “SubMesh” tindrà assignat un sol objecte de tipus “Material”. Si volem una entitat amb un sol tipus de material, tindrem una “Mesh” amb un sol “SubMesh”. A més, des de la classe “Mesh” podem accedir a tots els “SubMesh” que la composen per canviar, si ho desitgem, els materials que composen la malla.

A continuació podem veure dos meshes, una amb textures (figura 2.3.6.2) i una altra sense (figura 2.3.6.3).

Figura 2.3.6.2 - Ogrehead amb textures

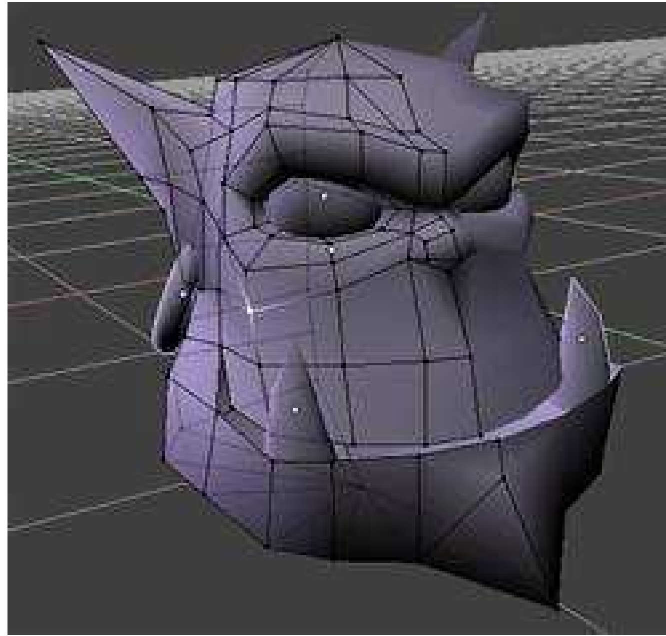


Figura 2.3.6.3 - Ogrehead sense textures

2.3.7.- Classe Camera

La classe "Camera" és, com el seu nom indica, la que representa els objectes que ens permetran visualitzar una escena. OGRE s'encarrega de renderitzar les escenes des d'un punt de vista (que ve determinat per la posició i un vector que determina on ha de mirar la càmara) cap a un búfer, que sol ser una finestra o una textura.

Podem escollir entre projeccions ortogràfiques (l'objecte no canvia de mida a mesura que s'allunya de la càmera) i projeccions en perspectiva (l'objecte si que canvia de mida a mesura que s'allunya), depenent de l'efecte que desitgem.

Les càmeres són una sub-classe de "MovableObject", per tant es poden assignar a un "SceneNode". Amb això podem determinar la posició de la càmera a través del node al que pertany, fer moure la càmera si el node s'està movent o posar una càmara al punt de vista d'una persona. A continuació podem veure, a la figura 2.3.7.1, el diagrama de classes d'UML amb les relacions entre les classes MovableObject, Camera i SceneNode.

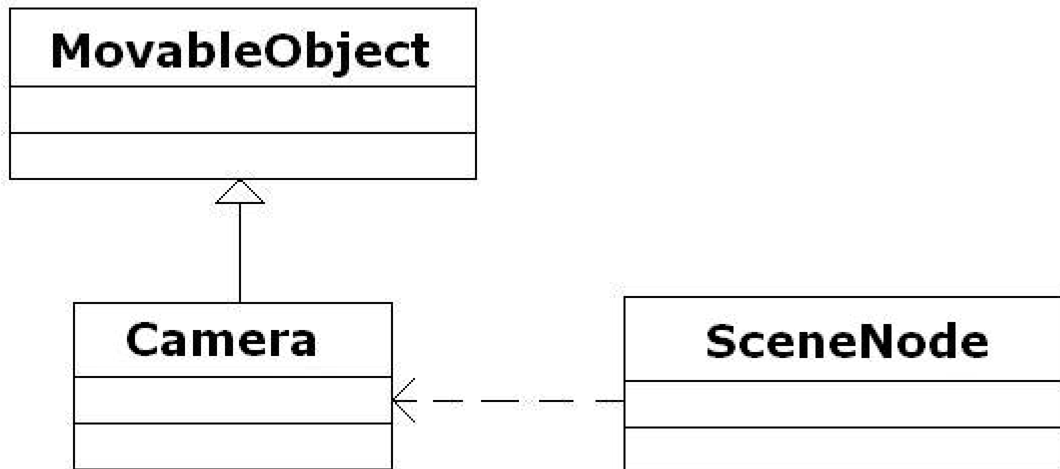


Figura 2.3.7.1 – Diagrama de classes amb les relacions entre Camera, MovableObject i SceneNode

2.3.8.- Classe Light

Classe que representa una llum, molt important pel desenvolupament del projecte. Aquesta classe també hereta de "MovableObject", per tant pot ser vinculada a un "SceneNode". OGRE suporta ombres però pel nostre cas seran desactivades, doncs faria falta un càlcul extra per a computar-les. Tot i així enumerarem els 3 tipus d'ombres suportats, ja que un ens permetrà tractar les llums de cada objecte independentment:

- Modulative Texture Shadows: genera una textura amb l'ombra que després s'aplica a l'escena.
- Modulative Stencil Shadows: després de que tots els objectes no transparents de l'escena s'hagin renderitzat es generen totes les ombres com a modulacions.
- Additive Stencil Shadows: cada llum implica un pas adicional a l'hora de renderitzar l'escena. És el mètode més costós i alhora el que dóna millor resultat. En el nostre projecte és el que fem servir, ja que, tot i no utilitzar ombres, ens permet tractar de forma independent cada llum dins el projecte i fer servir un mètode anomenat "setListener", per a adjudicar a cada objecte visible de l'escena un listener encarregat de decidir quines llums iluminaran aquest objecte.

A continuació un exemple de cada tipus d'ombra, Modulative Texture Shadows (figura 2.3.8.1), Modulative Stencil Shadows (2.3.8.2), Additive Stencil Shadows (2.3.8.3).

Figura 2.3.8.1 - Modulative Texture Shadows

Figura 2.3.8.2 - Modulative Stencil Shadows

Figura 2.3.8.3 - Additive Stencil Shadows

A més, OGRE diferencia entre 3 tipus diferents de llums:

- Point (LT_POINT): punt de llum que emet cap a totes les direccions possibles. És el tipus de llum que utilitzem en el projecte.
- Spotlight (LT_SPOTLIGHT): llum similar a la d'una llanterna, tenim un raig de llum que ilumina més al centre i menys com més lluny en som.
- Directional (LT_DIRECTIONAL): aquest tipus de llum simula la generada per una llum llunyana que impacta a tota l'escena, com podria ser la de la lluna o el sol.

Les propietats més importants de les llums són el color difús i l'especular, que determinaran com afecten al color difús i especular del material que estan iluminant. També podem determinar l'atenuació d'una llum puntual, és a dir, com la llum perd potència amb la distància.

Finalment, des de la classe "SceneManager" podem crear una llum que afecta igualment a tots els objectes de l'escena, anomenada "ambientLight", però que en el nostre cas no serà utilitzada, ja que ens interessa avaluar fielment l'efecte de les llums.

3.- Funcionament algoritme Lightcut

Lightcuts [Walter 2005] és un sistema escalable per calcular iluminació de caire realista. Manipula geometria arbitrària, materials no difusos i iluminació amb llums puntuals des de gran quantitat de fonts. La part central és un algoritme per aproximar acuradament la iluminació des de diferents punts de llum amb un cost sublinear. Gràcies a aquest, un gran nombre de llums pot ser econòmicament aproximat donat un error màxim d'aproximació. Llavors utilitzem un arbre binari de llums i càlculs de visibilitat per a dividir les llums en grups per valorar el grau d'error i els costos.

3.1.- Introducció

Mentre molta recerca s'ha centrat en generar escenes amb geometria i materials complexos, poca cosa s'ha fet a l'hora de manipular grans conjunts de llums. En els sistemes més típics, els costos de renderització augmenten linealment amb el nombre de llums. Les escenes utilitzades al món real solen contenir gran quantitat de punts de llum i els estudis mostren que la gent sol preferir imatges amb iluminació rica i realista. D'altra banda, al món de la informàtica gràfica, estem forçats a utilitzar menys llums o a desactivar una part important dels efectes d'iluminació, com les ombres, per a evitar excessius costos de renderització.

Lightcuts es presenta com un algoritme escalable per a computar la iluminació d'una escena amb molts punts de llum. El seu cost de renderització és fortament sublinear amb el nombre de punts de llum alhora que es manté una alta fidelitat amb la solució exacta. Proveeix un sistema ràpit d'aproximar la iluminació d'un grup de llums i, el que és més important, límits raonables a l'error màxim en els resultats. També proveeix un sistema d'agrupació de llums en grups automàtic i en funció de la seva posició per controlar la relació entre el cost i la mida de l'error. Les llums s'organitzen en un arbre binari per a una eficient partició. Els lightcuts poden manipular materials no difusos i qualsevol geometria que pugui ser sotmesa a un procés de "ray casting".

Tenir un algoritme escalable permet manipular un nombre de llums extremadament elevat. Això és especialment útil perquè molts altres problemes difícils d'iluminació poden ser simulats utilitzant l'iluminació de sufícients punts de llum.

3.2.- Treball previ

Hi ha una gran quantitat de treball realitzat en el camp de la computació d'iluminació i ombres de punts de llum. La majoria de les tècniques acceleren el processament de llums individuals però augmenten el cost linealment a mesura que augmentem el nombre de llums.

Unes quantes tècniques han tractat directament amb el problema de les grans quantitats de llums. [Ward 1994] ordena les llums segons la màxima contribució en l'escena i llavors evalua la seva visibilitat en ordre decreixent fins que s'arriba a un error màxim. [Shirley 1996] divideix l'escena en cel·les i per cada cel·la separa les llums en importants i no importants amb aquestes escassament mostrejades. Aquestes tècniques de Monte Carlo o similars poden comportar-se bé donada una escena creada adequadament per aquest propòsit, però amb escenes arbitràries encara és un problema. [Paquette 1998] presenta una aproximació jeràrquica similar a la de "lightcuts". Proveeixen errors màxims i escalabilitat però no suporten ombres. [Fernandez 2002] accelera el cost d'iluminació amb moltes llums guardant l'informació de cada llum a la caché de l'escena, però comporta requeriments de memòria molt elevats si el nombre de llum és alt. [Wald 2003] pot manipular eficientment moltes llums assumint que només un conjunt reduit de llums afecta a cada imatge. Aquest conjunt es determina utilitzant un preprocés de traça de partícules.

3.3.- Notes d'implementació de Lightcuts

En aquest apartat es presentaran algunes definicions bàsiques per entendre correctament el funcionament del projecte.

Definicions:

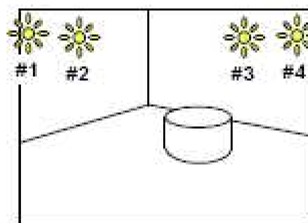


Figura 3.3.1 – Escena amb 4 llums

Arbre de llums: un arbre de llums és un arbre binari on les fulles del qual són llums individuals i els nodes interiors són clústers de llums. Entenem clúster com una agrupació de dues llums on una d'aquestes és la representant. Alhora, aquestes llums poden ser fulles (per tant, llums individuals) o clústers una altra vegada. Gràcies als clústers es genera l'estructura d'arbre binari, on el representant d'un clúster conté totes les llums des del mateix clúster fins a les fulles. A la figura 3.3.1 podem veure una escena amb 4 llums i a la figura 3.3.2 un exemple d'arbre de llums.

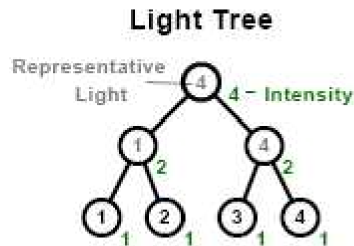


Figura 3.3.2 – Arbre de llums

Cut de llums: un cut de l'arbre es defineix com un conjunt de nodes de manera que, tot camí des de l'arrel de l'arbre fins una fulla contindrà exactament un sol node del cut. Per tant es pot dir que cada cut es correspon amb una partició vàlida de les llums en clústers. A la figura 3.3.3 podem veure un exemple d'un cut de llums.

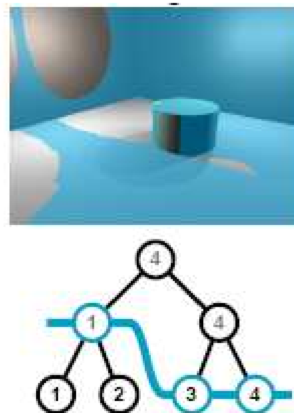


Figura 3.3.3 – Cut de llums

Lightcut: un cut que cumpleix el criteri d'energia total, seguint les fórmules explicades a la secció 3.5. A la figura 3.3.4 es poden veure 3 lightcuts diferents, cadascun dels quals es correspon al punt indicat amb una creu a l'escena.

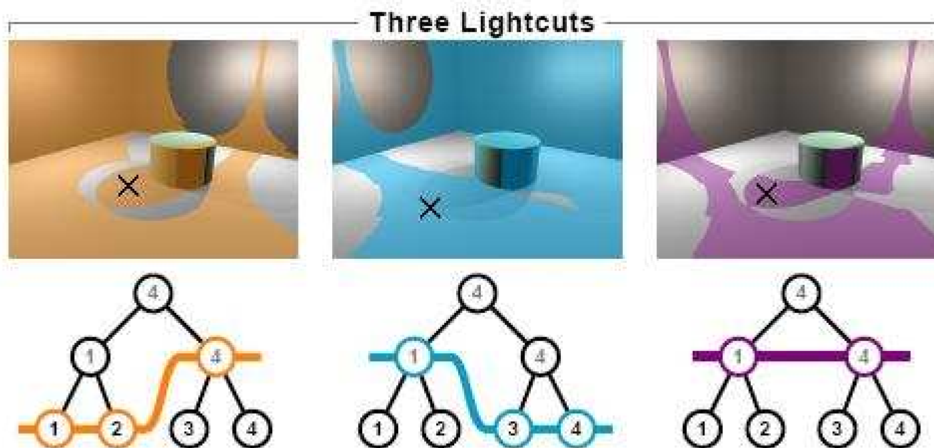


Figura 3.3.4 – 3 exemples de lightcut corresponents cadascun a un punt diferent de l'escena

3.4.- L'enfocament de Lightcuts

Donat un conjunt de llums 'S', la radiància 'L' causada per la seva iluminació directa en un punt de la superfície 'x' vist des d'una direcció 'W' és el producte dels termes material, geometria, visibiltat i intensitat sumats per totes i cadascuna de les llums. A la figura 3.4.1 tenim l'equació de radiància a la que fem menció.

$$L_{\mathbb{S}}(\mathbf{x},\omega)=\sum_{i\in\mathbb{S}} \overbrace{M_i(\mathbf{x},\omega)}^{\text{material}}\ \underbrace{G_i(\mathbf{x})}_{\text{geometric}}\ \overbrace{V_i(\mathbf{x})}^{\text{visibility}}\ \underbrace{I_i}_{\text{intensity}}$$

Figura 3.4.1 - Equació de radiància

El cost de la solució exacta és lineal respecte al nombre de llums ja que aquests termes s'han d'avaluar per a cada llum. Per a crear un mètode sublinear hem d'aproximar la contribució d'un grup de llums sense haver d'avaluar cada llum individualment.

Definirem un clúster "C" com un conjunt de llums amb un representant "j". La iluminació directa del clúster pot ser aproximada utilitzant el material, la geometria i la visibilitat del representant per a totes les llums. D'aquesta manera obtenim l'equació mostrada a la figura 3.4.2.

$$L_{\mathbb{C}}(\mathbf{x},\omega) = \sum_{i\in\mathbb{C}} M_i(\mathbf{x},\omega)\, G_i(\mathbf{x})\, V_i(\mathbf{x})\, I_i$$
$$\approx M_j(\mathbf{x},\omega)\, G_j(\mathbf{x})\, V_j(\mathbf{x}) \sum_{i\in\mathbb{C}} I_i$$

Figura 3.4.2 - Equació de radiància aproximada pel representant

L'intensitat del clúster pot ser precalculada i guardada per fer el cost d'avaluació d'un clúster aproximadament igual al cost d'avaluar una sola llum (seria el mateix que dir que hem substituit un clúster de llums per una sola llum més potent). La mida de l'error dependrà de com s'assemblin els termes material, geomètria i visibilitat en tot el clúster.

Construcció de l'arbre de llums.

No hi ha cap partició de llums en clústers que pugui funcionar bé amb tota la imatge, però trobar dinàmicament una partició de clústers per cada punt seria excessivament costós. Per tant, utilitzarem un arbre de llums general per a calcular eficientment particions de clústers locals. Un arbre de llums és un arbre binari on les fulles són fonts puntuals de llum i els nodes interiors són clústers de llums que contenen les llums que tenen per sota a l'arbre. Anomenarem "cut" a un conjunt de nodes de manera que cada camí des de l'arrel fins a una fulla contindrà exactament un node del cut. Per tant tot "cut" correspondrà a una partició vàlida de les llums en clústers. A la figura 3.4.3 es pot veure un exemple d'arbre de llums i 3 cuts diferents.

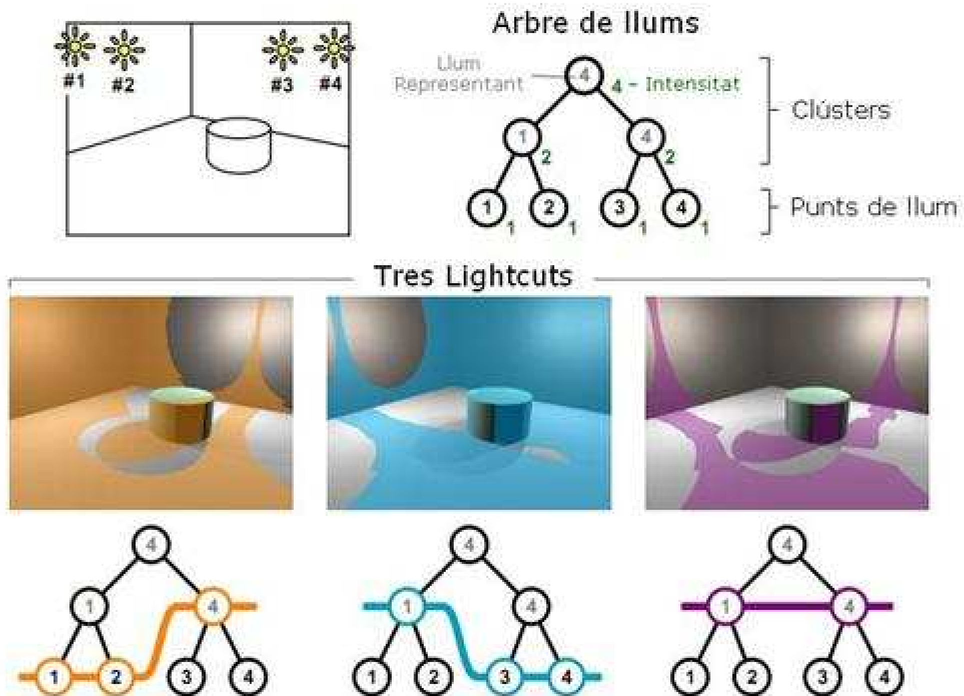


Figura 3.4.3 - Arbre de llums i 3 possibles cuts

Mentre cada “cut” correspon a una partició de clústers vàlida, poden variar enormement en cost i en qualitat de l’aproximació de l’iluminació. Necessitem un sistema robust i automatitzat per a escollir automàticament el cut apropiat per a utilitzar localment. Com que variarem els “cuts” en tota l’escena, alguns punts poden utilitzar un clúster particular per a reduir costos mentre d’altres el substituiran pels seus fills per obtenir més precisió. Per evitar forts canvis a causa de l’elecció dels clústers, només utilitzarem el clúster si podem garantir que l’aproximació de l’error introduït pel clúster serà menor a un determinat llindar, inferior al perceptible. La llei de Weber diu que el canvi màxim perceptible en una senyal visual és pràcticament igual a un percentatge fixat de la senyal original. Sota les pitjors condicions, la vista humana només pot detectar aquests canvis per sota d’un 1%, tot i que, a la pràctica, aquest percentatge és més elevat. Utilitzant percentatges més elevats reduirem els costos d’avaluació de l’escena, però perdrem precisió, per tant és important escollir correctament aquesta dada.

Elecció dels lightcuts.

En funció del percentatge d’error que decidim tenir, necessitarem una estimació de la radiància total abans que poguem decidir si podem o no fer servir un clúster. Per a solventar aquesta dificultat, començarem sempre amb un “cut” molt poc agosarat, com pot ser el node arrel, i progressivament el refinarem fins a assolir l’error desitjat. Per cada node del cut calcularem una estimació d’iluminació i

l'error màxim que podem assolir. A cada pas de refinament tindrem en compte el node del cut amb un error a assolir més gran. Si el seu error a assolir (**E**) menys la seva iluminació estimada (**I**) és més gran que el percentatge d'error triat (**P**) multiplicat per l'actual iluminació aproximada total (**It**), el treurem del cut i el substituirem pels seus dos fills de l'arbre de llums. A la figura 3.4.4 podem veure la fórmula per determinar si l'error supera el llindar. A continuació repetirem els càlculs d'error a assolir i iluminació aproximada pels 2 fills i, finalment, recalcularem l'iluminació aproximada total. Altrament, el cut compleix el percentatge que ens havíem fixat i, per tant, ja hem acabat. Al cut obtingut se l'anomena "lightcut".

Les llums individuals presents al cut es calculen directament i per tant no tenen cap tipus d'error (no podem refinar més una llum que ja és una llum puntual).

A la figura 3.4.5 podem veure el pseudocodi que implementa l'elecció dels lightcuts.

$$(\mathbf{E} - \mathbf{I}) > (\mathbf{P} \cdot \mathbf{It})$$

Figura 3.4.4 – Fórmula per determinar si l'error supera el llindar

```
mètode buildLightCut(Punt p, NodeLlum node) ret Llista_de_Node{
      Llista_de_Node cutActual;
      cutActual.inserir(node);
      Real currentTotal =
      computeTotalIlluminationEstimate(p,cutActual);
      boolea fi = fals;
      mentre (!fi) {
            worst = findWorstErrorBound(objecte,cutActual)
            si ((worst.getErrorBound() -
            worst.getIlluminationEstimate()) >
            percentatge*currentTotal) llavors {
                  cutActual = eliminar(cutActual,worst)
                  computeBoundAndEstimateForChildren(worst,p)
                  cutActual.push_back(worst->fillDreta())
                  cutActual.push_back(worst->fillEsq())
                  currentTotal = updateTotalIllumination(worst,
                  currentTotal, p)
            }
            altrament {
                  fi = cert;
            }
      }

      retorna cutActual
}
```

Figura 3.4.5 – Pseudocodi corresponent a l'elecció dels lightcuts

Per una explicació més detallada d'aquest troç de codi anar a l'apartat 4.3.3.4, classe GeneradorArbre, on es comenta el seu funcionament.

3.5.- Implementant l'algoritme lightcuts

La nostra implementació suporta les llums puntuals: és a dir, aquelles que emeten llum equitativament en totes direccions.

Construïnt l'arbre de llums

L'arbre de llums agrupa els punts de llums en clústers. Ens interessaria maximitzar la qualitat del clúster que es crea (per exemple, combinant les llums que més s'assemblen en els seus termes de material, geometria i visibilitat). Aproximarem aquesta semblança agrupant les llums basant-nos en proximitat espacial.

Cada clúster de l'abre emmagatzemarà els seus dos fills, la seva llum representant, la seva intensitat total (suma de les intensitats dels seus dos fills) i la seva "bounding box". A la figura 3.5.1 es pot veure un exemple de clúster de llums, on les dues fulles, 1 i 2, s'agrupen escollint com a representant la 1.

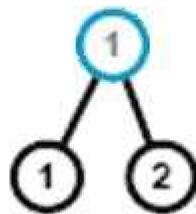


Figura 3.5.1 – Clúster de llums

Funcionament de l'algoritme constructor

Cada arbre és construït utilitzant un mètode voraç que progressivament combina parelles de llums i/o clústers. Primer de tot calculem tots els clústers de llums individuals que es poden generar, assignant a cada clúster una llum representant. Llavors, a cada passa, escollim la parella amb els 2 fills més propers. La resta de parelles que tinguin com a fill el representant del clúster escollit se li assignarà com a fill el nou clúster. En canvi, les parelles que tinguin com a fill la llum no representant del nou clúster seran descartades. D'aquesta manera ens assegurarem que una llum no es pugui repetir al cut i, alhora, fem que tota llum representant d'un clúster sigui, també, representant d'un dels seus fills. Per les llums individuals, elles mateixes són les seves pròpies representants.

El cost de càlcul de la construcció de l'arbre no pot ser sublinear en funció del nombre de llums, però el seu cost és insignificant a causa

de que només s'ha de calcular una vegada per tota l'escena, sempre que la distribució de llums no canvii.

3.6.- Càlcul dels errors màxims

Per a poder utilitzar l'algoritme de lightcuts necessitem poder calcular amb un cost raonablement baix els límits dels errors als clústers. Si calculem les cotes màximes dels termes material, geometria i visibilitat per un clúster, podem multiplicar aquests màxims per la intensitat del clúster i obtenir una cota pel resultat exacte i aproximat del clúster.

Càlcul del terme visibilitat.

El terme visibilitat per a un punt de llum serà sempre un o zero, ja que no tractarem amb superfícies semitransparents. Per estalviar-nos càlculs innecessaris i costosos suposarem, com a cota màxima, que totes les llums de l'escena són visibles. És a dir, el terme visibilitat valdrà sempre 1.

Càlcul del terme geometria.

Si suposem un terme $\mathbf{y}_i$, que representa la posició de la llum, i $\mathbf{x}$ representa la posició del punt a renderitzar, a la figura 3.6.1 podem veure la fórmula exacta per a calcular el terme.

$$G_i(\mathbf{x}) = \frac{1}{\|\mathbf{y}_i - \mathbf{x}\|^2}$$

Figura 3.6.1 – Fórmula exacta del terme geometria

Per a un clúster el càlcul és molt senzill, és la distància mínima entre el punt **x** i el punt **y**, que es tria com el punt de la bounding box més proper al punt **x**.

Càlcul del terme material.

El terme material és divideix en dues parts: el material difús i l'especular.

- El valor del material difús es calcula com el producte entre el color difús de la llum i el cosinus de l'angle que formen el vector entre la posició de la llum, i el punt i el vector normal del

punt. Com que el cosinus el podem acotar a 1, la cota del component difús resulta el valor del color difús.

- El valor del material especular és el valor del color especular de la llum multiplicat pel cosinus del producte entre els vectors V i R (V és el vector entre la posició de la càmera i el punt i R és el vector reflexió generat per la llum al punt) elevat a un factor anomenat rugositat. Si acotem el cosinus de la part especular, ens quedarà 1 elevat a la rugositat, que també podríem acotar a 1 (el seu valor oscilarà entre 0 i 1 en funció de lo rugós que sigui el material) i, per tant, ens quedarem amb el color especular.

En conclusió, el càlcul del terme material és ben senzill, simplement s'han de sumar els colors difús i especular de la llum.

3.7.- Càlcul del la iluminació estimada

El càlcul de la iluminació estimada, tant la total del cut actual com la d'un node, es calculen de manera similar a l'error màxim.

El terme visibilitat es calcula exactament igual, ja que no disposem ni d'ombres ni d'objectes semitransparents. El terme geometria també és casi igual, però enlloc d'utilitzar el punt més proper de la bounding box, utilitzem el mateix punt de la llum. Pel que fa al terme material, enlloc d'aproximar els cosinus els calculem.

4.- Disseny

Tot projecte disposa d'una sèrie de fases que permeten analitzar el conjunt de tasques i que cal realitzar per tal d'assolir els objectius proposats. Aquestes són les que es concreten en els apartats següents, és a dir, la descripció dels requeriments (punt 4.1), l'anàlisi (punt 4.2) i el disseny del sistema (punt 4.3) i finalment la implementació pròpiament dita amb els algoritmes convenients.

4.1.- Requeriments del sistema

En aquest projecte s'ha utilitzat una metodologia Orientada a Objectes basada en UML pel desenvolupament del software. La nomenclatura UML no defineix una metodologia concreta de desenvolupament de programari orientat a objectes, sinó que és un llenguatge que permet modelar, construir i documentar els elements que conformen un sistema orientat a objectes.

En aquest apartat recullirem, a grans trets, els principals objectius de l'aplicació juntament amb les funcionalitats que s'espera que l'usuari pugui arribar a realitzar.

Així, podem diferenciar entre dos grans grups de requeriments:

- **Requeriments Funcionals:** descriuen quins són els serveis o funcionalitats que ens oferirà l'aplicació independentment de l'implementació que haguem seguit.
- **Requeriments No Funcionals:** informen sobre les restriccions que venen imposades pel client o, en el nostre cas, pel sistema. Generalment són aquells requeriments considerats quantificables, com poden ser els terminis, els pressupostos, el programari utilitzat per desenvolupar, els recursos disponibles, la portabilitat, etc.

A continuació entrarem en detall amb els dos tipus de requeriments esmentats.

4.1.1.- Requeriments funcionals

Com hem comentat a l'apartat anterior, entenem com a requeriments funcionals les principals funcionalitats del programari a desenvolupar. A continuació es mostren els principals requeriments que s'han demanat a l'aplicació:

- **Iluminar una escena:** encarregar-se normalment de la iluminació de qualsevol escena.
- **Respectar un marge d'error:** no introduïr un error massa elevat al resultat final. Està en mans de l'usuari decidir quin percentatge d'error vol, però sempre s'ha de garantitzar que es

respectarà el percentatge escollit (lògicament, com més baix sigui aquest percentatge, més es ressentirà l'optimització).

- **Optimitzar el rendiment de l'escena:** donada una escena, garantitzem que, si el percentatge d'error introduït per l'usuari és prou alt, n'augmentarà significativament el rendiment.

4.1.2.- Requeriments no funcionals

Quan es porta a terme qualsevol tipus de projecte hem de posar especial atenció a tots aquells aspectes a tenir en compte a l'hora de dissenyar el sistema, més enllà dels requeriments funcionals detallats anteriorment. En general, s'afirma que els requeriments no funcionals d'un sistema són restriccions, és a dir, fan referència a requisits com el temps de resposta, la robustesa, la disponibilitat de recursos, el manteniment, la seguretat, les interfícies externes, etc. És quan totes aquestes condicions es compleixen quan el programa pot funcionar sense cap tipus de problema.

Pel que fa a la seguretat de l'aplicació, com que no fem servir cap tipus de dada confidencial o que mereixi una especial atenció (sempre parlant de seguretat) no té sentit implementar un sistema de protecció de dades.

L'aplicació ha estat implementada sobre un PC treballant amb Windows XP SP3, tot i que el motor gràfic utilitzat, OGRE3D, pot funcionar sobre altres plataformes com Linux o Mac OS X. Els motius pels quals s'ha escollit Windows i no Linux o Mac OS X són els següents:

- **Disponibilitat i comoditat:** tot i que Linux té moltes distribucions gratuïtes, el fet de ja tenir un Windows XP instal·lat en una màquina i tenir-ne més que suficients coneixements fan decantar la balança sobre Windows. A més, altres sistemes com Mac OS X queden descartats a causa dels nuls coneixements d'ús.
- **Programari utilitzat:** alguns programes dels utilitzats són privats i de pagament (Microsoft Visio o Microsoft Visual C++) i d'altres són lliures i gratuïts (OGRE3D, CEGUI), però el fet d'existir-ne versions de prova per estudiants o ja disposar-ne abans de començar el projecte han fet que utilitzar la plataforma Windows no hagi estat un problema.

La implementació del projecte s'ha dut a terme amb un Windows XP SP3. L'equip utilitzat durant el desenvolupament i el període de proves ha estat el següent:

Intel Core 2 Duo P8400 2.66Ghz.

3.2 GB de RAM

Targeta gràfica ATI Mobility Radeon HD 3470

A causa de la gran quantitat de llums i objectes que podem arribar a tenir en pantalla el processador, la memòria RAM i la targeta gràfica són elements clau per a un rendiment òptim.

Pel que fa als perifèrics d'entrada, el sistema desenvolupat no necessita cap perifèric concret, això dependrà de l'aplicació que l'utilitzi.
El sistema necessita tenir instal·lades les llibreries gràfiques Direct3D i/o OpenGL, ja que OGRE3D les utilitza per a renderitzar.

4.1.3.- Casos d'ús

Els casos d'ús descriuen el comportament d'un sistema des del punt de vista de l'usuari. Permeten definir els límits del sistema i les relacions entre aquest i el seu entorn. Podem dir que són descripcions de les funcionalitats del sistema independentment de la implementació. Estan basats en el llenguatge natural, de manera que puguin ser accessibles per tots els usuaris.

Amb els diagrames de casos d'ús que apareixen en aquest apartat podem observar les diferents connexions entre les funcionalitats del sistema definides als requeriments funcionals, a l'apartat 4.1.1.

A causa de que l'algoritme de Lightcuts està pensat per a ser utilitzat en una aplicació gràfica, l'usuari no hi intervé directament, sinó que intervé a través de l'aplicació en si. Per tant, es podria dir que qui intervé amb l'algoritme és la aplicació gràfica i, per tant, la única funcionalitat de l'algoritme de Lightcuts seria iluminar correctament l'escena. Però com ja s'ha comentat anteriorment, aquest procés es divideix en dues parts importants, que són la generació de l'arbre de llums i la creació del cut de llums per un punt determinat. A la figura 4.1.3.1 es pot veure el diagrama de casos d'ús del projecte.

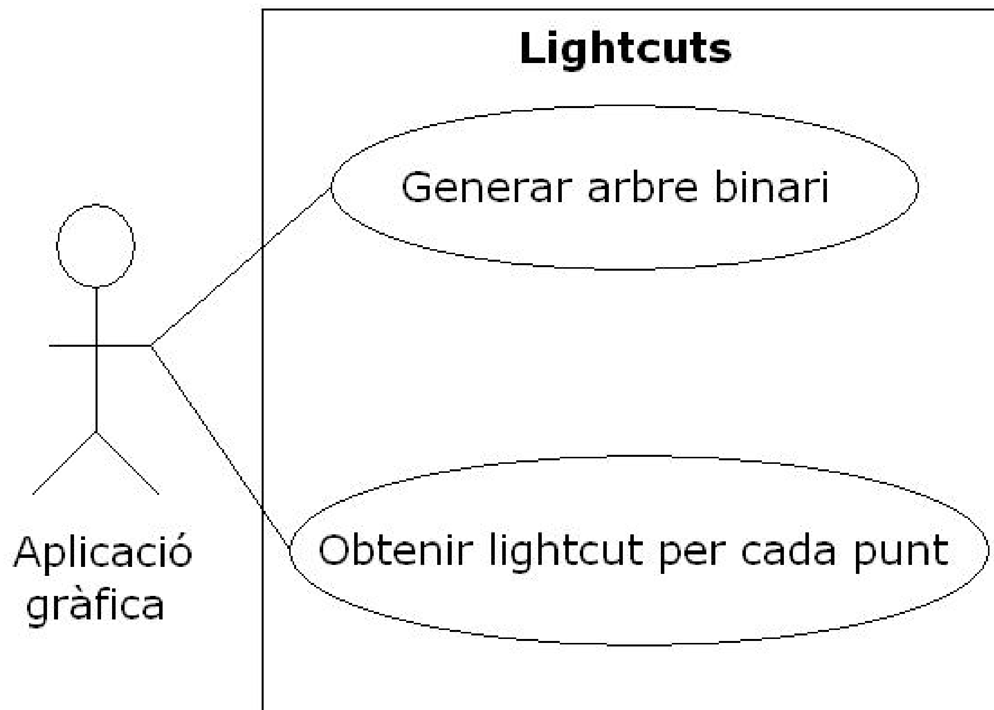


Figura 4.1.3.1 – Diagrama de casos d’ús del projecte.

A continuació mostrarem les dues fitxes de casos d’ús, una per cadascuna de les dues funcionalitats definides al diagrama de casos d’ús.

Cas d'ús 1: Generar arbre binari.

Fitxa	Cas d'ús "Generar arbre binari"
Funcionalitat	A partir de les llums que hi ha a l'escena generem un arbre binari de llums que ens permetrà escollir quines llums són més necessàries per cada punt.
Actor	Aplicació gràfica
Pre-condició	Que hi hagi llums definides a l'escena i algun objecte per iluminar.
Flux principal	1.- Generar tants nodes fulla com llums tinguem a l'escena. 2.- Crear una llista amb totes les combinacions possibles entre les llums de l'escena. 3.- Mentre quedin més d'una parella a la llista de parelles, fer: 3.1.- Obtenir la millor parella de la llista. 3.2.- Esborrar de la llista les parelles que continguin la llum no representant de la parella escollida. 3.3.- Substituir a la llista de les parelles totes les llums que continguin la llum representant de la parella per la parella triada. 4.- Fi.
Post-condició	S'haurà creat l'arbre binari de llums que contindrà totes les llums de l'escena.
Comentaris	Per una explicació més detallada veure l'apartat 4.3.3.4, de la classe GeneradorArbre, que és l'encarregada de dur a terme aquesta funció.

Cas d'ús 2: Obtenir lightcut per cada punt.

Fitxa	Cas d'ús "Obtenir lightcut per cada punt"
Funcionalitat	Per a cada punt de l'escena calculem quin conjunt de llums el poden iluminar més eficientment i mantenir un determinat marge d'error
Actor	Aplicació gràfica
Pre-condició	Que s'hagi generat l'arbre binari de llums.
Flux principal	1.- Agafem el primer node de l'arbre, l'arrel. 2.- N'afegim la llum al cut actual i evaluem l'error introduït a l'escena. 3.- Mentre l'error introduït sigui major que l'esperat, fem: 3.1.- Obtenim el millor node del cut actual. 3.2.- Si sobrepassa l'error màxim permès: 3.2.1.- Calculem l'error introduït per cadascun dels seus fills. 3.2.2.- Afegim els seus dos nodes fills al cut. 3.3.- Altrament, saltem al punt 4. 4.- Fi.
Post-condició	S'obtindrà una llista de les llums que han d'iluminar un objecte.
Comentaris	Per una explicació més detallada veure l'apartat 4.3.3.7, de la classe ListenerLlums, que és l'encarregada de dur a terme aquesta funció.

4.2.- Anàlisi del sistema

Un cop definits els requeriments mínims del sistema cal realitzar un estudi previ abans de començar a entrar en el disseny de l'aplicació. L'objectiu principal d'aquesta etapa és comprendre amb precisió les necessitats del sistema, és a dir, investigar el problema a resoldre però sense preocupar-se en trobar una solució.

Durant aquest procés es traduïran els requeriments comentats al capítol anterior a un llenguatge més formal a través de l'Enginyeria del Software. Així, podríem dir que el propòsit de l'anàlisi orientat a objectes és definir un diagrama de classes amb totes les classes, atributs i mètodes que es necessitaran per a resoldre el problema.

En aquesta memòria es persegueixen dos objectius. Per una banda mostrar les etapes complides durant el desenvolupament del projecte per tal de donar a conèixer els elements teòrics utilitzats. D'altra banda es vol documentar el sistema final. Per aquest motiu, per exemple, en aquesta etapa d'anàlisi presentem els diagrames en l'etapa final, és a dir, incorporant decisions de disseny. Aquest fet es dóna perquè l'orientació a objectes permet que els mateixos models siguin utilitzats de forma iterativa en les diferents etapes de la vida del software fent-los créixer des dels requeriments fins a la implementació.

4.2.1.- Diagrames d'activitat

El diagrama d'activitat és un tipus de diagrama UML que s'utilitza pel modelat dels aspectes dinàmics del sistema. Un diagrama d'activitat és fonamentalment un diagrama de flux entre activitats involucrades en un procés, generalment dins del marc d'un o diversos casos d'ús. Mostren l'ordre amb que s'executen les parts del procés i com depenen unes de les altres. No proporcionen informació del comportament d'un objecte ni de les col·laboracionis entre ells.

En aquest apartat es mostraran els diagrames d'activitat d'algunes de les operacions més complexes per deixar clar quin és el flux d'informació.

Diagrama d’activitat de Generar arbre binari.

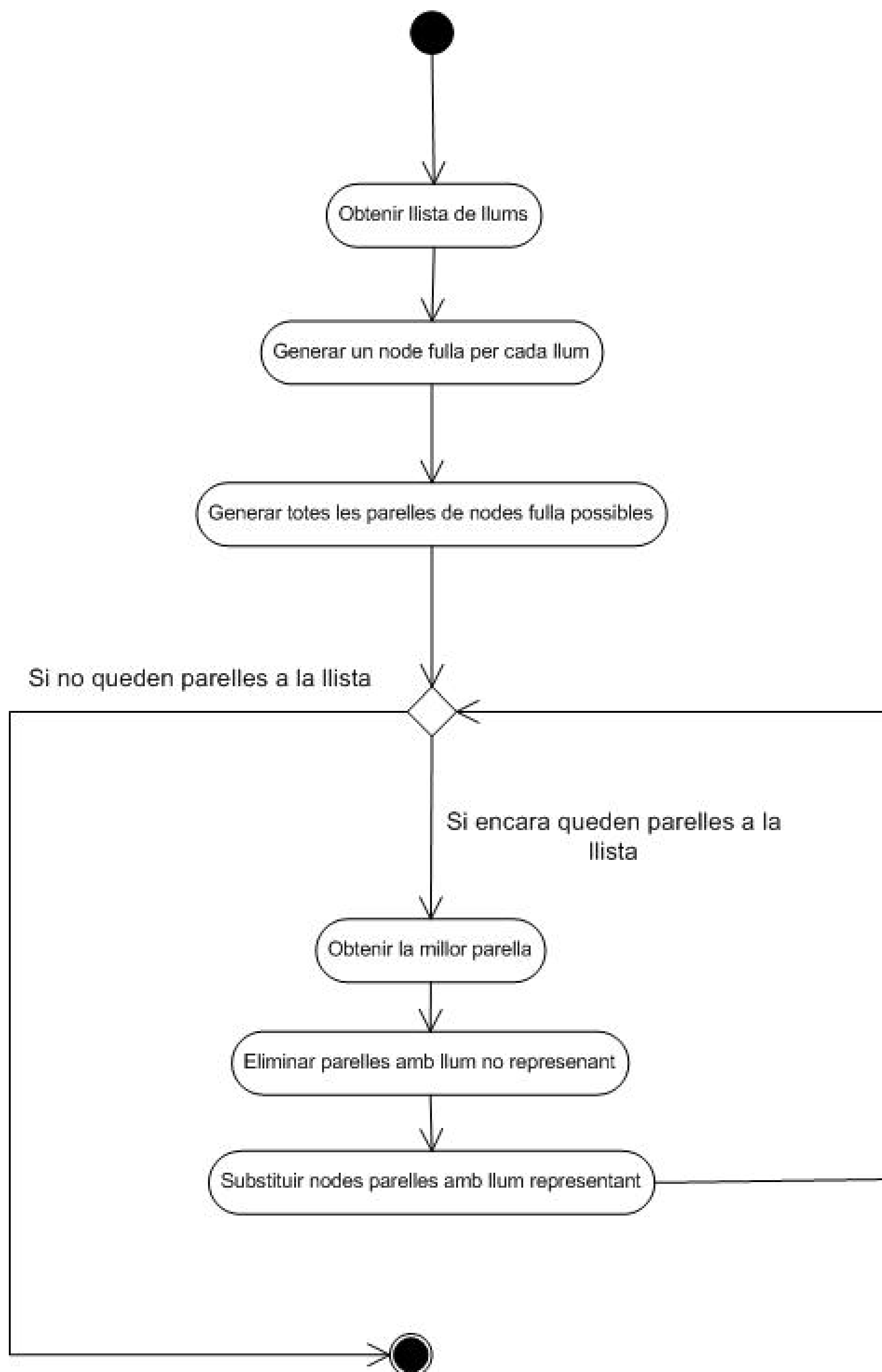
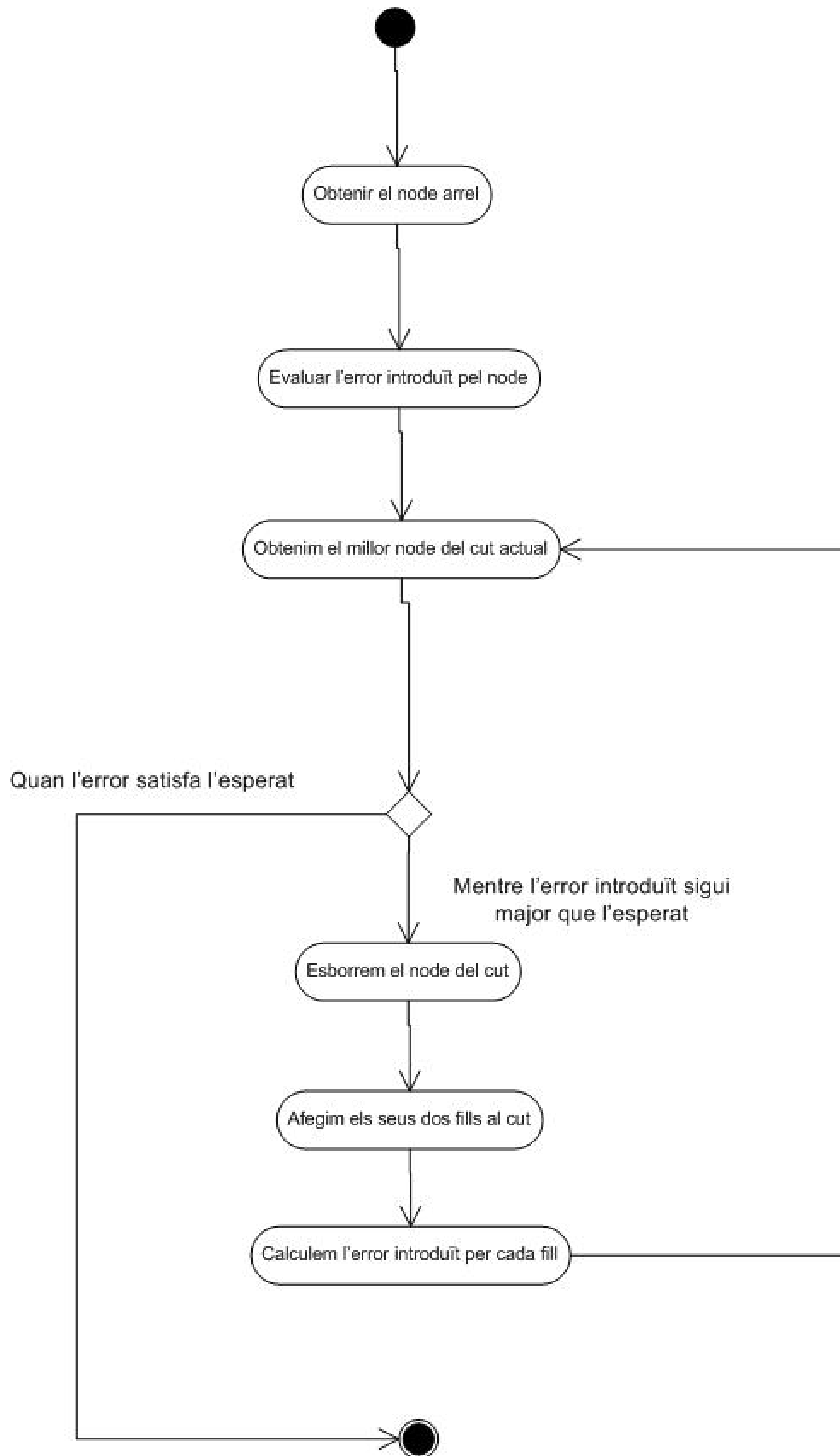


Diagrama d’activitat d’obtenir lightcut per cada punt



4.3.- Disseny del sistema

Després de les fases de descripció de requeriments i anàlisi del sistema, iniciem l'etapa de disseny del sistema. A l'anterior apartat, a l'anàlisi del sistema, ens hem centrat en la descripció de cadascuna de les funcionalitats de l'aplicació de forma abstracta. A partir dels resultats de l'análisi en aquesta etapa ens encarregarem de crear una solució conceptual que permeti implementar els requeriments del sistema desitjat. Ja que és l'última etapa teòrica haurem de definir les classes que necessitarem per a que puguin ser implementades.

En aquest apartat parlarem del disseny utilitzat en la interfície, dels patrons de disseny utilitzats i, finalment, de les classes que hem escollit.

4.3.1.- Disseny de la interfície

Tot programa orientat a l'usuari ha de tenir una interfície gràfica que li permeti interactuar amb l'aplicació de forma senzilla i intuitivament. Tot i que el mòdul de lightcuts està pensat per a utilitzar-se en una aplicació gràfica ja dissenyada, hem desenvolupat una presentació amb una senzilla interfície d'usuari que permet fer una petita demostració del funcionament del projecte.

Aquesta interfície s'ha desenvolupat utilitzant la llibreria de codi obert CEGUI (Crazy Eddie's Gui System) ja que es tracta de la llibreria que incorpora OGRE per defecte. A més, disposa d'una gran comunitat d'usuaris, molt bona documentació i moltes pàgines d'ajuda a la xarxa. A continuació es pot veure, a la figura 4.3.1.1, el logotip de la llibreria CEGUI.

Figura 4.3.1.1 – Logotip de la llibreria CEGUI

A l'apartat 5.3 farem una breu introducció a la llibreria, remarcant les característiques principals i les classes utilizades, però sense profunditzar massa, perquè no és l'objectiu del projecte.

4.3.2.- Patrons utilitzats

Singleton

Patró que serveix per a mantenir una sola instància d'un objecte i proveir accés global a aquesta. En el nostre cas l'utilitza la classe Root d'OGRE3D (veure apartat 2.3.1).

El funcionament del diagrama de classes, a la figura 4.2.1, és el següent: la classe Root emmagatzema una instància privada d'ella mateixa anomenada **ms_singleton**. Només es pot accedir a aquesta instància a través del mètode estàtic **getSingleton**, que garantitza que només hi hagi una instància de Root i que s'hi pugui accedir des de qualsevol lloc. A la figura 4.3.2.2 es pot veure un troç de codi que mostra com obtenir una instància de la classe Root i a la figura 4.3.2.3 la implementació del mètode getSingleton de la classe Root.

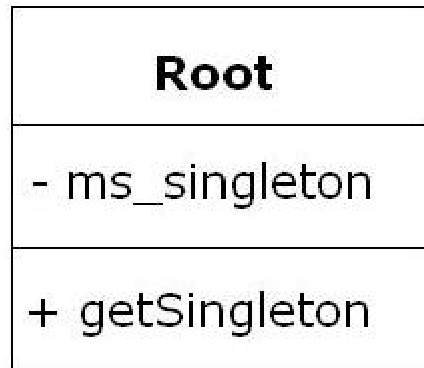


Figura 4.3.2.1 – Patró singleton de la classe Root

```
Root objecteRoot = Root.getSingleton();
```

Figura 4.3.2.2 –Pseudocodi que obté una instància de Root

```
mètode getSingleton() ret Root {
     si (ms_singleton == NULL) llavors{
          ms_singleton = new Root();
     }
     retorna ms_singleton;
}
```

Figura 4.3.2.3 –Pseudocodi corresponent al mètode getSingleton

Multiton

Patró que permet tenir una instància d'una classe associada a una clau determinada i accedir-hi globalment. És un concepte similar al

del singleton però enlloc de tenir una sola instància en podem tenir vàries, sempre controlades i associades a una clau.

En el cas d'OGRE tenim la classe Light, de la qual podem tenir vàries instàncies i cadascuna d'aquestes té un nom determinat, el qual fa alhora de clau. Aquestes instàncies es controlen des de la classe SceneManager, que seria la que té el patró Multiton implementat. El mateix passa amb les classes Entity o Camera, també controlades des de SceneManager. Es pot veure a la figura 4.3.2.4. A la figura 4.3.2.5 tenim un exemple de com generem una llum d'OGRE i a la figura 4.3.2.6 es troba el codi del mètode getLight, que implementa el patró Multiton a la classe SceneManager.

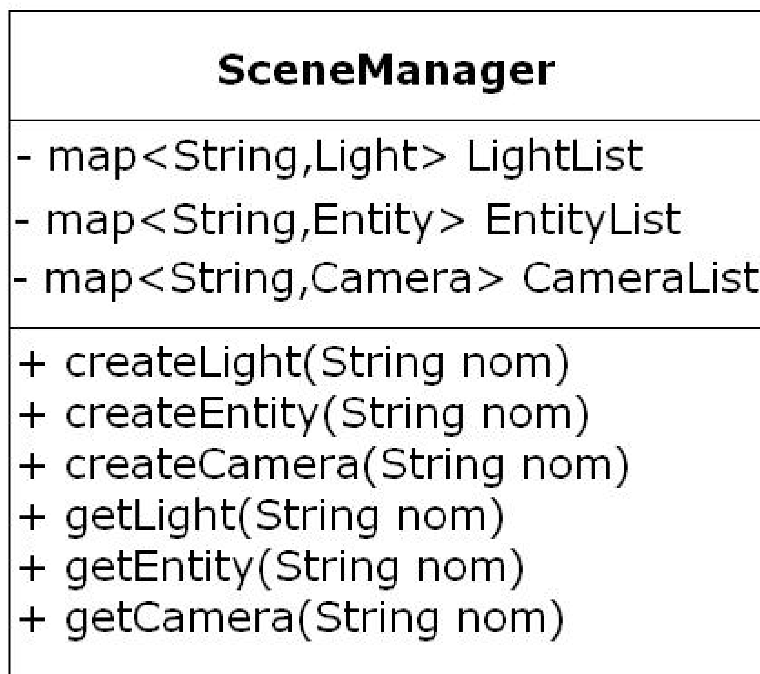


Figura 4.3.2.4 – Patró multiton de la classe SceneManager

```
//Suposem una instància de SceneManager anomenada mSceneMgr.
Light llum = mSceneMgr.getLight("LlumDeProva");
```

Figura 4.3.2.5 – Pseudocodi que obté de la classe SceneManager amb el patró Multiton una llum anomenada "LlumDeProva"

```
/*La classe SceneManager té un atribut anomenat "llistaLlums que conté
totes les llums, cadascuna identificada amb un String*/
mètode getLight(String nomLlum) ret Light {
      retorna llistaLlums.obtenirLlum(nomLlum);
}
```

Figura 4.3.2.6 – Pseudocodi corresponent al mètode getLight de la classe SceneManager

Listener

Aquest patró ens permet afegir funcionalitats a una altra classe dinàmicament sense haver-la de modificar. Tot i que a la comunitat d'Ogre se l'anomena Listener aquest patró és més conegut com Decorator.

Imaginem una classe **ClasseA** amb un mètode **void mètodeA()**. Si implementessim directament aquest mètode podríem definir-hi vàries funcionalitats, però sempre que volguessim afegir-ne hauríem de modificar el codi.

En canvi, si relacionem la **ClasseA** amb una altra classe (anomenada **Listener**) encarregada de la funcionalitat del mètode **mètodeA** aconseguirem fer més dinàmica la funció d'aquest mètode. Això a nivell de codi es pot fer tal i com mostra la figura 4.3.2.7.

```
ClasseA
     Atributs:
            Listener L

     Mètodes:

     mètode setListener(Listener lis) {
       L = lis
     }

     mètode mètodeA(){
       L.mètodeA()
     }
```

Figura 4.3.2.7 – Codi d'exemple del patró Listener

Com es pot comprovar al codi de la figura 4.3.2.7, a l'hora d'executar-se el codi del **mètodeA** deixem que sigui la classe **Listener** l'encarregada d'aquest mètode.

És el cas del mètode queryLights, de la classe MovableObject, que s'encarrega de decidir quines llums iluminen l'objecte. Si a un MovableObject li posem un Listener, amb el mètode setListener, quan cridem el mètode queryLights, la classe Listener serà l'encarregada

de decidir quines seran les llums que afectaran aquest objecte. A la figura 4.3.2.8 es pot veure el diagrama de classes de MovableObject i Listener.

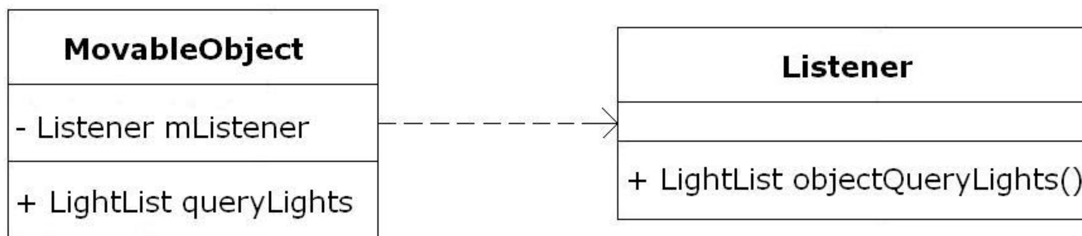


Figura 4.3.2.8 – Patró Listener de la classe MovableObject

Composite

El patró composite permet que un conjunt d'objectes sigui tractat de la mateixa manera, com si tots fossin del mateix tipus. A més, permet agrupar-los en forma d'arbre i tractar de la mateixa manera a objectes compostos i individuals.

Això ens és útil al projecte per a generar l'arbre binari de llums i poder distingir entre nodes individuals (objectes NodePuntual) i nodes compostos (objectes NodeGrup) a través de la interfície genèrica NodeLlum. Es pot comprovar l'estructura a la figura 4.2.5.

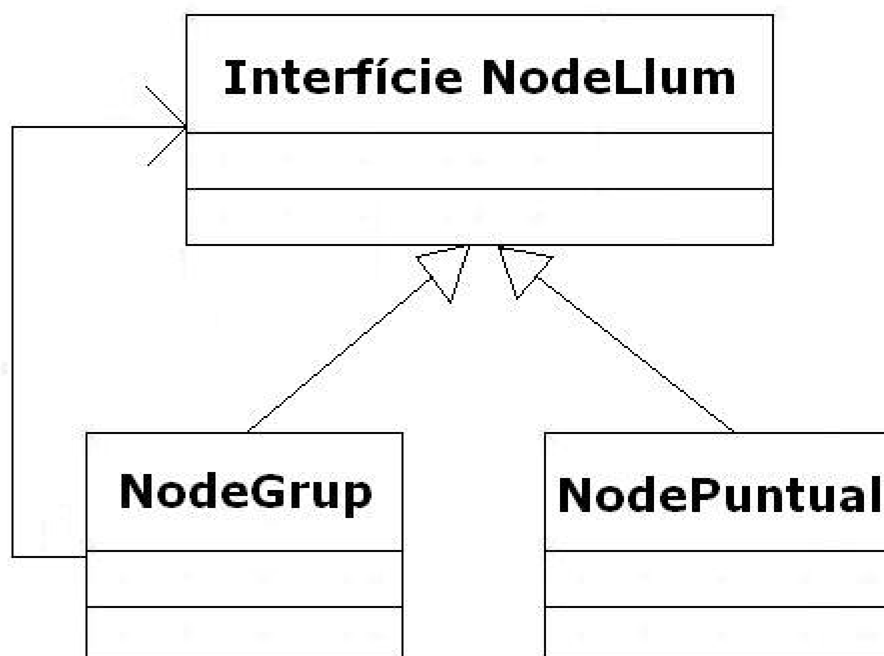


Figura 4.3.2.5 – Patró Composite de la classe NodeLlum

4.3.3.- Diagrama de classes

Els diagrames de classes són els més importants a l'hora de modelar un sistema orientat a objectes. Aquest tipus de diagrama s'utilitza per a poder analitzar o especificar els requeriments a l'hora de construir el model d'anàlisi. A més, mostra un conjunt de classes, interfícies i col·laboracions i totes les relacions entre elles.

Un diagrama de classes proporciona una visió estàtica del sistema a desenvolupar ja que, tal i com hem esmentat anteriorment, només mostra les classes interactuant, sense mostrar el que passa quan aquestes ho fan (això es pot veure als diagrames de seqüència, per exemple).

En aquest apartat es detallarà l'estructura de classes utilitzada, així com els components de cadascuna de les classes del sistema. A la figura 4.3.3.1 es pot veure el diagrama de classes del projecte. Les classes amb la paraula Ogre:: davant pertanyen al conjunt de classes d'OGRE3D, però juguen un paper important al projecte.

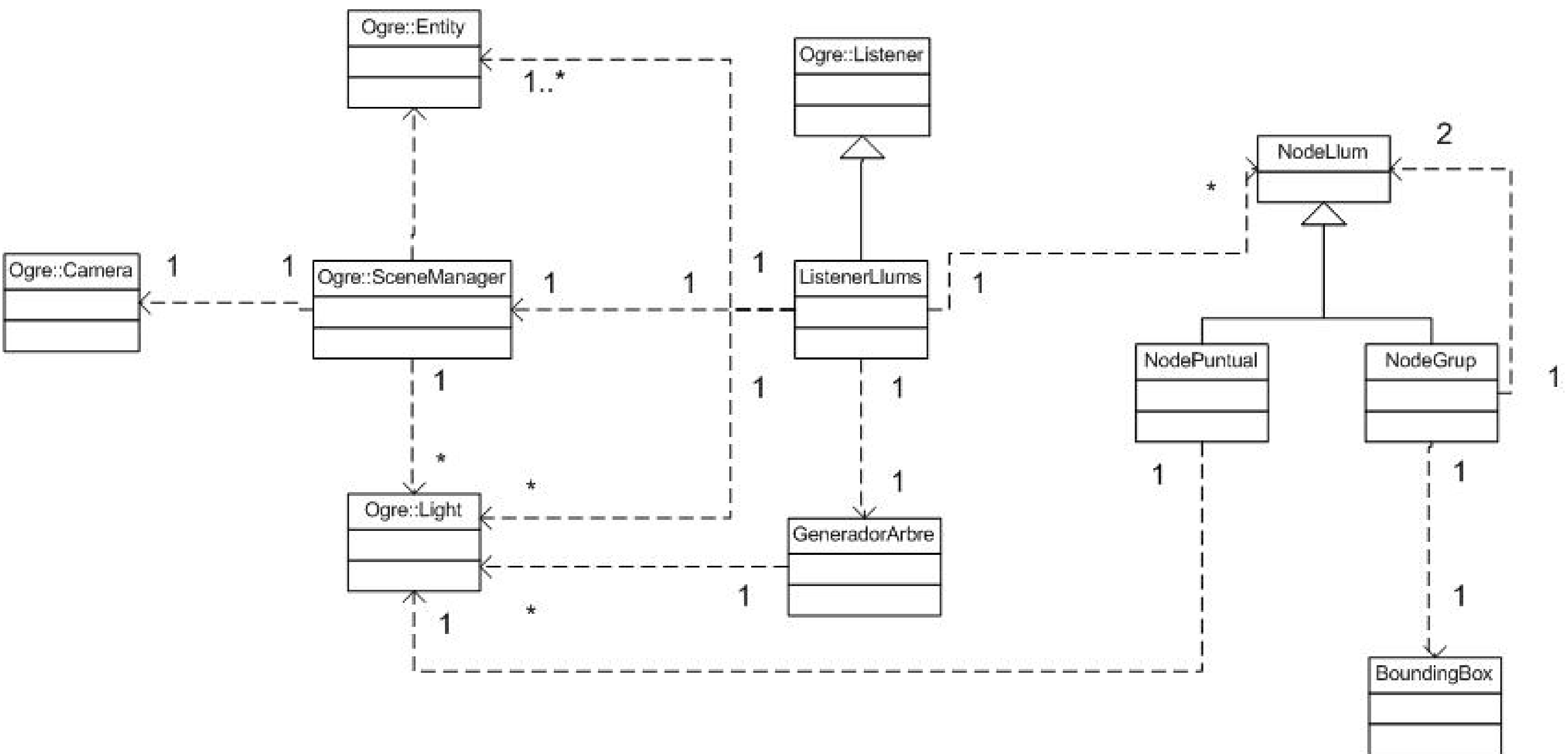


Figura 4.3.3.1 – Diagrama de classes del projecte

A continuació es farà un breu resum, en ordre alfabètic, de cadascuna de les classes que conforma el diagrama. Les classes pròpies de les llibreries d'OGRE ja esmentades a l'apartat 2.3 no seran comentades amb profunditat, només s'explicarà quina és la seva funcionalitat dins el projecte.

4.3.3.1.- Classe BoundingBox

Es podria traduir la paraula BoundingBox com "caixa envolupant". És la caixa generada per punts a l'espai, agafant les coordenades x,y,z més petites i més grans de cada punt per a formar una caixa amb 8 vèrtexs. És una classe que només s'utilitza a l'hora de generar el cut de llums, quan hem de calcular el component geometria de l'error màxim, tal i com s'explica a l'apartat 3.2. A la figura 4.3.3.1.1 es pot veure el diagrama de classes particular d'aquesta classe, amb tots els atributs i mètodes pertinents. Els mètodes menys importants, com per exemple aquells per tractar amb els atributs, no apareixen, per evitar codi innecessari.

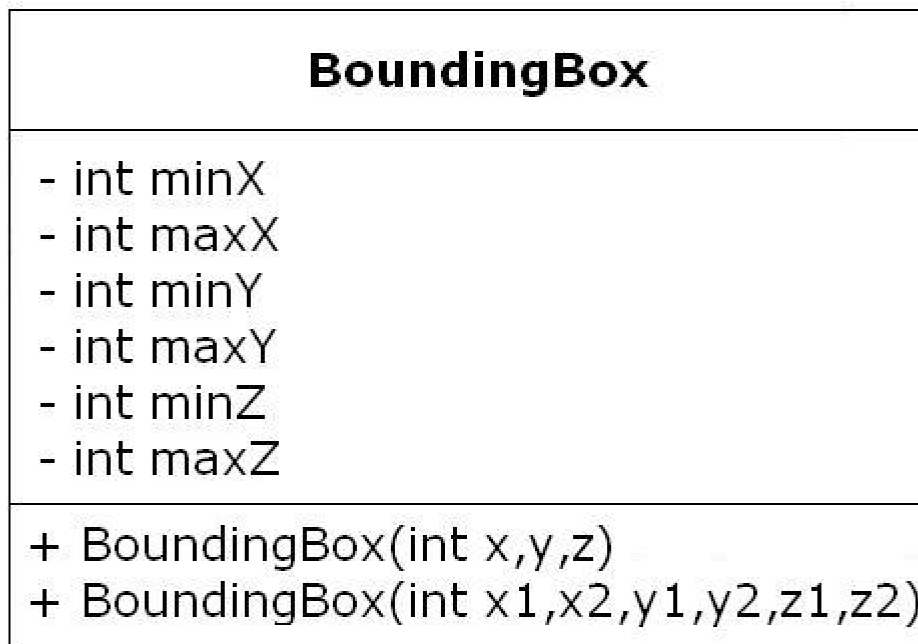


Figura 4.3.3.1.1 – Classe BoundingBox

- **BoundingBox(int x1,x2,y1,y2,z1,z2):** constructor que genera una BoundingBox amb valors minX=x1, maxX=x2, minY=y1 i així successivament.

- **BoundingBox(int x,y,z):** constructor que fa que els valors mínims i màxims valguin el mateix (és a dir, minX i maxX valdran x, i el mateix amb les y i les z). Cridar aquest mètode només té sentit per a objectes tipus NodePuntual, que

representen una llum individual i la caixa que generen és un punt.

4.3.3.2.- Classe Ogre::Camera

Aquesta és la classe d'Ogre encarregada de generar la càmera amb la que veurem l'escena resultant. Hereta de la classe MovableObject i és generada per la classe SceneManager. A l'apartat 2.3.7 es parla més extensament del funcionament de la classe. A la figura 4.3.3.2.1 es pot veure el diagrama de classes de la classe Camera amb els mètodes més importants.

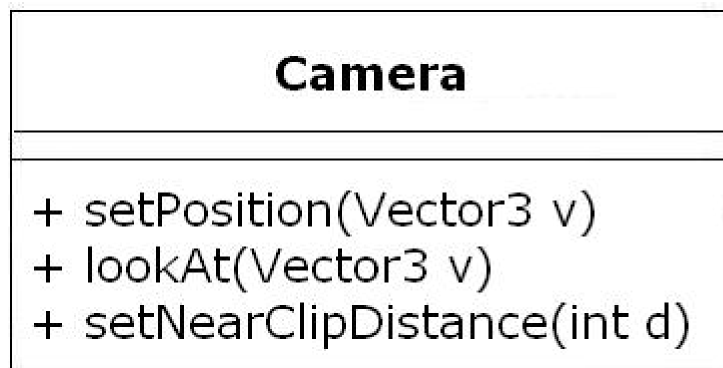


Figura 4.3.3.2.1 – Classe Camera

A continuació es farà un breu resum dels mètodes de la classe Camera:

- **setPosition:** mètode que, passant-li un vector de coordenades tridimensionals (classe Vector3 d'Ogre), li assigna una posició a l'espai a la càmera.
- **lookAt:** mètode que, a través d'un vector de coordenades tridimensionals, dóna a la càmera un punt al que enfocar.
- **setNearClipDistance:** aquest mètode serveix per a dir fins a quina distància ha de ser funcional la càmera que estem fent servir.

4.3.3.3.- Classe OGRE::Entity

Amb aquesta classe instanciem cadascun dels objectes de l'escena, on cada objecte correspon a un polígon. El més important és que hereta de la classe d'OGRE MovableObject i, per tant, se li pot assignar un Listener passant-lo per paràmetre amb el mètode setListener. Aquest Listener, comentat a l'apartat 4.3.2, és l'encarregat d'indicar a la classe quines són les llums que volem que

iluminin l'entitat. A la figura 4.3.3.3.1 es pot veure la classe juntament amb els mètodes més importants.

Figura 4.3.3.3.1 – Classe Entity

A continuació farem una breu descripció dels mètodes de la classe Entity:

- **setMaterialName:** mètode que ens permet assignar-li a l'entitat un material identificat per l'atribut name.
- **setCastShadows:** mètode que serveix per a decidir si l'entitat projectarà o no llums, en funció del booleà passat per paràmetre.
- **setListener :** mètode que ens permet assignar el Listener passat per paràmetre a la classe Entity.

4.3.3.4.- Classe GeneradorArbre

Classe encarregada de generar l'arbre binari de llums. Només li fa falta una llista amb llums (objectes Light d'OGRE) i pot retornar un node apuntant a l'arrel de l'arbre binari que ha generat. A la figura 4.3.3.4.1 es pot veure el diagrama de classes de la classe GeneradorArbre amb els atributs i mètodes més importants.

GeneradorArbre	
- Llista_de_Llum	llistaDeLlums
- Llista_de_NodeLlum	parellesLlums
- NodeLlum	arbre
+ GeneradorArbre(LlistaLlums llista)	
- escollirRepresentant	
- ordenarParelles	
- eliminarNoRepresentant	
- substituirRepresentant	

Figura 4.3.3.4.1 – Classe GeneradorArbre

Els atributs són:

- **llistaLlums:** una llista d'objectes Light.
- **parellesLlums:** una llista de NodeLlum, que representa les parelles generades.
- **arbre:** un NodeLlum que apuntarà a l'arrel de l'arbre un cop acabat el procés de generar l'arbre.

El mètode principal és el mateix constructor, GeneradorArbre, que amb la mencionada llista de llums genera l'arbre binari. A la figura 4.3.3.4.2 es pot veure el codi corresponent al constructor de GeneradorArbre.

```
constructor GeneradorArbre(Llista_de_Llums llums) {
       Llistallums = llums;
       NodeLlum aux;
       BoundingBox auxBB;

       per (i de 0 fins a llistaLlums.mida() pas 1) fer{
              auxBB = new BoundingBox(/*paràmetres1*/);
              aux = new NodePuntual(/*paràmetres2*/);
              parellesLlums.afegir(aux);
       }
       parellesLlums = combinarParelles();
       mentre (parellesLlums.mida()>1) fer {
              aux = getMillorParella();
              eliminarNoRepresentant(aux.llumNoRepresentant());
              substituirRepresentant(aux);
       }

       arbre = parellesLlums[0];
}
```

Figura 4.3.3.4.2 – Pseudocodi corresponent al constructor GeneradorArbre

El codi del constructor **GeneradorArbre** es pot dividir en dues parts. Primerament, un bucle per crear les fulles de l'arbre a partir de la llista de llums **llistaLlums** i posteriorment generem totes les combinacions de parelles possibles amb el mètode **combinarParelles**. A la segona part hi ha un altre bucle que a cada iteració farà el següent:

1.- Triar la millor parella, **aux**, amb el mètode **getMillorParella** (en funció de la distància entre les llums que la componen).

2.- Eliminar les parelles que tinguin com a fill la llum no representant d'**aux**.

3.- Substituir els nodes fills que tinguin com a representant la mateixa llum que la parella triada.

Per a sortir d'aquest bucle el nombre de parelles de la llista parellesLlums haurà de ser igual a 1. Aquesta parella restant serà el node arrel de l'arbre, i la resta de llums penjaran d'ell. Per ilustrar millor funcionament d'aquest mètode es pot seguir el diagrama de seqüència de la figura 4.3.3.4.3.

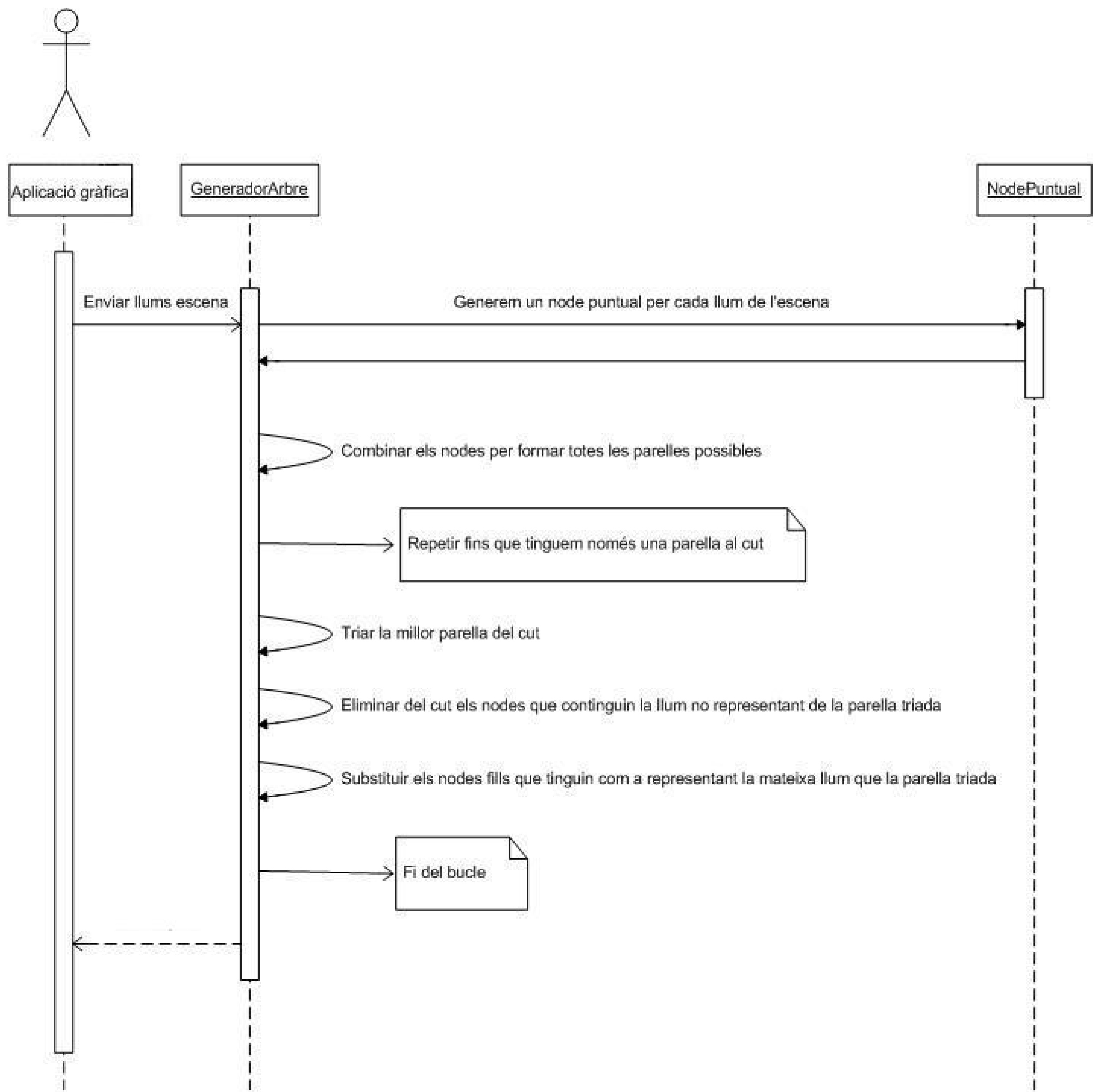


Figura 4.3.3.4.3 – Diagrama de seqüència del constructor GeneradorArbre

El diagrama de seqüència que es pot veure a la figura 4.3.3.4.3 fa el següent: primerament l'aplicació gràfica envia les llums de l'escena al constructor GeneradorArbre. A continuació generem tants objectes NodePuntual com llums té l'escena. Seguidament, combinem els nodes per formar totes les parelles possibles. Entrem en un bucle on, triem la millor parella del cut, eliminem del cut els nodes que contenen la llum no representant de la parella triada i substituim els nodes fills que tinguin com a representant la mateixa llum que la parella triada. Per sortir del bucle ha de quedar només una llum al cut.

A continuació comentarem els mètodes privats que hem separat per no recarregar el codi, que s'encarreguen de feines concretes com:

- **combinarParelles:** mètode encarregat de trobar totes les combinacions de parelles possibles entre les llums de l'atribut **parellesLlums**. Aquest mètode només es farà servir una vegada, quan cridem el constructor de la classe GeneradorArbre, i tots els NodeLlum continguts a l'atribut parellesLlums seran NodePuntuals. Per a fer-ho, combinarem cada NodeLlum de l'atribut amb la resta de llums (sense repetir combinacions) generant un clúster de NodeGrup que contindrà els dos NodePuntual. Podem veure el codi a la figura 4.3.3.4.4.

```
mètode combinarParelles() ret Llista_de_NodeLlum {
      Llista_de_NodeLlum combinacio;
      NodeLlum * aux;

      per (i de 0 fins a parellesLlums.mida() pas 1) fer{
            per (j de i+1 fins a parellesLlums.mida() pas 1) fer{
                  aux = new NodeGrup(/*paràmetres…*/)
                  aux.computeBoundingBox();
                  combinacio.afegir(aux);
            }
      }

      retorna combinacio;

}
```

Figura 4.3.3.4.4 – Codi corresponent al mètode combinarParelles

El codi del mètode combinarParelles té 2 bucles que generaran totes les combinacions de parelles possibles amb nombres de 0 fins a la mida de parellesLlums. El primer bucle s'encarregarà de generar un nombre i el segon l'altre, de manera que el segon sempre començarà a generar una posició més enllà que el primer. Per tant, si suposem un atribut parellesLlums de mida 4, les combinacions resultants seran (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4).

- **ordenarParelles:** ordena la llista de NodeLlum que li passem per paràmetre en funció de la distància entre aquestes.

Utilitzem el mètode d'ordenació de la bombolla, tal i com es pot veure a la figura 4.3.3.4.5, que mostra el codi del mètode.

```
mètode ordenarParelles(NodeLlum combinacioLlums) ret Llista_de_NodeLlum{
      Llista_de_NodeLlum combinacio = combinacioLlums;

      per (i de 1 fins a combinacio.mida() pas 1) fer {
            per(j de 0 fins a combinacio.mida() - 1 pas 1) fer {
                  si (combinacio[j]->getDistancia() > combinacio[j+1]->
                  getDistancia()) llavors{
                        intercanviem(i,j);
                  }
            }
      }
      retorna combinacio;
}
```

Figura 4.3.3.4.5 – Codi corresponent al mètode d'ordenació ordenarParelles

- **eliminarNoRepresentant**: mètode encarregat d'eliminar els NodeLlum de l'atribut parellesLlums que continguin la llum passada per paràmetre (sigui o no representant). Per a fer-ho, recorrem tota la llista i, per cada NodeLlum, mirem si un dels seus dos fills té com a representant el que li hem passat per paràmetre. Si és el representant de la llum no el guardem al cut actual, altrament, si. A la figura 4.3.3.4.6 es pot veure el codi corresponent a aquest mètode.

```
mètode eliminarNoRepresentant(int noRep) {
      Llista_de_NodeLlum llistaAux;
      NodeLlum parellaAux;
      per (i de 0 fins a parellesLlums.mida() pas 1) fer{
            parellaAux = parellesLlums[i];
            si ((parellaAux.esNodeGrup() i
            parellaAux.fillDreta().llumRepresentant() != noRep i
            parellaAux.fillEsq().llumRepresentant() != noRep))
            llavors{
                  llistaAux.afegir(parellaAux);
            }
      }
      parellesLlums = llistaAux;
}
```

Figura 4.3.3.4.6 – Codi corresponent al mètode d'ordenació eliminarNoRepresentant

Com es pot veure al codi del mètode **eliminarNoRepresentant,** fem un bucle per recorre totes les llums del cut actual. Si cap dels fills de la llum tractada té com

a representant el que hem passat per paràmetre amb l'enter **noRep**, el guardem a una llista temporal **llistaAux**. Finalment substituim l'atribut parellesLlums per la llista temporal llistaAux.

- **substituirRepresentant:** aquest mètode rep com a paràmetre un objecte de NodeLlums. Tots els NodeLlum de l'atribut parellesLlums que tinguin com un dels fills el representant de la parella passada per paràmetre es substituiran per aquesta parella. A la figura 4.3.3.4.7 es pot veure el codi corresponent a aquest mètode.

```
mètode substituirRepresentant(NodeLlum llumRep) {
	NodeLlum parellaAux;
	per (i de 0 fins a parellesLlums.mida() pas 1) fer{
		parellaAux = parellesLlums[i];
		si (parellaAux.esNodeGrup()) llavors {
			si (parellaAux.fillDreta().llumRepresentant() ==
			llumRep.llumRepresentant()) llavors {
				parellaAux.setLightDreta(llumRep);
				parellaAux.recalcularIntensitat();
			}
			si (parellaAux.fillEsq().llumRepresentant() ==
			llumRep.llumRepresentant()) llavors {
				parellaAux.setLightEsq(llumRep);
				parellaAux.recalcularIntensitat();
			}
		}
	}
}
```

Figura 4.3.3.4.7 – Codi corresponent al mètode substituirRepresentant

El mètode **substituirRepresentant** té un bucle que recorre totes les llums de la llista **parellesLlums**. Per cada llum, comprova si algun dels seus fills té com a representant el mateix que el NodeLlum **llumRep** que hem passat per paràmetre. Si és així, substituim aquest fill pel node llumRep. Aquest mètode és el que, a cada iteració del bucle principal de GeneradorArbre, va creant l'arbre binari de llums.

4.3.3.5.- Classe OGRE::Light

Classe que representa un punt de llum a l'Ogre. La fem servir per generar cadascuna de les llums d'una l'escena, que posteriorment es guarden en una llista i es passen a la classe GeneradorArbre.

Guarda informació com la posició de la llum, el tipus de llum (com ja hem comentat a l'apartat 2.3.8 només utilitzem les llums puntuals o LT_POINT), el color difús i el color especular. Té moltes altres

característiques que no hem fet servir al projecte, i per tant no enumerarem. A la figura 4.3.3.5.1 es pot veure el diagrama de classes de la classe Light.

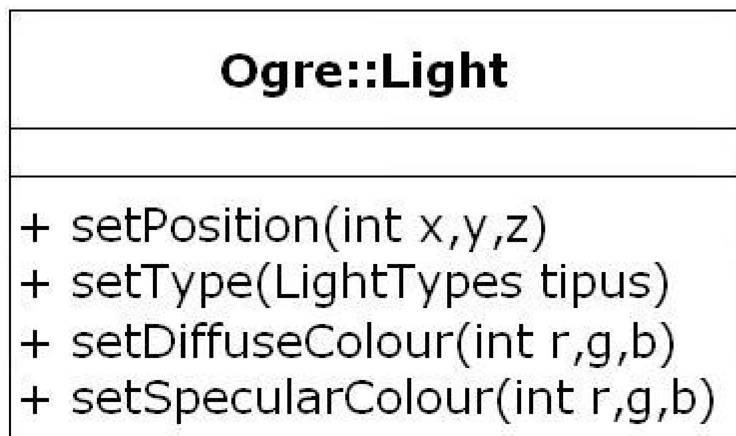


Figura 4.3.3.5.1 – Classe Light

El mètode **setPosition** serveix per a definir la posició de la llum, a partir de les coordenades tridimensionals x, y, z. El mètode **setType** serveix per a adjudicar un tipus de llum a la llum, d'entre els tres tipus mencionats a l'apartat 2.3.8. Finalment, els mètodes **setDiffuseColour** i **setSpecularColour**, permeten definir els colors difús i especular de la llum, a través de les components RGB.

4.3.3.6.- Classe OGRE::Listener

Classe abstracta que afegeix funcionalitat a un objecte de tipus MovableObject (com Entity, en el nostre cas). En concret permet que s'executi el mètode objectQueryLights, que hem de sobreescriure, i retorna una llista de llums que són les que afectaran a l'objecte MovableObjct al que estigui lligat el Listener. El nostre Listener s'anomena ListenerLlums i a continuació el comentarem. A la figura 4.3.3.6.1 es pot veure el diagrama de classes de la classe Listener.

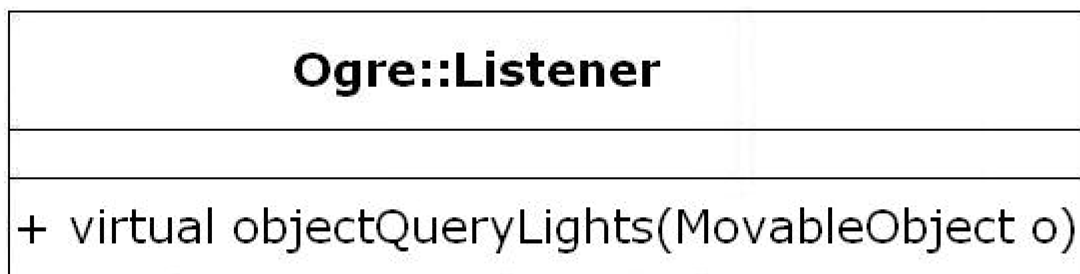


Figura 4.3.3.6.1 – Classe Listener

4.3.3.7.- Classe ListenerLlums

Classe que implementa la classe Listener d'Ogre. És l'encarregada de recorrer l'arbre binari de llums i decidir quin lightcut li correspon a l'objecte que té assignat. Per tant, tenim un objecte de la classe ListenerLlums per cada punt de l'escena (és a dir, un per cada Entity).

A l'hora d'instanciar-ne un objecte, li passarem un booleà que dirà si volem que recorri l'arbre de llums i trobi un cut o simplement volem que utilitzi totes les llums de l'escena (és a dir, les fulles de l'arbre de llums). Això serà útil si volem comparar una mateixa escena utilitzant l'algoritme de Lightcuts o sense utilitzar-lo. A més, també li passarem un objecte GeneradorArbre i un objecte SceneManager, que guardarem per poder-hi accedir més endavant. A la figura 4.3.3.7.1 es pot veure el diagrama de classes de la classe ListenerLlums.

```
ListenerLlums
-------------------------------------------------------------
- GeneradorArbre          arbre
- SceneManager            mSceneMgr
- Llista_de_NodeLlum      llistaNodesLlums
- Llista_de_Light         llistaLlums
- booleà                  actiu
-------------------------------------------------------------
+ ListenerLlums (GA tree, SM manager, booleà act)
- buildLightCut (MO objecte, NodeLlum node)
- calcularCota (MO objecte, NodeLlum node)
- illuminationEstimate (MO objecte, NodeLlum node)
- computeTotalIlluminationEstimate (MO objecte,
  Llista_de_NodeLlum cutActual)
- findWorstErrorBound (MO objecte,
  Llista_de_NodeLlum cutActual)
- computeBoundAndEstimateForChildren (NodeLlum
  node, MO objecte)
- updateTotalIllumination (NodeLlum node, Real
  currentTotal, MO objecte)
- sonTotFulles (Llista_de_NodeLlum lightcut)

//GA = GeneradorArbre
//SM = SceneManager
//MO = MovableObject
```

Figura 4.3.3.7.1 – Classe ListenerLlums

Els atributs que conté són els següents:

- **arbre**: referència de la classe GeneradorArbre necessària per obtenir la llista de llums (objectes Light).
- **mSceneMgr**: objecte de la classe SceneManager que necessitem per obtenir informació de les llums i de la càmera per a poder fer càlculs com la iluminació estimada i l'error màxim.
- **llistaNodesLlums**: llista de nodes que utilitzarem per generar el cut de llums.
- **llistaLlums**: llista d'objectes Light que retornarem a través del mètode objectQueryLights. Pot ser que les llums a mostrar

siguin una part del conjunt (un lightcut) o tot el conjunt, en funció del que l'usuari decideixi.

- **actiu**: variable booleana que indica si hem de mostrar les llums del lightcut o totes les llums de l'escena.

El mètode més important d'aquesta classe s'anomena **buildLightCut** i conté tota la lògica que permet recorrer l'arbre binari de llums i trobar un lightcut adient al punt i l'arbre de que disposem. A la figura 4.3.3.7.2 podem veure el troç de codi que correspon a aquest mètode.

```
mètode buildLightCut(Punt p, NodeLlum node) ret Llista_de_Node{
	Llista_de_Node cutActual;
	cutActual.inserir(node);
	Real currentTotal =
	computeTotalIlluminationEstimate(p,cutActual);
	boolea fi = fals;
	mentre (!fi) fer {
		worst = findWorstErrorBound(objecte,cutActual)
		si ((worst.getErrorBound() -
		worst.getIlluminationEstimate()) >
		percentatge*currentTotal) llavors {
			cutActual = eliminar(cutActual,worst)
			computeBoundAndEstimateForChildren(worst,p)
			cutActual.push_back(worst->fillDreta())
			cutActual.push_back(worst->fillEsq())
			currentTotal = updateTotalIllumination(worst,
			currentTotal, p)
		}
		altrament {
			fi = cert;
		}
	}

	retorna cutActual
}
```

Figura 4.3.3.7.2 – Codi corresponent al mètode buildLightCut

El funcionament del mètode és el següent:

1. Escollim el node arrel de l'arbre com a primer node a avaluar.
2. Calculem el seu error màxim i la iluminació estimada, juntament amb l'illuminació total de l'escena (amb aquest primer node valdrà el mateix que la seva estimació d'iluminació).
3. Entrem al bucle.
4. Trobem el node amb error màxim més gran (en la primera volta serà sempre el node arrel).

5. Si la diferència entre l'error màxim i la iluminació estimada és major que la iluminació estimada total multiplicada pel percentatge triat, fem:
    5.1. Eliminem del cut actual el node triat.
    5.2. Calculem l'error màxim i la iluminació estimada pels dos nodes fills del node triat.
    5.3. Guardem al cut actual els dos nodes fills.
    5.4. Actualitzem la iluminació estimada total amb les dades dels nodes fills.
6. Altrament, sortim del bucle.
7. Retornem el cut resultant.

A continuació detallarem altres mètodes importants de la classe ListenerLlums:

- **calcularCota**: aquest mètode calcula l'error màxim d'un NodeLlum i el MovableObject passats per paràmetre seguint la fórmula detallada a l'apartat 3.7. Per més detalls consultar el capítol 6, implementació.
- **illuminationEstimate**: mètode encarregat de calcular la iluminació estimada d'un NodeLlum i el MovableObject passats per paràmetre seguint la fórmula de l'apartat 3.8. Es pot trobar més informació al capítol d'implementació.
- **computeTotalIlluminationEstimate**: mètode que calcula la iluminació estimada total del cut actual. A la figura 4.3.3.7.3 es pot veure el codi corresponent a aquest mètode.

```
mètode computeTotalIlluminationEstimate(MovableObject objecte,
Llista_de_NodeLlum cutActual) ret Real{
	Real aux = 0;
	per (i de 0 fins a cutActual.mida() pas 1) fer{
		aux += illuminationEstimate(objecte,cutActual[i]);
	}
	retorna aux;
}
```

Figura 4.3.3.7.3 – Pseudocodi corresponent al mètode computeTotalIlluminationEstimate.

El bucle del mètode **computeTotalIlluminationEstimate** recorre totes les llums del cut **cutActual** i, per cadascuna, en calcula la iluminació estimada amb el mètode

illuminationEstimate. Finalment, retorna el sumatori de les iluminacions estimades que hem calculat.

- **findWorstErrorBound**: mètode que busca al cut actual el node amb error màxim més gran. A la figura 4.3.3.7.4 es pot veure el codi corresponent a aquest mètode.

```
mètode findWorstErrorBound(MovableObject objecte, Llista_de_NodeLlum
cutActual) ret NodeLlum{
      Llista_de_NodeLlum cutAux;
      per (a de 0 fins a cutActual.mida() pas 1) fer {
            si (cutActual[a].esNodeGrup()) llavors {
                  cutAux.inserir(cutActual[a]);
            }
      }
      per (i de 1 fins a cutAux.mida() pas 1) fer {
            per(j de 0 fins a cutAux.mida() - 1 pas 1) fer {
                  si (cutAux[j].getErrorBound() <
                  cutAux[j+1].getErrorBound()) llavors {
                        intercanviem(j,j+1);
                  }
            }
      }
      si (cutAux.mida()>0) llavors retorna cutAux[0];
      altrament retorna NULL;
}
```

Figura 4.3.3.7.4 – Pseudocodi corresponent al mètode findWorstErrorBound.

El mètode **findWorstErrorBound** es divideix en dues parts. Primer de tot, generem una llista de NodeLlum on només ens guardem els NodeGrup, ja que no té sentit intentar refinar un NodePuntual, perquè no té fills. A continuació ordenem, amb l'algoritme de la bombolla, la nova llista **cutAux** resultant. Finalment, si la llista té algun node, en retornem la primera posició, que serà la que es correspongui amb el NodeLlum amb error màxim més gran.

- **computeBoundAndEstimateForChildren**: aquest mètode calcula l'error màxim i la iluminació estimada pels fills del node passat per paràmetre. A la figura 4.3.3.7.5 es pot veure el codi corresponent a aquest mètode.

```
mètode computeBoundAndEstimateForChildren(NodeLlum node, MovableObject
objecte) {
      node.fillDreta().setErrorBound(calcularCota(
      objecte,node.fillDreta()));
      node.fillDreta().setIlluminationEstimate(
      illuminationEstimate(objecte,node.fillDreta()));

      node.fillEsq().setErrorBound(calcularCota(
      objecte,node.fillEsq()));
      node.fillEsq().setIlluminationEstimate(
      illuminationEstimate(objecte,node.fillEsq()));
}
```

Figura 4.3.3.7.5 – Pseudocodi corresponent al mètode computeBoundAndEstimateForChildren.

El mètode **computeBoundAndEstimateForChildren** és bastant senzill, calcula l'error màxim i la iluminació estimada pel fill dret i els canvia, i llavors fa el mateix amb el fill esquerra.

- **updateTotalIllumination**: mètode que actualitza la iluminació estimada total amb les dades dels dos nodes fill del node passat per paràmetre. A la figura 4.3.3.7.6 es pot veure el codi que implementa aquest mètode.

```
mètode updateTotalIllumination(NodeLlum node, Real currentTotal,
MovableObject objecte) ret Real{
      Real resultat = currentTotal;
      resultat -= node.getIlluminationEstimate();

      si (node.esNodeGrup()) llavors {
            resultat += node.fillDreta().getIlluminationEstimate();
            resultat += node.fillEsq().getIlluminationEstimate();
      }

      retorna resultat;
}
```

Figura 4.3.3.7.6 – Pseudocodi corresponent al mètode updateTotalIllumination.

El mètode **updateTotalIllumination** resta a la iluminació estimada total actual la iluminació estimada del **NodeLlum** passat per paràmetre. A continuació li suma la iluminació estimada del fill dret i el fill esquerra i retorna el resultat. Això és necessari per actualitzar el valor de la iluminació estimada total quan substituïm un node amb els seus dos fills.

- **sonTotFulles**: mètode que comprova si tots els nodes que componen el cut actual són fulles. A la figura 4.3.3.7.7 es pot veure el codi corresponent a aquest mètode.

```
mètode sonTotFulles(Llista_de_NodeLlum lightCut) ret booleà {
	booleà fulles = cert;
	int i = 0;
	mentre (fulles && i<lightCut.mida()) fer {
		si (lightCut[i].esNodePuntual()) llavors {
			fulles = true;
		}
		altrament {
			fulles = false;
		}
		i++;
	}

	retorna fulles;
}
```

Figura 4.3.3.7.7 – Pseudocodi corresponent al mètode sonTotFulles.

Al codi, declarem una variable booleana inicialitzada a cert, **fulles**. Mentre no haguem recorregut tot el **lightCut** i fulles valgui cert, comprovem si el node actual és un **NodePuntual**. Si és així, fulles seguirà valent cert. Altrament, vol dir que algun dels nodes no és puntual, per tant fulles valdrà fals i podrem sortir del bucle. En acabar retornem el valor del booleà fulles.

4.3.3.8.- Classe NodeGrup

Classe que implementa la classe abstracta NodeLlum i representa un clúster binari de llums. Aquesta classe és la que permet generar l'arbre binari de llums, ja que permet emmagatzemar dues llums filles als atributs lightEsq i lightDreta. A la figura 4.3.3.8.1 es pot veure el diagrama de classes de NodeGrup.

NodeGrup

- \- NodeLlum lightEsq
- \- NodeLlum lightDreta
- \- Real distancia

- \+ NodeGrup(NodeLlum esq, dreta, int rep, Real dist, ColourValue dif, ColourValue spec, Real intens)
- \+ llumNoRepresentant()
- \+ recalcularIntensitat()
- \+ computeBoundingBox

Figura 4.3.3.8.1 – Classe NodeGrup

A continuació es farà una breu descripció dels mètodes de la classe NodeGrup:

- **NodeGrup:** constructor de la classe, se li passa per paràmetre els dos NodeLlum fills, el número del representant, la distància entre les dos llums, el color difús, el color especular i la intensitat de la llum.
- **llumNoRepresentant:** mètode que retorna el número de representant que té la llum no representant del NodeGrup. A la figura 4.3.3.8.2 es pot veure el codi corresponent a aquest mètode.

```
mètode llumNoRepresentant() ret int{
      si (representant == lightEsq->llumRepresentant()) llavors {
            retorna lightDreta->llumRepresentant(); }
      altrament {
            retorna lightEsq->llumRepresentant();
      }
}
```

Figura 4.3.3.8.2 – Pseudocodi corresponent al mètode llumNoRepresentant

El funcionament d'aquest mètode és el següent: comprovem si el representant és la llum esquerra i, si és així, retornem la representant de la llum dreta. Altrament, retornem el representant de la llum esquerra.

- **recalcularIntensitat:** mètode que recalcula la intensitat del NodeGrup actual sumant les intensitats dels dos fills. Aquest mètode s'utilitza cada vegada que substituim un NodeLlum al mètode substituirRepresentant de la classe GeneradorArbre (veure apartat 4.3.3.4). A la figura 4.3.3.8.3 es pot veure el codi corresponent a aquest mètode.

```
mètode recalcularIntensitat() {
      intensitat = fillEsq().getIntensitat() +
      fillDreta().getIntensitat();
}
```

Figura 4.3.3.8.3 – Pseudocodi corresponent al mètode recalcularIntensitat

Aquest mètode suma les intensitats dels dos nodes fill del NogeGrup actual, a través dels mètodes **fillEsq** i **fillDreta**.

- **computeBoundingBox**: mètode que recalcula la caixa envolupant del NogeGrup actual en funció de les posicions de les dues llums filles. Aquest mètode es fa servir també al mètode substituirRepresentant de la classe GeneradorArbre, com el recalcularIntensitat (apartat 4.3.3.4). A la figura 4.3.3.8.4 es pot veure el codi corresponent a aquest mètode.

```
mètode computeBoundingBox(){
       int xMinActual = INT_MAX, xMaxActual = -INT_MAX, yMinActual =
       INT_MAX, yMaxActual = -INT_MAX, zMinActual = INT_MAX, zMaxActual
       = -INT_MAX;

       si (fillEsq().getBB().getMinX() < xMinActual)
       llavors xMinActual = fillEsq().getBB().getMinX();
       si (fillEsq().getBB().getMaxX() > xMaxActual)
       llavors xMaxActual = fillEsq().getBB().getMaxX();
       si (fillEsq().getBB().getMinY() < yMinActual)
       llavors yMinActual = fillEsq().getBB().getMinY();
       si (fillEsq().getBB().getMaxY() > yMaxActual)
       llavors yMaxActual = fillEsq().getBB().getMaxY();
       si (fillEsq().getBB().getMinZ() < zMinActual)
       llavors zMinActual = fillEsq().getBB().getMinZ();
       si (fillEsq().getBB().getMaxZ() > zMaxActual)
       llavors zMaxActual = fillEsq().getBB().getMaxZ();
       si (fillDreta().getBB().getMinX() < xMinActual)
       llavors xMinActual = fillDreta().getBB().getMinX();
       si (fillDreta().getBB().getMaxX() > xMaxActual)
       llavors xMaxActual = fillDreta().getBB().getMaxX();
       si (fillDreta().getBB().getMinY() < yMinActual)
       llavors yMinActual = fillDreta().getBB().getMinY();
       si (fillDreta().getBB().getMaxY() > yMaxActual)
       llavors yMaxActual = fillDreta().getBB().getMaxY();
       si (fillDreta().getBB().getMinZ() < zMinActual)
       llavors zMinActual = fillDreta().getBB().getMinZ();
       si (fillDreta().getBB().getMaxZ() > zMaxActual)
       llavors zMaxActual = fillDreta().getBB().getMaxZ();

       BB = new BoundingBox(xMinActual, xMaxActual, yMinActual,
       yMaxActual, zMinActual, zMaxActual);
}
```

Figura 4.3.3.8.4 – Pseudocodi corresponent al mètode computeBoundingBox

El funcionament del mètode és el següent: declarem 6 variables enteres, 3 d'elles inicialitzades amb el valor enter màxim positiu possible (**xMinActual, yMinActual i zMinActual**) i 3 d'elles amb el valor enter màxim negatiu possible (**xMaxActual, yMaxActual, i zMaxActual**). A continuació, comprovem, pel fill esquerra, si **minX** és més petit que xMinActual, si és així, substituim el valor de la variable pel valor de minX fill esquerra. Per **maxX** fem el mateix però al revés, si maxX és major que **xMaxActual** ens guardem el valor de maxX. Fem el mateix per les Y i les X i després repetim l'operació amb el fill dret. D'aquesta manera ens assegurem d'haver guardat els valors mínims i màxims de cada variable.

4.3.3.9.- Classe NodeLlum

Classe que representa un node de llum de l'arbre binari de llums. És una classe abstracta de la qual hereten NodeGrup i NodePuntual. Tal i

com s'ha explicat a l'apartat 4.3.2 de patrons utilitzats aquesta classe implementa el patró Composite. Les dues classes filles n'hereten el seu comportament comú, de manera que poden ser tractades de la mateixa manera però conservant funcionalitats pròpies. A la figura 4.3.3.9.1 es pot veure el diagrama de classes de la classe NodeLlum.

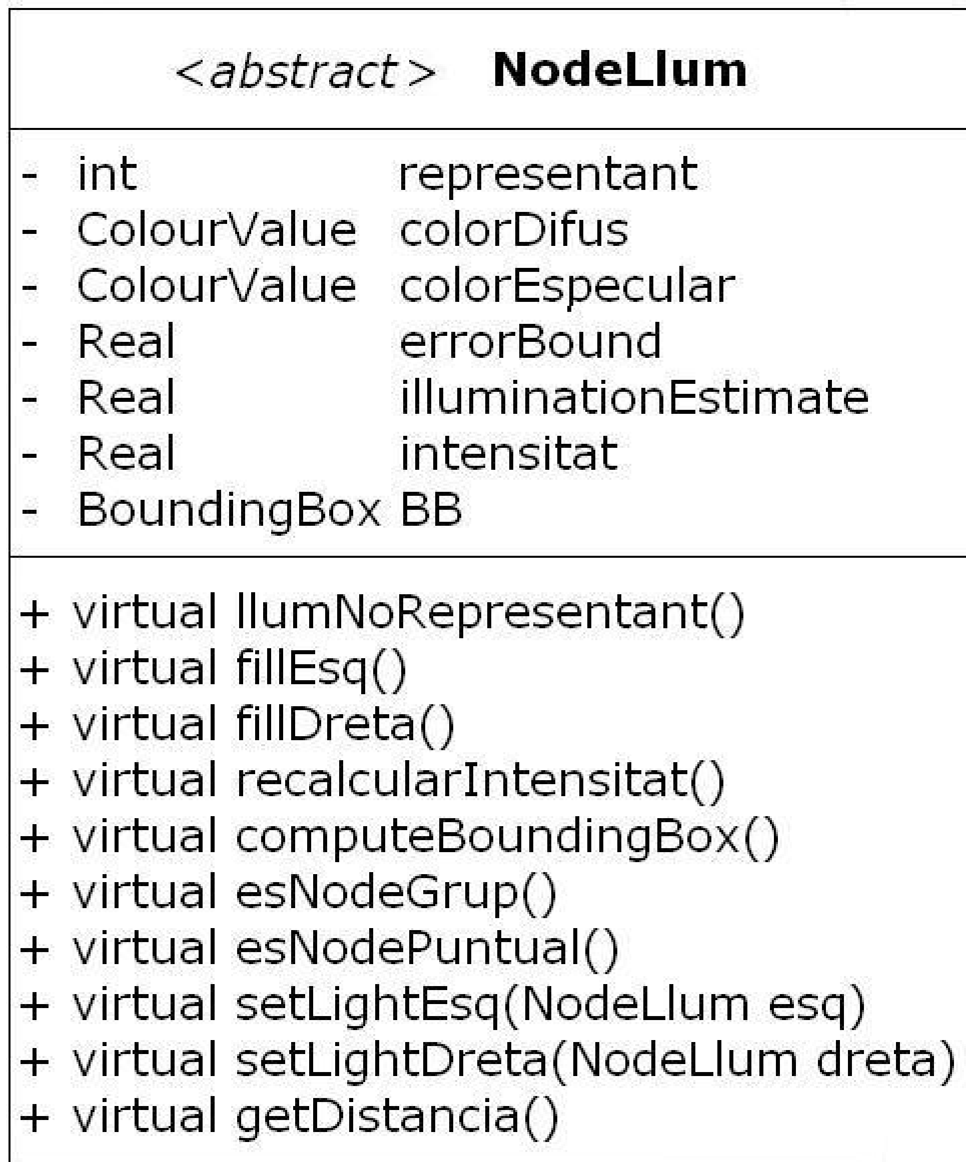


Figura 4.3.3.9.1 – Classe NodeLlum

Els atributs de la classe NodeLlum són els següents:

- **respresentant**: nombre enter que ens indicia quina és la llum representant del NodeLlum.
- **colorDifus**: variable ColourValue d'Ogre que indica el color difús de la llum.
- **colorEspecular**: variable ColourValue d'Ogre que indica el color especular de la llum.

- **errorBound**: variable que conté l'error màxim del node actual.
- **illuminationEstimate**: variable que conté la iluminació estimada del node actual.
- **intensitat**: variable que conté la intensitat de la llum actual.
- **BB**: BoundingBox del node actual.

A continuació es farà un breu resum dels mètodes més importants de la classe. Tots ells són abstractes i, per tant, es comentarà de quina manera es comporten en funció de si l'objecte que els crida és un NodePuntual o un NodeGrup.

- **llumNoRepresentant**: en el cas d'un NodeGrup, retornarà el número que identifica la llum no representant. Pels NodePuntual retornarà -1 ja que al ser llums puntuals no tenen llum no representant.
- **fillEsq i fillDreta**: mètode que retorna el fill esquerra i el fill dret del node actual, en el cas de que sigui un NodeGrup. Pels NodePuntual aquest mètode no té sentit, ja que són llums individuals, i per tant retornarà la variable nul·la.
- **recalcularIntensitat**: mètode que serveix per recalcular la intensitat d'un node amb fills. Per tant, només s'implementa a NodeGrup, a NodePuntual és un mètode que no té sentit.
- **computeBoundingBox**: mètode que recalcula la BoundingBox del Node amb els valors dels seus dos fills.
- **esNodeGrup**: mètode que ens indica si l'objecte actual és un NodeGrup. Per tant, en el cas de la classe NodeGrup retornarà cert, i a la classe NodePuntual retornarà fals.
- **esNodePuntual**: mètode que diu si l'objecte actual és un NodePuntual. Per tant, en el cas de la classe NodePuntual retornarà cert, i a la classe NodeGrup retornarà fals.
- **setLightEsq** i setLightDreta: mètodes que afegeixen, respectivament, els atributs lightEsq i lightDreta al node actual. Només té sentit pels nodes amb fills, per tant només l'implementem a NodeGrup.
- **getDistancia**: l'atribut distancia, que es refereix a la distància entre dues llums, només existeix a la classe NodeGrup, ja que no té sentit implementar-lo per una llum puntual.

4.3.3.10.- Classe NodePuntual

Classe que implementa la classe abstracta NodeLlum i que representa una llum puntual. A l'arbre binari de llums les fulles seran sempre objectes de tipus NodePuntual i n'hi haurà tantes com llums tingui

l'escena. A la figura 4.3.3.10.1 es pot veure un diagrama de classes de la classe NodePuntual.

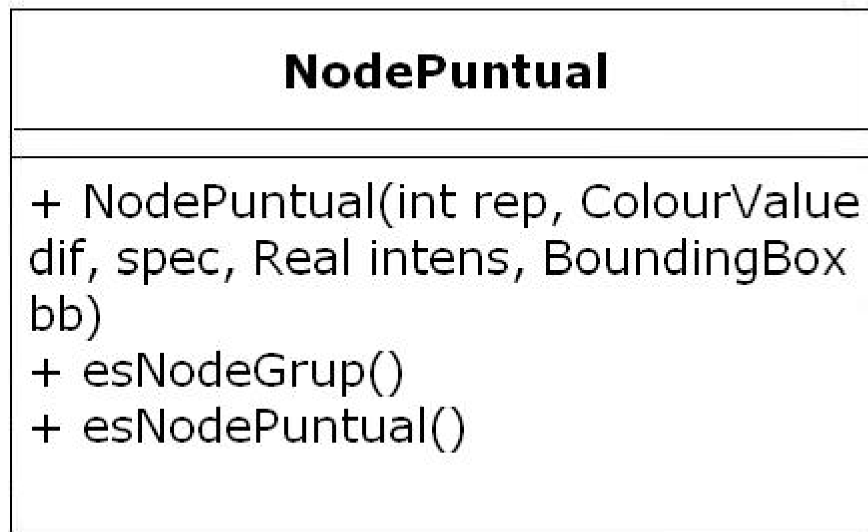


Figura 4.3.3.10.1 – Classe NodePuntual

A continuació es farà un breu resum dels mètodes de la classe NodePuntual.

- **NodePuntual**: constructor de la classe. Se li passa per paràmetre el número de la llum representant, el color difús, el color especular, el valor de la intensitat i la BoundingBox corresponents.
- **esNodeGrup i esNodePuntual:** mètodes que retornen fals i cert respectivament, ja que la classe actual no és NodeGrup, sinó NodePuntual. A l'apartat 4.3.3.9 es pot trobar més informació al respecte.

4.3.3.11.- Classe OGRE::SceneManager

Classe encarregada de gestionar tots els aspectes d'una escena, des de càmeres a llums o entitats. Al nostre projecte la fem servir per a crear objectes Light, Entity i SceneNode que farem servir per a guardar aquestes entitats (per més informació consultar els apartats 2.3.2 i 2.3.3). Per accedir a aquesta classe ho farem des de la classe ListenerLlums ja que és la que necessitarà la informació dels objectes mencionats. A la figura 4.3.3.11.1 es pot veure el diagrama de classe de la classe SceneManager.

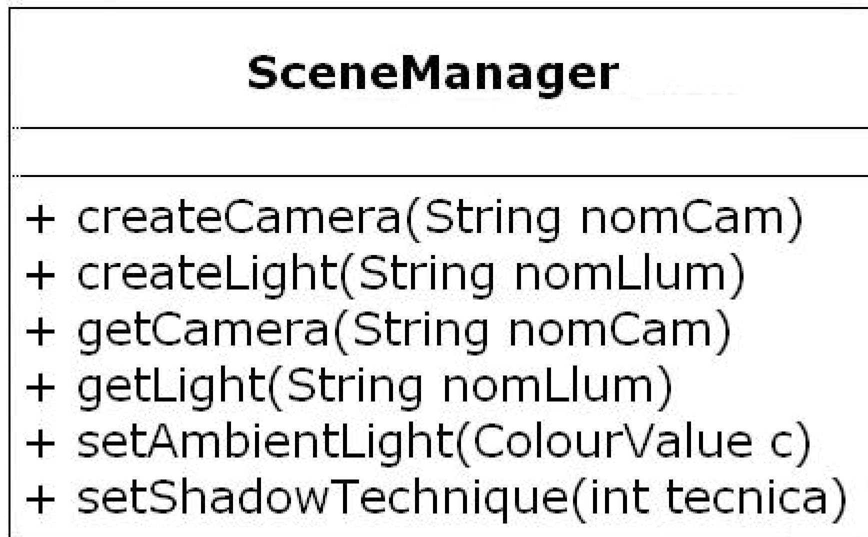


Figura 4.3.3.11.1 – Classe SceneManager

A continuació farem una breu descripció dels mètodes utilitzats a la classe SceneManager:

- **createCamera:** mètode utilitzat per generar una càmera a l'escena, que tindrà com a nom el paràmetre **nomCam**. Aquest mètode retornarà una referència a la càmera creada (classe Camera, apartat 4.3.3.2), la qual haurem de configurar apropiadament**.**
- **createLight:** mètode que genera una llum a l'escena, que tindrà com a nom el paràmetre **nomLlum**. Aquest mètode retornarà una referència a la llum creada (classe Ogre::Light, apartat 4.3.3.5).
- **getCamera:** retorna una referència a l'objecte Camera que te com a nom **nomCam**.
- **getLight:** retorna una referència a l'objecte Light que té com a nom **nomLlum**.
- **setAmbientLight:** mètode que defineix un color per la llum d'ambient que iluminarà la nostra escena. En el nostre cas serà (0,0,0) ja que no volem llum d'ambient, com hem comentat a l'apartat 2.3.8 de la classe Light.
- **setShadowTechnique:** mètode que defineix la tècnica d'iluminació escollida. En el nostre cas és "Additive Stencil Shadows" com es comenta a l'apartat 2.3.8 de la classe Light.

5.- Implementació

Sens dubte una de les fases més importants dins de qualsevol projecte és la implementació. La implementació d'aquesta aplicació s'ha dut a terme amb el llenguatge C++, el qual ens ha aportat conceptes útils per millor la distribució i eficiència del programa final, com són la utilització de classes, l'herència, etc. A més, el motor d'Ogre està escrit en C++ i, tot i que ja n'existeixen versions per altres llenguatges, és utilitzant aquest llenguatge quan trobem més facilitats

En aquest capítol parlarem sobre la implementació dels algoritmes més complexes i més importants de l'aplicació. Dividirem els apartats en classes, exactament igual que a l'apartat 4.3.3, i de cada classe explicarem els algoritmes que trobem més adequats. Hi haurà mètodes que es repetiran, però aquí s'explicaran amb més detall i el codi que els acompanyarà serà el codi definitiu, escrit en C++ i no en pseudocodi, de manera que es podrà tenir una visió més acurada de com i per què es fan determinades coses.

5.1.- Classe GeneradorArbre.

Com ja s'ha vist a l'apartat 4.3.3.3, la classe GeneradorArbre és de les més importants de l'aplicació. A continuació veurem el codi dels mètodes més rellevants juntament a una explicació del seu funcionament.

5.1.1.- Constructor GeneradorArbre

Constructor de la classe i mètode que genera l'arbre binari de llums. Rep una **deque** d'objectes Ogre::Light. Una deque és una llista de de la llibreria STL de C++, Standard Template Library, que permet emmagatzemar objectes del tipus que volguem. A la figura 5.1.1.1 podem veure el codi C++ que implementa el constructor.

```
GeneradorArbre(std::deque<Light> llums) {
      llistaLlums = llums;
      NodeLlum * aux;
      BoundingBox * auxBB;

      for (uint i=0;i<llums.size();i++){
           auxBB = new BoundingBox(llistaLlums[i].getPosition().x,
           llistaLlums[i].getPosition().y,
           llistaLlums[i].getPosition().z);
           aux = new NodePuntual(i,llistaLlums[i].getDiffuseColour(),
           llistaLlums[i].getSpecularColour(),llistaLlums[i].
           getDiffuseColour().b, auxBB);
           parellesLlums.push_back(aux);
      }

      parellesLlums = combinarParelles();

      while (parellesLlums.size() > 1){
            aux = getMillorParella();
            int noRep = aux->llumNoRepresentant();
            eliminarNoRepresentant(noRep);
            substituirRepresentant(aux);
      }
      arbre = parellesLlums[0];
    }
```

Figura 5.1.1.1 – Codi corresponent al constructor GeneradorArbre

Com es pot veure al codi, el mètode està clarament dividit en dos bucles.

Al primer bucle, de tipus **for**, generem els nodes fulla de l'arbre, un per cada llum de la llista **llistaLlums**. Aquests nodes fulla són del tipus **NodePuntual**, classe de la que parlem a l'apartat 4.3.3.10. Per a generar un objecte NodePuntual, li hem de passar, entre d'altres, un objecte de tipus **BoundingBox**. Com que els nodes puntuals són llums individuals no generen més caixa envolupant que el punt que els representa, i és per això que fem servir el constructor que només necessita 3 paràmetres (a l'apartat 4.3.3.1 es menciona el dos tipus de constructor). A més, es pot veure com la quarta variable que li passem per paràmetre al construcotr de NodePuntual és el color difús blau, ja que els nodes puntuals necessiten saber la intensitat de la llum i com que Ogre no defineix el concepte d'intensitat per la llum fem servir el color per a definir-la.

Abans d'entrar al segon bucle combinarem les parelles entre si de totes les maneres possibles, però evitant repetir combinacions i sense que una parella es pugui combinar amb si mateixa. D'això se n'encarrega el mètode **combinarParelles** i a l'apartat 5.1.2 es donen més detalls de la implementació d'aquest mètode.

Al segon bucle, un **while**, generem l'estructura de l'arbre. Primerament obtenim la millor llum del cut actual, amb el mètode **getMillorParella**, que ordena les parelles en funció de la distància

entre si i retorna la millor. A l'apartat 5.1.4 es pot trobar més informació d'aquest mètode. A continuació obtenim la llum no representant de la parella triada per passar-li per paràmetre al mètode **eliminarNoRepresentant**, que es detalla a l'apartat 5.1.6. Finalment, cridem el mètode **substituirRepresentant**, explicat a l'apartat 5.1.7.

Per a surtir d'aquest bucle només hi ha com a condició que parellesLlums tingui mida 1. A cada iteració s'eliminen els nodes que contenen la llum no representant de la parella triada, al mètode eliminarNoRepresentant, per tant ens assegurem que sempre es complirà aquesta condició.

5.1.2.- Mètode combinarParelles

Mètode que combina tots els nodes de l'atribut parellesLlums i els retorna dins una **deque**. A la figura 5.1.2.1 es pot veure el codi corresponent a aquest mètode.

```
std::deque<NodeLlum*> combinarParelles(){
      std::deque<NodeLlum*> combinacio;
      NodeLlum * aux;

      for (uint i=0;i<parellesLlums.size();i++){
            for (uint j=i+1;j<parellesLlums.size();j++){
                  aux = new NodeGrup(parellesLlums[i],
                  parellesLlums[j],
                  escollirRepresentant(parellesLlums[i]-
                  >llumRepresentant(),
                  parellesLlums[j]->llumRepresentant()),
                  distancia(i,j),
                  parellesLlums[j]->getColorDifus() +
                  parellesLlums[i]->getColorDifus(),
                  parellesLlums[j]->getColorEspecular() +
                  parellesLlums[i]->getColorEspecular(),
                  parellesLlums[j]->getColorDifus().b +
                  parellesLlums[i]->getColorDifus().b);
                  aux->computeBoundingBox();
                  combinacio.push_back(aux);
            }
      }

      return combinacio;

}
```

Figura 5.1.2.1 – Mètode combinarParelles de la classe GeneradorArbre

Com es pot veure al codi, utilitzem un bucle **for** dins un altre bucle per a generar totes les combinacions possibles de números de 0 fins a la mida de **parellesLlums**. Per evitar repetits el primer bucle s'inicialitza en i = 0 i el segon bucle s'inicialitza en j = i+1. Dins el segon bucle es crea un objecte de tipus **NodeGrup**. Rep com a

paràmetres els seus dos fills, que seran dos objectes de tipus **NodeLlum** però que realment estaran instanciats com objectes **NodePuntual**, ja que aquest mètode només es crida una vegada per generar l'arbre binari de llums. A més, li passem com a paràmetres el color difús i el color especular, resultat de sumar els colors corresponents dels seus dos fills. Finalment, calculem la **BoundingBox** generada pels dos fills amb el mètode **computeBoundingBox** i afegim el NodeGrup resultant a la llista que hem de retornar.

5.1.3.- Mètode distancia

Mètode que calcula la distància entre dues llums. Es fa servir al mètode combinarParelles d'aquesta mateixa cllase. A la figura 5.1.3.1 es pot veure el codi corresponent a aquest mètode.

```
Real distancia(int a, int b){
      Light llumA = llistaLlums[parellesLlums[a]->llumRepresentant()];
      Light llumB = llistaLlums[parellesLlums[b]->llumRepresentant()];
      return llumA.getPosition().squaredDistance(llumB.getPosition());
}
```

Figura 5.1.3.1 – Mètode distancia de la classe GeneradorArbre

Com es veu al codi, aquest mètode és bastant senzill. Primerament obté les instàncies de les dues llums a partir dels 2 enters que li han passat per paràmetre, que indiquen el representant de cada llum. A continuació calcula la distància al quadrat entre les posicions dels dos punts amb el mètode **squaredDistance**, de manera que retorna la distància elevada al quadrat. Utilitzem aquest mètode perquè és més eficient que calcular la distància normal i només necessitem la distància per a comparar-les entre elles.

5.1.4.- Mètode getMillorParella

Mètode que retorna el millor node de la llista de nodes actual, en funció de la distància que en separa els seus dos fills. A la figura 5.1.4.1 es pot veure el codi corresponent a aquest mètode.

```
NodeLlum * getMillorParella(){
      parellesLlums = ordenarParelles(parellesLlums);
      return parellesLlums[0];
}
```

Figura 5.1.4.1 – Mètode getMillorParella de la classe GeneradorArbre

Com es pot veure al codi, és un mètode molt senzill, ja que delega la responsabilitat d'ordenar les parelles en funció de la distància al mètode **ordenarParelles**. Aquest mètode està explicat a l'apartat 5.1.5. Un cop fet això només li fa falta obtenir el primer element de la llista ja ordenada.

5.1.5.- Mètode ordenarParelles

Mètode que ordena les parelles de la llista passada per paràmetre en funció de la distància que separa els seus nodes fills. A la figura 5.1.5.1 es pot veure el codi corresponent a aquest mètode.

```
std::deque<NodeLlum*> ordenarParelles(std::deque<NodeLlum*>
combinacioLlums){
	std::deque<NodeLlum*> combinacio = combinacioLlums;

	for (uint i=1; i<combinacio.size();i++) {
		for(uint j=0;j<combinacio.size() - 1;j++){
			if (combinacio[j]->getDistancia() >
			combinacio[j+1]->getDistancia()) {
				NodeLlum * temp = combinacio[j];
				combinacio[j] = combinacio[j+1];
				combinacio[j+1] = temp;
			}
		}
	}
	return combinacio;
}
```

Figura 5.1.5.1 – Mètode ordenarParelles de la classe GeneradorArbre

Com es pot veure al codi, el sistema d'ordenació és el de l'algoritme de la bombolla. Tot i que hi ha mètodes més eficients s'ha triat aquest per ser senzill d'implementar i molt utilitzat. Funciona revisant cada element de la llista que ha de ser ordenada amb el següent, intercanviant-los de posició si estan en l'ordre equivocat. És necessari revisar la llista fins que tots els elements estiguin ordenats.

5.1.6.- Mètode eliminarNoRepresentant

Mètode que elimina de la llista de parelles aquells nodes que tenen com a fill el representant passat per paràmetre. A la figura 5.1.6.1 es pot veure el codi corresponent a aquest mètode.

```
void eliminarNoRepresentant(int noRep) {
	std::deque<NodeLlum*> llistaAux;
	NodeLlum * parellaAux;
	for (uint i=0; i<parellesLlums.size();i++){
		parellaAux = parellesLlums[i];
		if (parellaAux->esNodeGrup() && parellaAux->fillDreta()-
		>llumRepresentant() != noRep && parellaAux->fillEsq()-
		>llumRepresentant() != noRep) {
			llistaAux.push_back(parellaAux);
		}
	}
	parellesLlums = llistaAux;
}
```

Figura 5.1.6.1 – Mètode eliminarNoRepresentant de la classe GeneradorArbre

Com es veu al codi, iterem la llista de nodes **parellesLlums** amb un bucle **for**. Generem una llista auxiliar on anirem afegint els nodes que volem conservar, anomenada **parellaAux**. Per cada parella de la llista comprovem si és un objecte de tipus **NodeGrup** amb el mètode booleà **esNodeGrup**. Si és així l'afegirem a la llista auxiliar i quan sortim del bucle la retornarem.

5.1.7.- Mètode substituirRepresentant

Mètode que substitueix a la llista de parelles els nodes fills pel node passat per paràmetre en el cas de que tinguin el mateix representant. A la figura 5.1.7.1 es pot veure el codi corresponent a aquest mètode.

```
void substituirRepresentant(NodeLlum * llumRep) {
      NodeLlum * parellaAux;
      for (uint i=0; i<parellesLlums.size();i++){
            parellaAux = parellesLlums[i];
            if (parellaAux->esNodeGrup()) {
                  if (parellaAux->fillDreta()->llumRepresentant() ==
                  llumRep->llumRepresentant()) {
                        parellaAux->setLightDreta(llumRep);
                        parellaAux->recalcularIntensitat();
                  }
                  if (parellaAux->fillEsq()->llumRepresentant() ==
                  llumRep->llumRepresentant()) {
                        parellaAux->setLightEsq(llumRep);
                        parellaAux->recalcularIntensitat();
                  }
            }
      }
}
```

Figura 5.1.7.1 – Mètode substituirRepresentant de la classe GeneradorArbre

Com es pot veure al codi, iterem la llista de nodes **parellesLlums** amb un bucle **for**. Per cada parella comprovem si es tracta d'un objecte de tipus **NodeGrup**. Si és així comprovem si el representant del seu fill esquerra o dret és el mateix que el del **NodeLlum** passat per paràmetre, **llumRep**. En aquest cas el nou fill esquerra o dret passarà a ser el node llumRep.

5.2.- Classe ListenerLlums

Una altra classe amb molt de pes al projecte, ja que és la que s'encarrega de recorrer l'arbre binari de llum generat per la classe GeneradorArbre i obtenir el lightcut corresponent a cada punt. A continuació veurem el codi dels mètodes més rellavants amb la seva explicació pertinent.

5.2.1.- Mètode objectQueryLights

Com ja s'ha comentat a l'apartat 4.3.3.7 de la classe Listener, aquesta classe hereta de la classe Listener d'Ogre (apartat 4.3.3.6) i n'implementa el mètode objectQueryLights. D'aquesta manera, si tenim un objecte de tipus MovableObject podem fer servir el mètode setListener i passar-li un objecte de la classe ListenerLlums (a l'apartat 4.3.2 secció Listener se'n comenta més al respecte). Així, quan el motor d'Ogre hagi de decidir quines llums iluminaran aquest objecte, cridarà el mètode objectQueryLights del Listener que tingui aquest MovableObject (en el nostre cas serà un objecte de tipus Entity, que hereta de MovableObject, apartat 2.3.6). A la figura 5.2.1.1 es pot veure el codi d'aquest mètode.

```
const LightList * objectQueryLights(const MovableObject * objecte) {
	if (llistaLlums->size() == 0) {
		if (actiu) {
			llistaNodesLlums = buildLightCut(objecte,arbre-
			>getArrel());
			emplenarLlistaLlums(llistaNodesLlums);
		}
		else {
			emplenarLlistaLlums(arbre->getLlistaLlums().size());
		}
	}

	return llistaLlums;
}
```

Figura 5.2.1.1 – Mètode objectQueryLights de la classe ListenerLlums

Com es pot veure al codi, el mètode **objectQueryLights** rep per paràmetre **l'objecte MovableObject** amb el que està aparellat. Primer de tot comprova si la llista de llums **llistaLlums** té mida zero, que voldrà dir que encara no s'ha calculat el Lightcut per aquest ListenerLlums. A continuació, comprovem si l'atribut **actiu** val cert. Aquesta variable ens indica si hem de fer servir l'algoritme de lightcuts o no. Si hem de fer servir l'algoritme, cridem el mètode **builtLightCut** passant-li el MovableObject **objecte** i el node arrel de l'arbre binari de llums, amb el mètode **getArrel**, i acabem cridant el mètode **emplenarLlistaLlums** (mètode explicat a l'apartat 5.2.3), que ens converteix el lightCut d'objectes NodeLlum en una llista d'objectes Ogre::**Light**. Si pel contrari no hem de fer servir l'algoritme cridarem el mètode **emplenarLlistaLlums** passant-li per paràmetre un enter amb el nombre de llums que haurem de mostrar (aquest mètode està explicat a l'apartat 5.2.3).

5.2.3.- Mètode buildLightCut

Mètode encarregat de recorrer l'arbre binari de llums i retornar una llista dels NodeLlum que representen el lightcut. Rep per paràmetre el MovableObject que s'ha d'iluminar i el node arrel de l'arbre binari. A la figura 5.2.3.1 es pot veure el codi corresponent al mètode.

```
std::deque<NodeLlum*> buildLightCut(const MovableObject * objecte,
NodeLlum * node){
	std::deque<NodeLlum*> cutActual;
	NodeLlum* worst;
	cutActual.push_back(node);
	Real currentTotal =
	computeTotalIlluminationEstimate(objecte,cutActual);
	node->setErrorBound(calcularCota(objecte,node));
	node->setIlluminationEstimate(
	illuminationEstimate(objecte,node));
	bool fi = false;
	Real errorBound, illuminationEstimate;

	while (!fi) {
		worst = findWorstErrorBound(objecte,cutActual);
		errorBound = worst->getErrorBound() - worst-
		>getIlluminationEstimate();
		illuminationEstimate = percentatge*currentTotal;

		if (errorBound<0) {errorBound = errorBound * (-1);}
		if ((errorBound==0)|| ((errorBound > illuminationEstimate)
		&& !(sonTotFulles(cutActual)))) {
			cutActual = eliminar(cutActual,worst);
			computeBoundAndEstimateForChildren(worst,objecte);
			cutActual.push_back(worst->fillDreta());
			cutActual.push_back(worst->fillEsq());
			currentTotal = updateTotalIllumination(worst,
			currentTotal, objecte);
		}
		else {
			fi = true;
		}
	}

	return cutActual;
}
```

Figura 5.2.3.1 – Mètode buildLightCut de la classe ListenerLlums

Com es pot veure al codi, aquest és un mètode bastant llarg i complexe. Primerament es pot veure com instanciem una llista de tipus **deque** que contindrà objectes **NodeLlum**, és a dir, el lightcut en si. A continuació hi posarem el node arrel i calcularem l'iluminació total de l'escena amb aquest únic node, mitjançant el mètode **computeTotalIlluminationEstimate** (comentat a l'apartat 5.2.4), i en guardarem el resultat a **currentTotal**. Després calcularem l'error màxim del node arrel amb el mètode **calcularCota** (comentat a l'apartat 5.2.5) i la iluminació estimada del node amb el mètode

illuminationEstimate (comentat apartat 5.2.6). A continuació entrem al bucle del mètode, que fa el següent.

Primerament instanciem la variable **worst** amb el pitjor node del cut (a la primera volta serà el mateix node arrel) utilitzant el metode **findWorstErrorBound** (comentat a l'apartat 5.2.7). A continuació calcularem el marge d'error restant l'error màxim i la iluminació estimada i en guardarem el resultat a la variable **errorBound**. A continuació multiplicarem la illuminació estimada total pel percentatge d'error que volem assolir i el guardarem a la variable **illuminationEstimate**. Després multiplicarem per -1 el contingut de la variable errorBound ja que volem que sempre sigui positiva i entrarem al **if** més important del mètode. Aquest condicional és el que decideix si refinem més el cut actual o ja ens serveix. Les condicions per sortir del bucle són:

- Que la iluminació estimada total multiplicada pel percentatge maxim d'error siguin més grans que la diferència entre l'error màxim i la iluminació estimada.
- Que tots els nodes del cut siguin fulles, utilitzant el mètode **sonTotFulles** (comentat a l'apartat 5.2.8).
- A més, es pot donar el cas de que l'errorBound valgui 0, cosa que vol dir que la cara de l'objecte a iluminar està en direcció oposada a la llum del node actual i, per tant, hem de refinar aquest node.

Aquesta condició es pot veure a la figura 5.2.3.2.

```
if ((errorBound==0)|| ((errorBound > illuminationEstimate) &&
!(sonTotFulles(cutActual))))
```

Figura 5.2.3.2 – Condició per continuar dins el bucle del mètode buildLightCut

Si entrem dins el condicional vol dir que hem de refinar més i per tant eliminem el node actual del cut i calculem l'error màxim i la iluminació estimada dels dos fills del node actual amb el mètode **computeBoundAndEstimateForChildren** (comentat a l'apartat 5.2.9). A continuació afegim el node dret i el node esquerra al cut actual i actualitzem la iluminació estimada total amb el mètode updateTotalIllumination (comentat a l'apartat 5.2.10).

Si no entrem al condicional farem que la variable booleana que controla el bucle passi a valer **true** i sortim.

Finalment retornarem la llista de NodeLlum **cutActual**.

5.2.4.- Mètode emplenarLlistaLlums

L'objectiu d'aquest mètode és que converteixi el lightcut (una llista d'objectes NodeLlum) en una llista d'objectes Ogre::Light. A més, en el cas que no volguem utilitzar l'algoritme de Lightcut hi ha una altra versió del mètode que simplement emplena la llista de llums amb totes les llums que hi ha declarades. A la figura 5.2.4.1 es pot veure el codi corresponent al primer mètode emplenarLlistaLlums, que té per paràmetre una llista d'objectes NodeLlum.

```
void emplenarLlistaLlums(std::deque<NodeLlum*> lightcut){
      llistaLlums = new LightList();
      if (lightcut.size()>0) {
            for (uint i=0;i<lightcut.size();i++) {
                  int representant = lightcut[i]->llumRepresentant();
                  string nomLlum = "Light";
                  char *myBuff = new char[100];
                  memset(myBuff,'\0',100);
                  itoa(representant,myBuff,10);
                  string numLlum = myBuff;
                  delete[] myBuff;
                  nomLlum.append(numLlum);
                  Light * llum = mSceneMgr->getLight(nomLlum);
                  llum->setDiffuseColour(lightcut[i]-
                  >getIntensitat(),lightcut[i]-
                  >getIntensitat(),lightcut[i]->getIntensitat());
                  llum->setSpecularColour(lightcut[i]-
                  >getIntensitat(),lightcut[i]-
                  >getIntensitat(),lightcut[i]->getIntensitat());
                  llistaLlums->push_back(llum);
            }
      }
}
```

Figura 5.2.4.1 – Primer mètode emplenarLlistaLlums de la classe ListenerLlums

Aquest mètode sembla molt complexe però el seu funcionament és bastant senzill. Entrem en un bucle que recorre els objectes **NodeLlum** de la llista **lightcut** i, per cadascun, n'obté el representant. A continuació construim una cadena de caràcters **nomLLum** amb la forma "Light**X"** on la **X** es correspondrà amb el número de representant. Després accedim al SceneManager (classe comentada a l'apartat 4.3.3.11) i cridem el mètode **getLight** passant-li per paràmetre la cadena nomLlum. Com que les llums han estat generades amb noms d'aquest tipus obtindrem la llum de l'escena.

Haurem de modificar-li el color a la llum passant-li la intensitat que té l'objecte NodeLlum, ja que si es tracta d'un clúster de llums probablement haurà augmentat d'intensitat. Això ho farem amb els

mètodes **setDiffuseColour** i **setSpecularColour**. Finalment guardarem la nova llum a la llista de llums.

A la figura 5.2.4.2 es pot veure el codi corresponent al segon mètode amb nom emplenarLlistaLlums, que té com a paràmetre un enter.

```
void emplenarLlistaLlums(int nLlums){
            llistaLlums = new LightList();
            for (uint i=0;i<nLlums;i++) {
                  string nomLlum = "Light";
                  char *myBuff = new char[100];
                  memset(myBuff,'\0',100);
                  itoa(i,myBuff,10);
                  string numLlum = myBuff;
                  delete[] myBuff;
                  nomLlum.append(numLlum);
                  Light * llum = mSceneMgr->getLight(nomLlum);
                  llistaLlums->push_back(llum);
            }

      }
```

Figura 5.2.4.2 – Segon mètode emplenarLlistaLlums de la classe ListenerLlums

Aquest mètode fa un bucle de 0 fins a **nLlums**, que és l'enter passat per paràmetre (correspon al nombre total de llums a l'escena). Per cada volta del bucle obtindrà del SceneManager una llum de nom "**LightX**", exactament igual que al mètode abans comentat. La única diferència és que aquí no modificarem el color de la llum ja que volem que es mostri tal qual i que agafarem totes les llums de l'escena.

5.2.5.- Mètode calcularCota

Aquest mètode calcula l'error màxim per un node i un objecte. A la figura 5.2.5.1 es pot veure el codi corresponent a aquest mètode.

```
Real calcularCota(const MovableObject * objecte, NodeLlum * node){
	ColourValue colorDifus = ((Entity*)objecte)->getSubEntity(0)-
	>getMaterial()->getBestTechnique(0)->getPass(0)->getDiffuse();
	ColourValue colorEspecular = ((Entity*)objecte)-
	>getSubEntity(0)->getMaterial()->getBestTechnique(0)-
	>getPass(0)->getSpecular();
	Real kDifus = (((Real)1/(Real)3)*colorDifus.r) +
	(((Real)1/(Real)3)*colorDifus.g) +
	(((Real)1/(Real)3)*colorDifus.b);
	Real kEspecular = (((Real)1/(Real)3)*colorEspecular.r) +
	(((Real)1/(Real)3)*colorEspecular.g) +
	(((Real)1/(Real)3)*colorEspecular.b);
	Real material = kDifus + kEspecular;

	Vector3 posicioObjecte = ((Entity*)objecte)-
	>getParentSceneNode()->getPosition();
	Vector3 posicioLlum = arbre->getLlum(node-
	>llumRepresentant()).getPosition();
	Real distancia = posicioLlum.distance(posicioObjecte);
	Real geometria = 1 / distancia;
	Vector3 vectorDiferencia = posicioLlum - posicioObjecte;
	Vector3 normalObjecte = Vector3::UNIT_Y;
	Radian angle = vectorDiferencia.normalisedCopy().angleBetween(
	normalObjecte.normalisedCopy());
	Real valorCosinus = Math::Cos(angle,false);
	Real resultat = 0;

	if (valorCosinus >=0) {
		Real intensitat = 0;
		if (node->esNodePuntual()) {
			intensitat = node->getIntensitat();
		}
		else if (node->esNodeGrup()){
			intensitat = node->fillDreta()->getIntensitat() +
			node->fillEsq()->getIntensitat();
		}
		resultat =  material * geometria * intensitat;
	}

	return resultat;
}
```

Figura 5.2.5.1 – Segon mètode calcularCota de la classe ListenerLlums

Sembla molt llarg i complexe però només agafa dades d'objectes de tipus Material (classe comentada a l'apartat 2.3.6), Light (classe comentada a l'apartat 2.3.8) o Camera (classe comentada a l'apartat 2.3.7). A l'apartat 3.6, càlcul dels errors màxims, es comenta com es calcula l'error màxim de cadascun dels components material,

geometria, intensitat i visibilitat, per tant no entrarem en més detall al respect i només explicarem els detalls tècnics de la implementació.

Primerament obtenim les dades del component material: el color difús i el color especular els guardem a les variables **colorDifus** i **colorEspecular**, després d'haver-les obtingut de l'entitat **objecte** passada per paràmetre. A continuació convertim a un valor enter els components RGB dels dos materials i els guardem a les variables **kDifus** i **kEspecular** respectivament. Finalment sumem aquests valors i els guardem a la variable **material**.

Seguim amb la variable geometria. Obtenim la posició de l'objecte i la posició de la llum, que guardarem a **posicioObjecte** i **posicioLlum** respectivament. Calcularem la distancia entre els dos punts fent servir el mètode **distance** de la classe **Vector3** d'Ogre i la guardarem a la variable **distancia**. El valor del terme geometric el guardarem a la variable **geometria**.

A continuació comprovem si la cara de l'objecte es veu afectada o no per la llum que l'ilumina. Per a fer-ho obtenim el vector normal de l'objecte i el vector generat per la posició de la llum i la posició de l'objecte i en calculem l'angle que generen. Si el cosinus resultant és negatiu vol dir que el resultat de l'error màxim és 0 i, per tant, hem de sortir del mètode.

Finalment calculem el terme intensitat sumant les intensitats dels dos nodes fills i en guardem el resultat a la variable **intensitat**.

Retornarem la multiplicació dels termes geometria, material i intensitat.

5.2.6.- Mètode illuminationEstimate

Aquest mètode calcula la iluminació estimada per un node i un objecte. El codi que l'implementa es pot veure a la figura 5.2.6.1.

```
Real illuminationEstimate(const MovableObject * objecte, NodeLlum *
node){
	Real resultat = 0;
	Vector3 posicioObjecte = ((Entity*)objecte)-
	>getParentSceneNode()->getPosition();
	Vector3 posicioLlum = arbre->getLlum(node-
	>llumRepresentant()).getPosition();
	Real distancia = posicioLlum.distance(posicioObjecte);
	Real geometria = 1 / distancia;
	Vector3 normalObjecte = Vector3::UNIT_Y;
	ColourValue colorDifus = ((Entity*)objecte)->getSubEntity(0)-
	>getMaterial()->getBestTechnique(0)->getPass(0)->getDiffuse();
	ColourValue colorEspecular = ((Entity*)objecte)-
	>getSubEntity(0)->getMaterial()->getBestTechnique(0)-
	>getPass(0)->getSpecular();
	Real kDifus = (((Real)1/(Real)3)*colorDifus.r) +
	(((Real)1/(Real)3)*colorDifus.g) +
	(((Real)1/(Real)3)*colorDifus.b);
	Real kEspecular = (((Real)1/(Real)3)*colorEspecular.r) +
	(((Real)1/(Real)3)*colorEspecular.g) +
	(((Real)1/(Real)3)*colorEspecular.b);
	Vector3 vectorDiferencia = posicioLlum - posicioObjecte;
	Radian angle =
	vectorDiferencia.normalisedCopy().angleBetween(normalObjecte.nor
	malisedCopy());
	Real component1 = Math::Cos(angle,false) * kDifus;
	if (component1>=0) {
		Real rugositat = ((Entity*)objecte)->getSubEntity(0)-
		>getMaterial()->getBestTechnique(0)->getPass(0)-
		>getShininess();
		Vector3 vectorR =
		vectorDiferencia.normalisedCopy().reflect(normalObjecte.no
		rmalisedCopy());
		Vector3 posicioCamara = mSceneMgr->getCamera("PlayerCam")-
		>getPosition();
		Vector3 vectorV = (posicioCamara -
		posicioObjecte).normalisedCopy();
		Real resultatRV = vectorR.absDotProduct(vectorV);
		Real component2 = Math::Pow(resultatRV,rugositat);
		if (component2<0) {
			component2= component2 * (-1);
		}
		Real material = component1 + component2;
		Vector3 * vectorL = new
		Vector3(posicioLlum.x,posicioLlum.y,posicioLlum.z);
		Real intensitat = 0;
		if (node->esNodePuntual()) {
			intensitat = node->getIntensitat();
		}
		else if (node->esNodeGrup()){
			intensitat = node->fillDreta()->getIntensitat() +
			node->fillEsq()->getIntensitat();
		}
		resultat = geometria * material * 1 * intensitat;
	}
	return resultat;
}
```

Figura 5.2.6.1 – Mètode illuminationEstimate de la classe ListenerLlums

Com es pot veure al codi, aquest mètode sorpren per la seva llargada, però igual que el mètode calcularCota abans comentat, dins seu no té cap bucle i realment és bastant senzill. A l'apartat 3.7, càlcul del la iluminació estimada, es comenta com es calcula la iluminació estimada, per tant no té sentit profunditzar més en aquest aspecte i ens centrarem en l'aspecte tècnic de la implementació.

Primerament calculem el terme geometric, que guardarem a la variable **geometria**. Aquest terme es calcula de la mateixa manera que al mètode calcularCota, comentat a l'apartat 5.2.5.

A continuació calculem el terme material, que es calcula de forma diferent, separat en dos parts. Primer de tot, obtenim el color difús i el color especular i en guardem un valor enter a les variables kDifus i kEspecular tal i com hem fet al mètode calcularCota. A continuació, calculem l'angle entre la normal de l'objecte a iluminar i el vector generat per la posició de la llum i la posició de l'objecte. Multipliquem el seu cosinus pel terme difús kDifus i obtenim la primera component del material. Si aquest valor és inferior a 0, vol dir que la llum no ilumina aquest objecte. Altrament, obtenim la rugositat de l'objecte i calculem el vector reflexió. Obtenim el vector amb la posició de la càmera i el multipliquem pel vector de reflexió, obtenint la variable real **resultatRV**. Aquesta variable elevada a la rugositat ens donen la segona component del material. Sumem les dues components del material i les guardem a la variable **material**.

El terme intensitat es calcula igual que al mètode calcularCota, sumant les intensitats dels dos nodes fill i guardant-les a la variable **intensitat**. Finalment retornem la multiplicació dels termes geometria, material i intensitat.

5.2.7.- Mètode findWorstErrorBound

Mètode utilitzat per saber quin és el pitjor node del cut actual. A la figura 5.2.7.1 es pot veure el codi corresponent a aquest mètode.

```
NodeLlum* findWorstErrorBound(const MovableObject * objecte,
std::deque<NodeLlum*> cutActual){
	std::deque<NodeLlum*> cutAux;
	for (uint a=0;a<cutActual.size();a++) {
		if (cutActual[a]->esNodeGrup()) {
			cutAux.push_back(cutActual[a]);
		}
	}
	for (uint i=1; i<cutAux.size();i++) {
		for(uint j=0;j<cutAux.size() - 1;j++){
			if (cutAux[j]->getErrorBound() < cutAux[j+1]-
			>getErrorBound()) {
				NodeLlum * temp = cutAux[j];
				cutAux[j] = cutAux[j+1];
				cutAux[j+1] = temp;
			}
		}
	}
	if (cutAux.size()>0) {
		return cutAux[0];
	}
	else {
		return NULL;
	}
}
```

Figura 5.2.7.1 –Mètode findWorstErrorBound de la classe ListenerLlums

Com es pot veure al codi, aquest mètode consta de dos bucles. Al primer bucle iterem la llista **cutActual** i ens guardem a la llista auxiliar **cutAux** els nodes que són **NodeGrup**. Això ho fem perquè només volem retornar clústers de llum i no nodes puntuals, ja que un node puntual no es pot refinar més dins l'algoritme buildLightCut (veure apartat 5.2.3).

A continuació, fem servir l'algoritme d'ordenació de la bombolla per ordenar la llista auxiliar en funció de l'error màxim de cada **NodeLlum**.

Finalment, retornem el primer element d'aquesta llista, que serà el que tindrà l'error més gran.

5.2.8.- Mètode sonTotFulles

Mètode utilitzat per saber si el cut actual està format només per objectes de tipus NodePuntual. A la figura 5.2.8.1 es pot veure el codi corresponent a aquest mètode.

```
bool sonTotFulles(std::deque<NodeLlum*> lightCut) {
	bool fulles = true;
	int i = 0;
	while (fulles && i<lightCut.size()) {
		if (lightCut[i]->esNodePuntual()){
			fulles = true;
		}
		else {
			fulles = false;
		}
		i++;
	}

	return fulles;
}
```

Figura 5.2.8.1 –Mètode sonTotFulles de la classe ListenerLlums

Com es pot veure al codi, el mètode recorre amb un bucle **while** tot el cut passat per paràmetre amb la variable **lightCut**. Si un dels nodes no és puntual sortirem del bucle i retornarem el valor de la variable **fulles**, que serà fals. Altrament, sortirem perque arribarem al final de la llista i la variable fulles valdrà cert.

5.2.9.- Mètode computeBoundAndEstimateForChildren

Mètode que calcula l'eror màxim i la iluminació estimada pels dos fills del node passat per paràmetre. A la figura 5.2.9.1 es pot veure el codi corresponent a aquest mètode.

```
void computeBoundAndEstimateForChildren(NodeLlum* node, const
MovableObject * objecte) {
	if (node->esNodeGrup()) {
		node->fillDreta()->setErrorBound(
		calcularCota(objecte,node->fillDreta()));
		node->fillDreta()->setIlluminationEstimate(
		illuminationEstimate(objecte,node->fillDreta()));
		node->fillEsq()->setErrorBound( calcularCota(
		objecte,node->fillEsq()));
		node->fillEsq()-
		>setIlluminationEstimate(illuminationEstimate
		(objecte,node->fillEsq()));
	}
}
```

Figura 5.2.9.1 –Mètode computeBoundAndEstimateForChildren de la classe ListenerLlums

Com es pot veure al codi, el mètode comprova si el **NodeLlum node** passat per paràmetre és de tipus **NodeGrup**, amb el mètode **esNodeGrup**. Si és així cridarà els mètodes **calcularCota** i **illuminationEstimate** pel fill esquerra i pel fill dret.

5.2.10.- Mètode updateTotalIllumination

Mètode que actualitza la iluminació total de l'escena amb els valors dels dos fills del NodeLLum passat per paràmetre. A la figura 5.2.10.1 es pot veure el codi corresponent a aquest mètode.

```
Real updateTotalIllumination(NodeLlum* node, Real currentTotal, const
MovableObject * objecte) {
	Real resultat = currentTotal;
	resultat -= node->getIlluminationEstimate();
	if (node->esNodePuntual()) {}
	else if (node->esNodeGrup()){
		resultat += node->fillDreta()->getIlluminationEstimate();
		resultat += node->fillEsq()->getIlluminationEstimate();
	}

	return resultat;
}
```

Figura 5.2.10.1 – Mètode updateTotalIllumination de la classe ListenerLlums

Com es pot veure al codi, restem a la variable **resultat**, que conté la iluminació estimada total, el valor de la iluminació estimada del **NodeLlum node** passat per paràmetre. Si el node passat per paràmetre és un NodeGrup sumem al resultat la iluminació estimada dels seus dos fills. Finalment retornem el resultat.

5.3.- Funcionament de CEGUI

La principal particularitat d'aquesta llibreria gràfica és que tots els elements hereten de la classe CEGUI::Window. El que fa que agafin una forma o una altra és l'aparença que adopten mitjançant un fitxer de text xml, que defineix el comportament d'aquests elements. A la figura 5.3.1 es pot veure un exemple de codi usant CEGUI.

```
CEGUI::Checkbox* cCas;
CCas=(CEGUI::Checkbox*)CEGUI::WindowManager::getSingleton().
createWindow("TaharezLook/Checkbox", (CEGUI::utf8*)"cCas");
```

Figura 5.3.1 – Codi d'exemple d'ús de la llibreria CEGUI

Al codi es pot veure com, a la primera línia, definim un objecte Checkbox. A continuació instanciem un nou objecte Window, però li fem un casting i el convertim a Checkbox. El fitxer TaharezLook ens diu quina forma ha de tenir la Checkbox creada.

El primer pas ha de ser sempre crear una finestra arrel, de la qual penjaran totes les altres, i un sistema GUI, que serà l'encarregat d'organitzar totes les finestres i de pintar-les per pantalla. Com a punt important, quan es crea una finestra se li ha de donar un nom únic, d'aquesta manera ens hi podrem referir de manera única. A la figura 5.3.2 es pot veure aquest procés.

```
mGUIRenderer = new CEGUI::OgreCEGUIRenderer(mWindow,
Ogre::RENDER_QUEUE_OVERLAY, false, 3000,mSceneMgr);
mGUISystem = new CEGUI::System(mGUIRenderer);

wRoot=CEGUI::WindowManager::getSingleton().createWindow
((CEGUI::utf8*)"DefaultWindow", (CEGUI::utf8*)"root");

mGUISystem->setGUISheet(wRoot);
mGUISystem->setDefaultMouseCursor((CEGUI::utf8*)"TaharezLook",
(CEGUI::utf8*)"MouseArrow");
```

Figura 5.3.2 – Codi necessari per crear una finestra arrel

Com es pot veure al codi, primerament inicialitzem el sistema que renderitza la GUI (classe **CEGUI::OgreCEGUIRenderer**), i a continuació inicialitzem la GUI en si mateixa (classe **CEGUI::System**). Quan ja tenim el sistema GUI apunt creem la finestra arrel, a la variable **wRoot**, a la qual anomenarem **root**. A continuació l'assignem al sistema GUI creat, **mGUISystem**, i definim el tipus de ratolí que veurem a pantalla, de tipus **mouseArrow**.

Un cop inicialitzat el sistema CEGUI, veurem com hem creat la resta d'elements. Els mètodes per canviar la posició i la mida estan en coordenades CEGUI, que van de 0 a 1, on el valor màxim serà la mida total de la pantalla. A continuació, veurem les classes utilitzades.

CEGUI::PushButton

Classe que com el seu nom indica serveix per a crear botons. A la figura 5.3. 3 es pot veure un l'aspecte que té un objecte d'aquest tipus.

Figura 5.3.3 – Exemple d’un botó de la llibreria CEGUI

A la figura 5.3.4 es pot veure el codi necessari per generar un botó de tipus PushButton.

```
CEGUI::Window*sort=CEGUI::WindowManager::getSingleton().
createWindow("TaharezLook/Button","surt");
sort->setText("Sortir");
sort->setSize(CEGUI::UVector2(CEGUI::UDim(0.15, 0),
CEGUI::UDim(0.10, 0)));
sort->setPosition(CEGUI::UVector2(cegui_reldim(0.55f),
cegui_reldim( 0.87f)) );
wRoot->addChildWindow(sort);
```

Figura 5.3.4 – Codi necessari per generar un botó de tipus PushButton

Com es pot veure al codi, primerament generem un objecte de tipus **Window** i de nom **surt**. A continuació li posem el text que volem que mostri, **Sortir** en aquest cas. Li canviem la mida amb el mètode **setSize** i la posició amb el mètode **setPosition**. Finalment l’afegim a la finestra arrel.

CEGUI::ScrollBar

Classe que serveix per a crear barres de desplaçament o scroll. A la figura 5.3.5 es pot veure un l’aspecte que té un objecte d’aquest tipus.

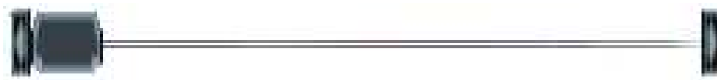


Figura 5.3.5 – Exemple d'un scroll de la llibreria CEGUI

A la figura 5.3.6 es pot veure el codi necessari per generar una barra de desplaçament de tipus ScrollBar.

```
CEGUI::Window * scroll = static_cast<CEGUI::Scrollbar*>
(CEGUI::WindowManager::getSingleton().createWindow
("TaharezLook/HorizontalScrollbar",(CEGUI::utf8*)"scroll"));
scroll->setPosition(CEGUI::UVector2(cegui_reldim(0.45f),
cegui_reldim( 0.30f)) );
scroll->setSize(CEGUI::UVector2(cegui_reldim(.4f),
cegui_reldim(0.05f)) );
scroll->setDocumentSize(1500);
scroll->setStepSize(1);
scroll->setScrollPosition(0);
wRoot->addChildWindow(scroll);
```

Figura 5.3.6 – Codi necessari per generar una barra de desplaçament de tipus PushButton

Com es pot veure al codi, primerament generem un objecte de tipus **Window** i de nom **scroll**. Li canviem la posició amb el mètode **setPosition** i la mida amb el mètode **setSize**. A continuació li assignem una mida màxima, amb el mètode **setDocumentSize**, una mida de quant avancarà al barra a cada pas, amb el mètode **setStepSize** i una posició inicial, amb el mètode **setScrollPosition**. Finalment l'afegim a la finestra arrel.

CEGUI::Checkbox

Classe que permet crear botons d'opció, com els que es poden veure a la figura 5.3.7.

Figura 5.3.7 – Exemple d'un botó d'opció de tipus Checkbox de la llibreria CEGUI

A la figura 5.3.8 es pot veure el codi necessari per generar un botó d'opció de tipus Checkbox.

```
CEGUI::Window * cCas = (CEGUI::Checkbox*)
CEGUI::WindowManager::getSingleton().createWindow
("TaharezLook/Checkbox", (CEGUI::utf8*)"cCas");
cCas->setPosition(CEGUI::UVector2(cegui_reldim(0.25f),
cegui_reldim( 0.50f)) );
cCas->setSize(CEGUI::UVector2(cegui_reldim(.05f),
cegui_reldim(.05f)) );
cCas->setSelected(true);
wRoot->addChildWindow(cCas);
```

Figura 5.3.8 – Codi necessari per generar un botó de tipus Checkbox

Com es pot veure al codi, generem un objecte de tipus **Window** i de nom **cCas**. Seguidament li assignem una posició i una mida amb els mètodes **setPosition** i **setSize** respectivament. Finalment decidim si volem que estigui marcat per defecte amb el mètode booleà **setSelected** i l'afegim a la finestra arrel.

Ara que ja coneixem els elements utilitzats a la interfície per a que l'usuari pugui entrar les seves preferències veurem com es fa per dotar de funcionalitats a tots aquests elements.

Events de CEGUI

Podem definir un event com una acció que passa dins la interfície del programa o dins el programa mateix provocada, generalment, per l'usuari. Per exemple, un event seria un click de ratolí sobre un botó, o tancar una finestra, escriure en un quadre de text o moure el ratolí en qualsevol punt de la pantalla. Tots aquests events poden ser programats per a generar la resposta que es vulgui.

Les classes encarregades de saber l'estat del ratolí i el teclat són, respectivament, OIS::MouseListener i OIS::KeyListener. D'aquesta manera podrem saber s'hi s'ha apretat una tecla o si hem fet click amb el ratolí, entre d'altres coses.

Mostrarem un exemple de com programar events en CEGUI, ja que tots es fan de manera similar. L'exemple és bastant senzill: agafem el botó de tipus PushButton que hem creat a la figura 4.3.1.1.4 i li assignarem un event que faci que la finestra a la qual pertany es tanqui quan es premi el botó.

Per a fer-ho hem d'introduïr un nou concepte, la classe FrameListener. Aquesta classe permet rebre notificacions just abans i

just després de que un fotograma s'hagi de renderitzar. Això ho fa amb dos mètodes, frameStarted i frameEnded, el primer rep notificacions abans de renderitzar i el segon després. Així, dincs el bucle principal d'Ogre, a cada iteració, fa el que es pot veure a la figura 5.3.9.

```
/*Suposem que tenim una llista amb els frameListener registrats,
anomenada "llistaFrameL"*/
int i = 0;
booleà frameS = cert;
mentre (i < llistaFrameL.mida() i frameS) fer {
     frameS = llistaFrameL[i].frameStarted();
     i++;
}
si (frameS) llavors {
     //Renderitzar l'escena
}
altrament {
     //sortir
}
i = 0;
frameS = cert;
mentre (i < llistaFrameL.mida() i frameS) fer {
     frameS = llistaFrameL[i].frameEnded();
     i++;
}
si (frameS) llavors {
     //repetir tot el procés amb el següent fotograma
}
altrament {
     //sortir
}
```

Figura 5.3.9 – Codi corresponent bucle principal d'Ogre.

Com es pot veure al codi, primer, crida **frameStarted** per tots els **FrameListener** registrats. Si tots han retornat cert renderitza el fotograma. Finalment crida **frameEnded** per tots els FrameListener registrats. Aquest bucle només acaba si algun FrameStarted o FrameListener retorna fals. Aquí és on intervé l'event que hem de programar, ha de fer que frameStarted retorni fals.

Per a fer-ho hem creat un FrameListener propi que sobreescriurà el mètode frameStarted, del qual es pot veure el codi a la figura 5.3.10.

```
mètode frameStarted(FrameEvent &evt) ret booleà{
     retorna mContinuar;
}
```

Figura 5.3.10 – Codi corresponent al mètode frameStarted.

Hem declarat un mètode frameStarted que no fa res, només retornar el valor de l'atribut booleà **mContinuar**. D'aquesta només haurem de fer que aquesta variable valgui fals per a que surti del programa.

A la figura 5.3.11 es pot veure el codi necessari per programar l'event.

```
//sort es el botó de sortida que hem creat a la figura 4.3.1.1.4
sort->subscribeEvent(CEGUI::PushButton::EventClicked,
CEGUI::Event::Subscriber(&quit, this));
```

Figura 5.3.11 – Codi necessari per programar l'event.

Com es pot veure al codi, li estem dient que quant es faci click al botó, vagi al mètode **quit**, que es pot veure a la figura 5.3.12.

```
mètode quit(const CEGUI::EventArgs &e) ret booleà {
    mContinuar = fals;
    retorna cert;
}
```

Figura 5.3.12 – Codi corresponent al mètode quit.

Com es pot veure al codi, modifiquem la variable **mContinuar** fent que valgui cert, de manera que sortirem del bucle en clickar el botó.

5.4.- Altres aspectes de la implementació

Tal i com hem mencionat a l'apartat 4.3.3.7, de la classe ListenerLlums, Ogre permet implementar un Listener que decideixi quines llums iluminen un MovableObject determinat.
El gran problema al que ens hem hagut d'enfronat ha estat que les malles que hem fet servir al projecte són carregades com objectes Entity (que hereta de MovableObject) i, per tant, només podríem posar-li un Listener a tota la Mesh. D'aquesta manera ens perdríem gran part de la gràcia de l'algoritme de lightcuts, ja que no podríem avaluar tots els punts de l'escena.
Per a solucionar-ho, a l'hora de carregar una Mesh, enlloc de fer-ho de manera convencional generarem tantes Entity com polígons tingui aquesta Mesh. D'aquesta manera, si tenim una Mesh amb 500 polígons, enlloc de carregar-la com una sola Entity, la carreguem com 500 Entities, de manera que a cada Entity se li pugui assignar un objecte Listener.
Això és bastant ineficient i incorpora una constant que redueix els fotogrames per segon de l'escena, però com que a l'hora de fer comparacions introduïrem sempre el mateix error, no ens ha d'importar.
El mètode encarregat de dividir la Mesh en Entities s'anomena separarTriangles, i ha estat escrit per en Santiago Barroso Juan, que estudia Enginyeria Tècnica Informàtica de Sistemes.

6.- Anàlisi dels resultats

En aquest capítol presentarem els resultats obtinguts utilitzant l'algoritme de lightcuts. Farem una comparació d'una mateixa escena utilitzant i no utilitzant l'algoritme per veure quina millora es pot apreciar.

Escena senzilla

Primerament montarem una escena molt senzilla, que constarà de 10 llums i 10 objectes plans, de 3 colors diferents. Això ens permetrà mostrar-ne un cut de llums real per un dels objectes. En aquest capítol anomenarem a aquesta escena "escena senzilla" ja que posteriorment es veurà una escena més complexa i carregada. A la figura 6.1 es pot apreciar l'aspecte que té "l'escena senzilla".

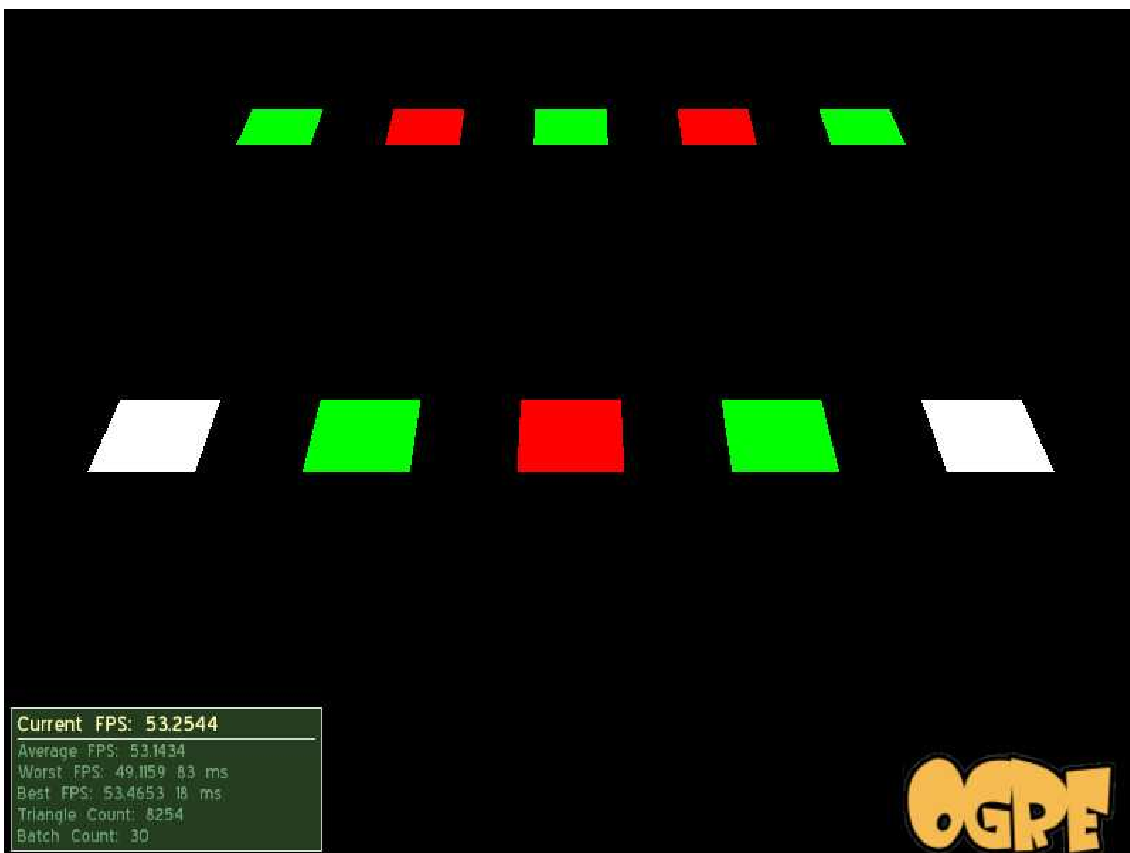


Figura 6.1 – Escena senzilla

Com es pot apreciar a la figura 6.1 aquesta escena consta de 2 fileres de 5 plans cadascuna, tots a la mateixa alçada i vists amb certa perspectiva. A continuació mostrarem dues imatges a la figura 6.2, una de l'escena sense utilitzar l'algoritme de lightcuts i l'altra de la mateixa escena utilitzant-lo amb un percentatge d'error del 15%. Aquest percentatge és molt elevat, però com que es tracta d'una escena amb colors molt simples es fa encara més difícil apreciar l'error, com es pot comprovar.

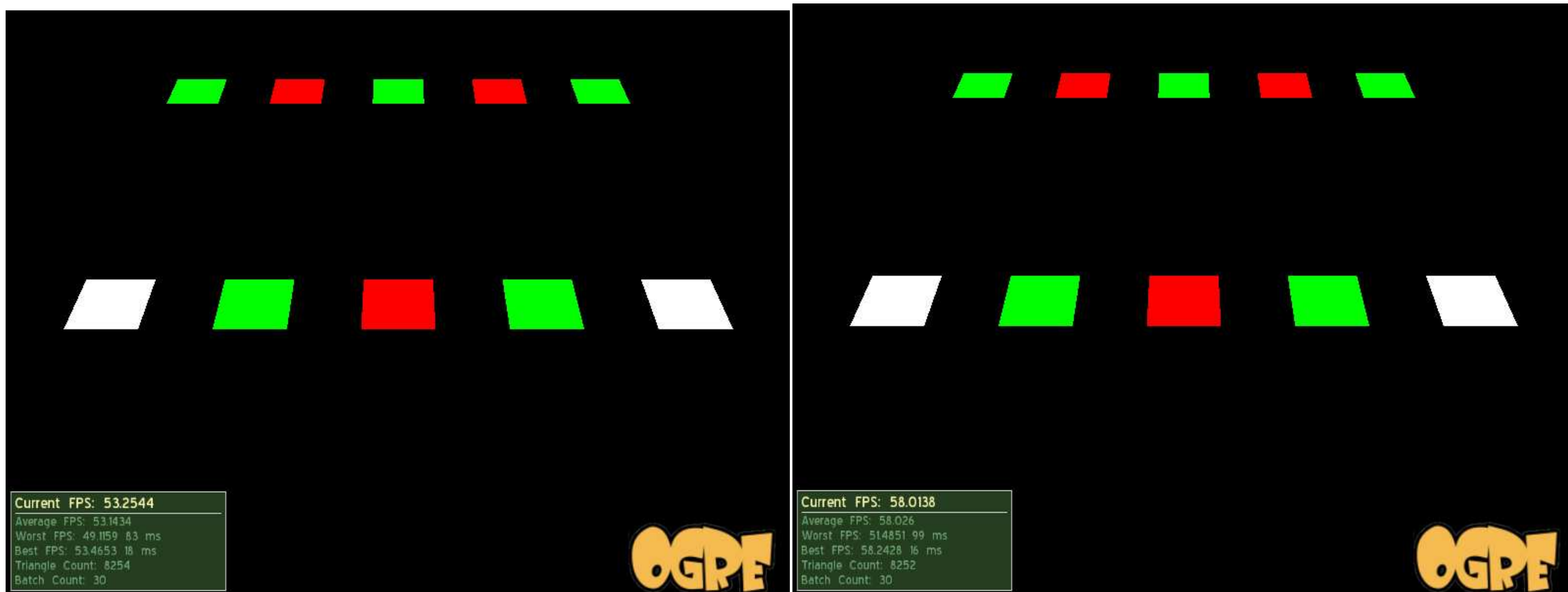


Figura 6.2 – Comparació de les dues escenes senzilles. A l’esquerra, no s’està utilitzant l’algoritme de lightcuts. A la dreta, s’està utilitzant amb un marge d’error del 15%.

Tal i com acabem de comentar, no es pot veure la diferència entre les dues imatges. En canvi, es pot apreciar com la imatge de la dreta, la que fa servir l'algoritme lightcuts, té una mitjana de 5 fotogrames per segon més que la de l'esquerra, que no fa servir lightcuts (es pot veure al quadre de diàleg inferior esquerre, a l'apartat Average FPS). Tenen 53.14 fotogrames per segon i 58.02 fps, respectivament.

A continuació mostrarem l'arbre de llums que s'ha generat per aquesta escena. El nom de cada llum és un número de 0 a 9 i l'arbre generat es pot veure a la figura 6.3.

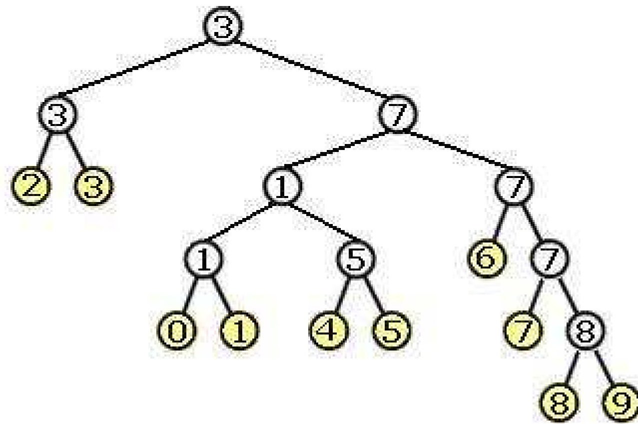


Figura 6.3 – Arbre de llums per l'escena senzilla

Els nodes pintats de groc són nodes arrel, és a dir, llums puntuals. A continuació mostrarem el cut per un dels plans que surten a la figura 6.2 de la dreta, més exactament pel pla de color blanc de la filera de davant a la dreta. El lightcut d'aquest pla es pot veure a la figura 6.4.

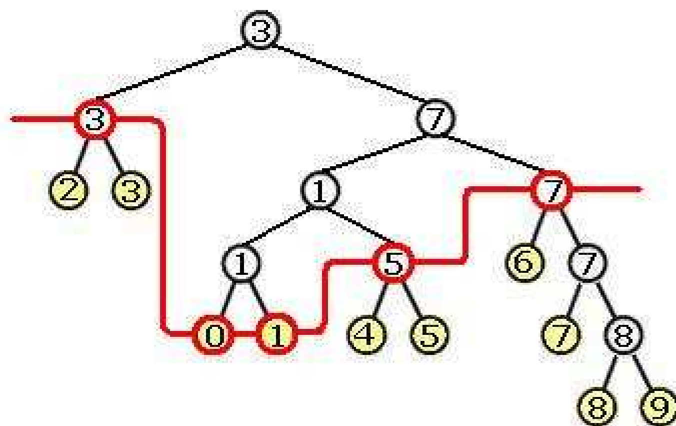


Figura 6.4 – Arbre de llums per l'escena senzilla amb el lightcut corresponent al pla blanc de la dreta

Com es pot comprovar a la figura, aquest cut consta de 5 nodes, 3 són nodes grup (3, 5 i 7) i 2 són nodes arrel (0 i 1).

Escena complexa

A continuació montarem una escena amb més carrega poligonal per seguir posant a prova l'algoritme de lightcut. L'escena consta d'un pla que fa de terra, una malla en forma d'avió, una altra en forma de bidó i dues malles en forma de palet de fusta. El nombre de llums segueix siguent de 10. A la figura 6.5 podem veure l'aspecte de "l'escena complexa" amb els objectes ja mencionats.

Figura 6.5 – Escena complexa

L'escena funciona a uns 27.8 fps. A continuació compararem l'escena sense algoritme de lightcuts amb una escena que si que el fa servir, amb un marge d'error del 10%. Aquestes dues imatges es poden veure a la figura 6.6.

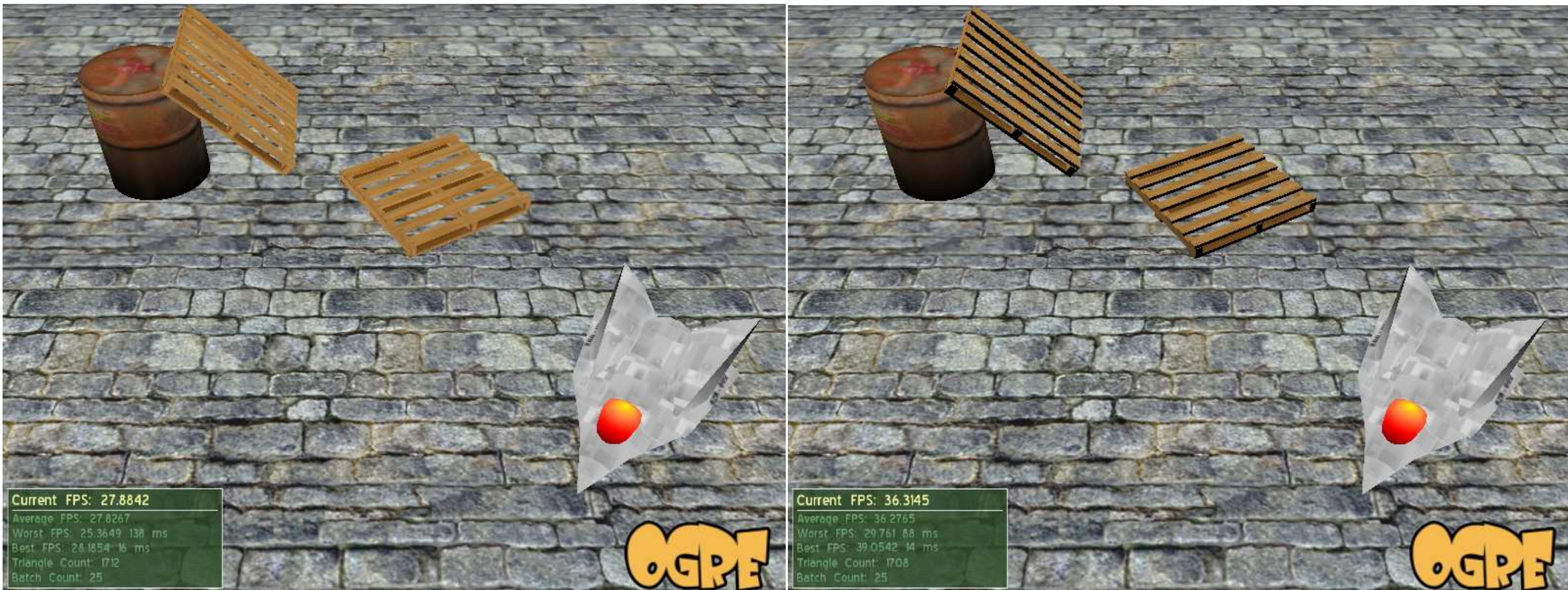


Figura 6.6 - Comparació de dues escenes complexes. A l'esquerra, no s'està utilitzant l'algoritme de lightcuts. A la dreta, s'està utilitzant amb un marge d'error del 10%.

Tal i com es pot veure a la figura 6.6 a l'escena on no fem servir lightcuts funciona a 27.8 fps mentre que a la que si que fem servir lightcuts funciona a 36.3 fps, això vol dir que l'escena que fa funcionar lightcuts funciona a 8.5 fps més ràpid. Com ja hem comentat a l'apartat 5.4, per poder renderitzar malles en Ogre hem hagut d'introduir una constant que ens redueix bastant els fotogrames per segon.

Es poden apreciar lleugeres diferències entre les dues escenes, sobretot el als dos palets de fusta, que es veuen més foscos a l'imatge de la dreta i també el bidó es veu una mica més fosc a la dreta. Tot i així la qualitat de la imatge és molt elevada i és difícil apreciar l'error entre una escena i una altra. Això fa pensar que, a l'hora de fer servir aquest algoritme en videojocs, on la càrrega de la CPU és molt elevada, seria factible elevar encara més el percentatge d'error per obtenir millor rendiment sacrificant una mica la qualitat de la imatge.

6.1.- Comparació percentatges d'error

Comprovarem en una escena de proves com afecta el percentatge d'error escollit en el nombre de llums que utilitzem a l'escena.

Hem fet servir la mateixa escena de l'apartat d'anàlisi dels resultats, amb 10 llums i 10 objectes (plans, evaluats cadascun com un sol punt) i veurem com, variant el percentatge d'error introduït, el lightcut escollit per cada punt de l'escena canviarà.

Utilitzarem la mitjana de llums per objecte per guiar-nos a l'hora de saber com de refinat està l'arbre de llums. El seu càlcul és ben senzill, es sumen les llums de cada lightcut per objecte i es divideix entre el nombre total d'objectes.

A la figura 6.1.1 es pot veure l'escena sense fer servir l'algoritme de lightcuts.

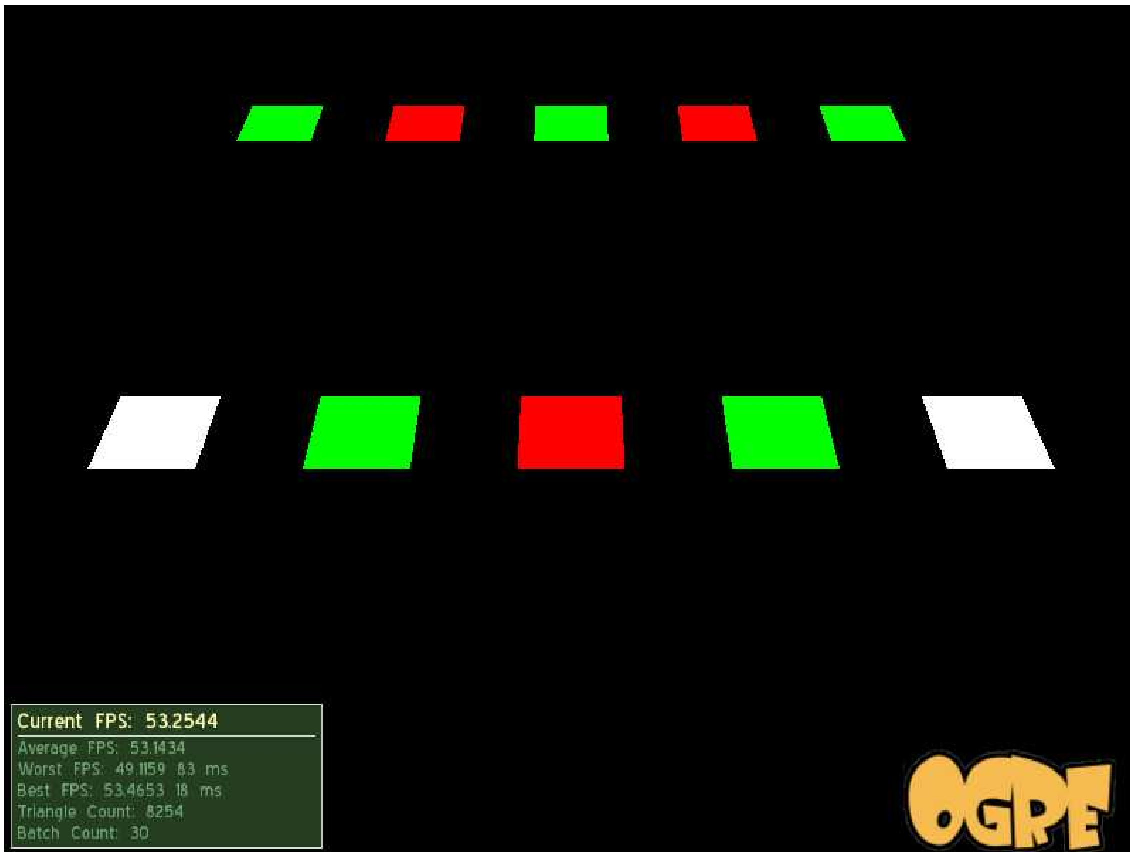


Figura 6.1.1 – Escena sense utilitzar algoritme de lightcuts

Com es pot veure es tracta de 10 plans que tenen 3 colors diferents. Les 10 llums estan totes per sobre dels plans.

A continuació veurem la mateixa imatge utilitzant l'algoritme de lightcuts introduïnt un error de l'1%. Es pot veure a la figura 6.1.2.

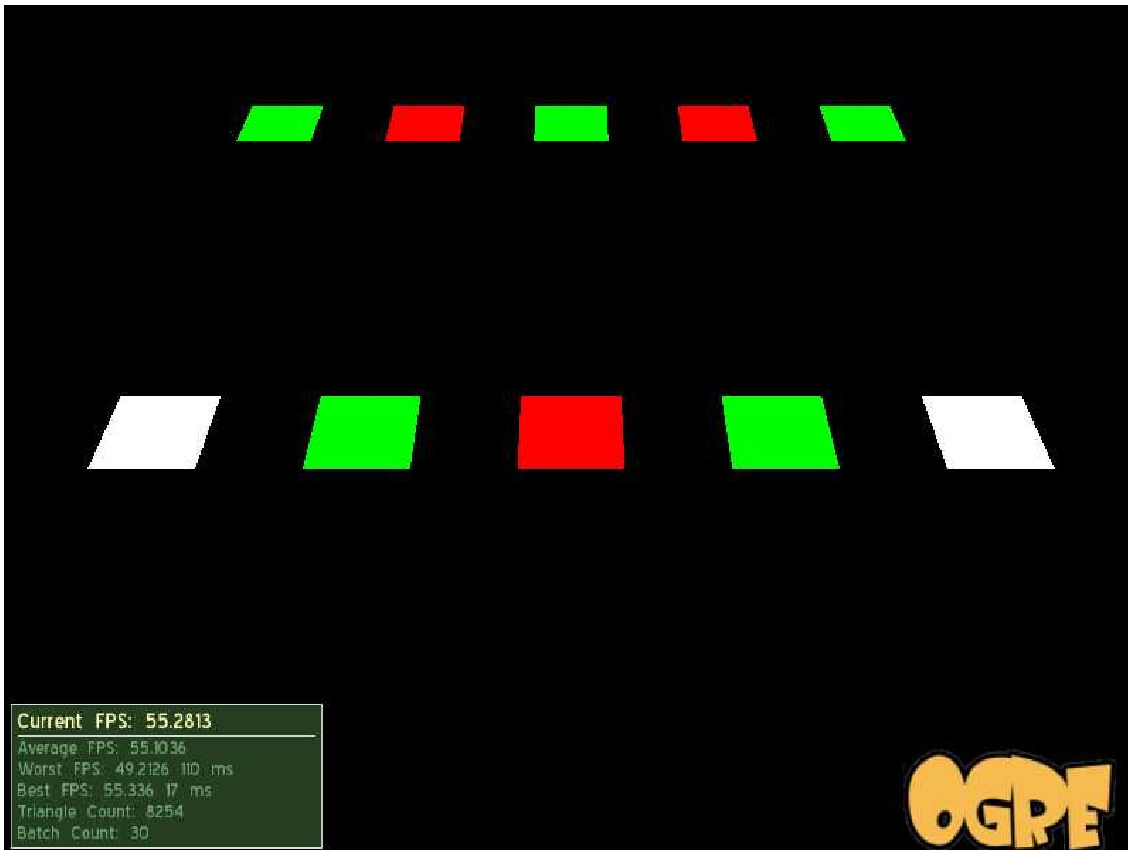


Figura 6.1.2 – Escena utilitzant l'algoritme de lightcuts amb un error de l'1%

Com que es tracta d'objectes tan senzills i amb colors plans no es pot apreciar cap error. Tot i així, la mitjana de llums per objecte en aquesta escena és de 8.9. Això vol dir que, de 10 llums que hi ha a l'escena, n'utilitzem unes 8.9 per iluminar cada objecte. Si augmentem el percentatge no es pot apreciar cap error fins als voltants del 20%. Si, per exemple, pugem el percentatge d'error fins al 15%, el resultat es pot veure a la figura 6.1.3.

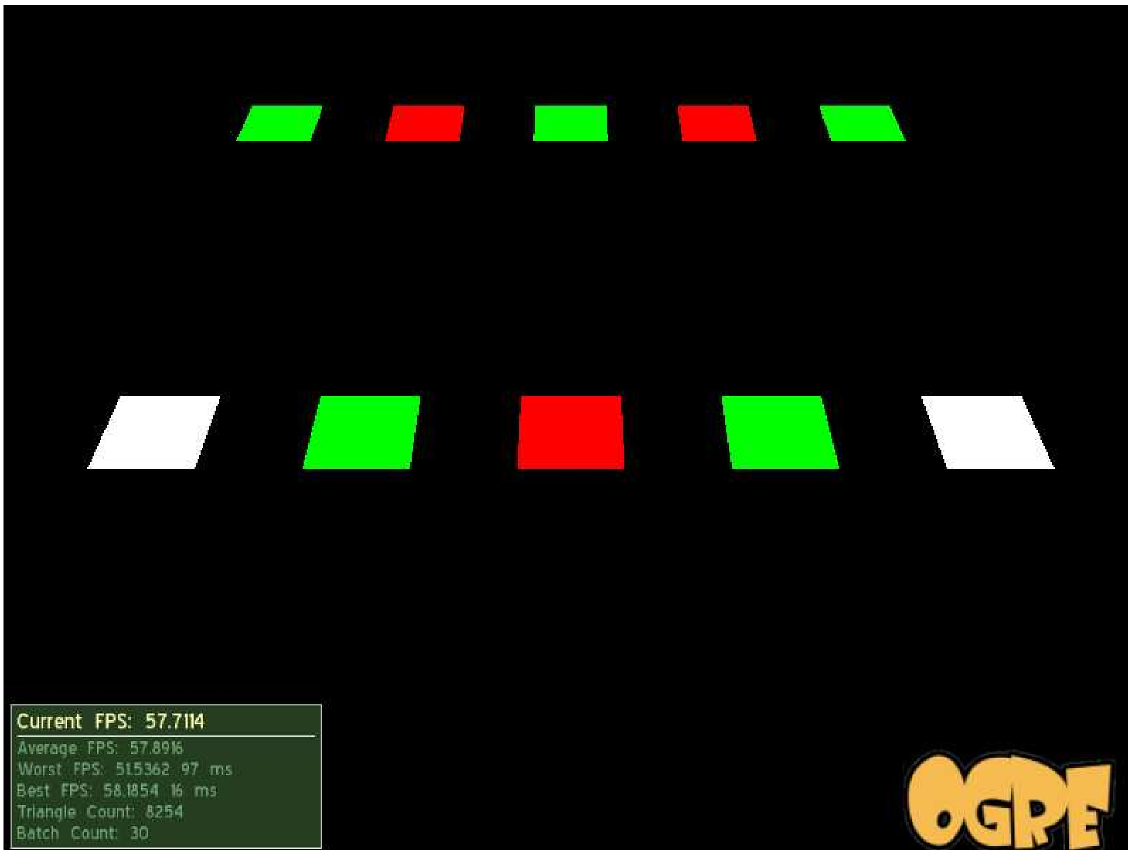


Figura 6.1.3 – Escena utilitzant l'algoritme de lightcuts amb un error del 15%

Com es pot comprovar en aquesta imatge encara es veu exactament igual. La mitjana de llums per objecte ara és de 4.4 i tot i així encara no es pot apreciar cap error). Augmentarem el percentatge d'error fins al 25%, el resultat es pot veure a la figura 6.1.4.

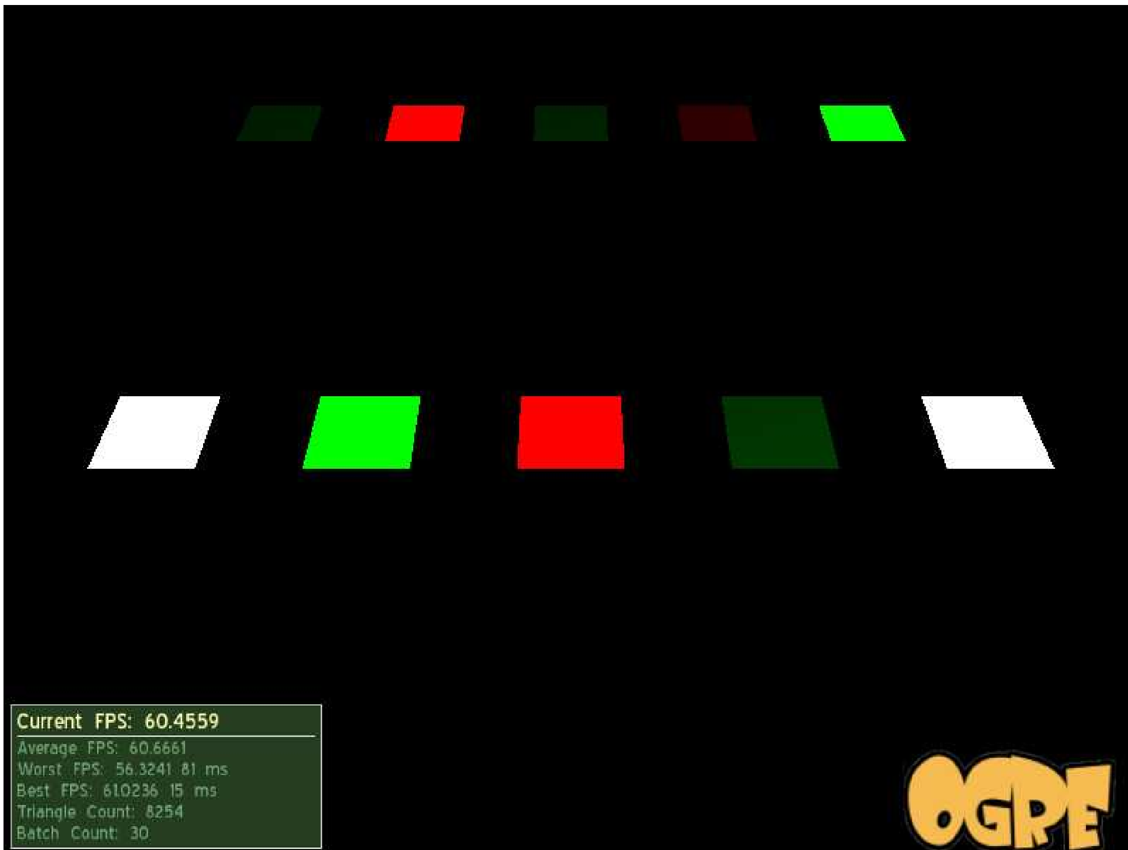


Figura 6.1.4 – Escena utilitzant l'algoritme de lightcuts amb un error del 25%

Aquí si que es pot apreciar un cert error a tres dels plans situats a la fila de darrera. Estan menys iluminats, degut a que hem utilitzat menys llums. La mitjana de llums per objecte és de 2.2.

Tot i que hem anat augmentat el percentatge d'error fins a un nombre considerablement alt la imatge pràcticament no ha canviat. A la figura 6.1.5 es pot veure una taula amb els percentatges d'error, el nombre de llums per objecte i la mitjana de fotogrames per segon en cada cas.

Percentatge d’error	Mitjana llum/objecte	Fotogrames/segon
Sense utilitzar lightcuts	-	53.1 fps
1%	8.9	55.1 fps
3%	7.9	55.9 fps
5%	6.6	57.1 fps
10%	5.2	57.4 fps
15%	4.4	58.0 fps
20%	2.5	59.9 fps
25%	2.2	60.6 fps

Figura 6.1.1 – Taula de comparació de percentatges.

Com podem comprovar a la taula, a mesura que augmentem el percentatge d’error el nombre mig de llums utilitzat per cada objecte va disminuint i els fotogrames per segon van augmentant. A més, només introduïnt un percentatge d’error de l’1% ja aconseguim una millora de 2 fotogrames per segon.

6.2.- Problemes trobats

Un dels primers problemes al que ens enfrontàvem era descobrir com se li podia indicar a l’Ogre quines llums iluminen un objecte determinat. Això ens va endarrerir bastant i, tot i poder avançar en altres temes, aquesta era una part molt important a solucionar. La solució al problema va arribar gràcies a la comunitat d’Ogre a la xarxa, més en concret dels fòrums oficials d’Ogre.

Un altre problema, derivat d’aquest primer, era com poder dividir una Mesh en diferents Entitys, per poder iluminar cada polígon. Això es va solucionar gràcies a l’ajuda d’un estudiant, tal i com s’ha comentat a l’apartat 5.4.

Tot i que aquests dos problemes mencionats han estat bastant importants, un tercer problema ha estat l’aprenentatge i l’utilització d’Ogre3D i CEGUI. Al llarg de la carrera se’ns ha donat nocions de programació gràfica són moltes les coses que s’han après sobre la marxa i s’ha hagut d’improvitzar bastant.

7.- Conclusions

En aquest capítol comentarem les conclusions extretes del desenvolupament d’aquest projecte. Es farà referència al compliment dels requeriments que s’havien definit previament i als objectius d’aprenentatge.

L’objectiu principal d’aquest projecte consistia en implementar l’algoritme de Lightcuts i montar una escena per provar-lo. Aquest objectiu, com s’ha pogut comprovar al llarg de la memòria, ha estat complert de forma més que satisfactòria. És cert, però, que han sorgit alguns problemes, ja comentats a l’apartat 6.2, que han impedit que el resultat del projecte hagi estat excelent, i és per això que al capítol 8 parlem de les millores que es podríen fer en el futur.

Pel que fa als objectius personals i d’aprenentatge es pot dir que han estat complerts de forma més que brillant i s’ha aconseguit ampliar coneixements de: llenguatge C++, Programació orientada a objectes, Microsoft Visual C++, ús d’eines de l’Enginyeria del Software com Visio 2007, etc. A més s’ha après en certa profunditat com utilitzar un motor gràfic bastant potent, complexe i utilitzat en el món real, com és l’Ogre3D i utilitzar-ne una interfície gràfica, CEGUI. Finalment, però no menys important, s’ha après a escriure una documentació de caire seriós, tema que no havíem tocat massa al llarg de la carrera.

8.- Treball futur

Com ja s'ha mencionat a l'apartat 5.4, per poder fer servir el Listener d'Ogre hem hagut de dividir manualment les Meshes utilitzades per tenir tantes Entity com polígons a la malla. Això ha introduït una constant que ha fet baixar els fotogrames per segon de l'escena resultant.

Una possible solució és la de fusionar aquest algoritme amb l'algoritme de Deferred Shading. Aquest algoritme és una tècnica de shading que divideix en vàries parts el resultat de l'algoritme de shading i guarda aquestes parts a búffers intermitjos. Així aquests búffers només són llegits quan es necessiten i estalviem temps i recursos. I és dins d'aquests búffers on trobem la informació de tots els vèrtexs que composen l'escena abans de que es pinti per pantalla, de manera que ja tindríem les dades necessàries per aplicar l'algoritme de lightcut. Això comportaria estudiar a fons aquest algoritme i veure en quin punt podem introduir-hi el nostre codi.

9.- Agraïments

En aquest capítol vull agraïr personalment l'ajuda que m'han donat determinades persones o col·lectius.

Primerament vull agraïr l'ajuda i el suport que m'ha donat durant tot el temps que ha durat el desenvolupament d'aquest projecte al director del meu projecte, en Gustavo Ariel Patow.

També vull agraïr a l'estudiant l'Ismael García Fernández l'ajuda que em va prestar per entendre el sistema d'iluminació d'Ogre.

Agraïr a en Santiago Barroso Juan que em deixés utilitzar part del seu codi necessari per dividir malles en entitats d'Ogre.

Finalment, agraïr a la comunitat virtual existent als fòrums d'Ogre, que han respòs a gran quantitat de dubtes que els he anat formulant al llarg del projecte.

10.- Referències Bibliografíques

En aquesta secció es troben les referències bibliogràfiques mencionades al llarg de la memòria.

Llegenda	Referència
Walter 2005	WALTER, B. 2005. Lightcuts: A scalable Approach to Illumination, Cornell University, New York.
Ward 1994	WARD, G. 1994. Adaptive shadow testing for ray tracing. In Photorealis- tic Rendering in Computer Graphics (Proceedings of the Second Euro- graphics Workshop on Rendering), Springer-Verlag, New York, 11–20.
Shirley	SHIRLEY, P., WANG, C., AND ZIMMERMAN, K. 1996. Monte carlo tech- niques for direct lighting calculations. ACM Transactions on Graphics 15, 1 (Jan.), 1–36.
Paquette 1998	PAQUETTE, E., POULIN, P., AND DRETTAKIS, G. 1998. A light hierarchy for fast rendering of scenes with many lights. Computer Graphics Forum 17, 3, 63–74.
Fernandez 2002	FERNANDEZ, S., BALA, K., AND GREENBERG, D. P. 2002. Local il- lumination environments for direct lighting acceleration. In Rendering Techniques 2002: 13th Eurographics Workshop on Rendering, 7–14.
Wald 2003	WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2003. Interactive global illumination in complex and highly occluded environments. In Euro- graphics Symposium on Rendering: 14th Eurographics Workshop on Rendering, 74–81.